DB2® Information Management Software

XML for DB2 Information Integration

Marrying XML documents and databases

Scenarios of XML usage

Using IBM WebSphere Studio Application Developer to build Web Services and XML

Bart Steegmans
Ronald Bourret
Owen Cline
Olivier Guyennet
Shrinivas Kulkarni
Stephen Priestley
Valeriy Sylenko
Ueli Wahli

Redbooks

ibm.com/redbooks

IBM

International Technical Support Organization

XML for DB2 Information Integration

July 2004

SG24-6994-00

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

First Edition (July 2004)

This edition applies to V8.1 FixPak 2 of DB2 UDB for Windows 2000/NT and V5.1 of WebSphere Studio Application Developer.

Contents

Figures

Tables

Examples

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

eserver®
eserver®
Redbooks (logo) ™
alphaWorks®
ibm.com®
iSeries™
z/OS®
zSeries®
AIX®
Cloudscape™
DB2 Universal Database™
DB2®
DRDA®
Informix®
IBM®
IMS™
Lotus®
MQSeries®
Notes®
OS/2®
OS/390®
Redbooks™
Redbooks (logo)™
S/390®
Tivoli®
TME®
WebSphere®

The following terms are trademarks of other companies:

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Intel Inside (logos), MMX, and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Preface

In many organizations, relational databases are the backbone for data storage and retrieval. Over the last couple of years, XML has become the de facto standard to exchange information between organizations, as well as between departments or applications within the same organization.
Since data tends to live in databases, it needs to be converted from a relational format into an XML format when involved in those data exchanges, as well as converted (back) from XML into a relational format for storage, or for handling by other applications.

How can we achieve this?

This IBM® Redbook describes how to design the mapping between XML and relational data, and vice versa, to enable a flexible exchange of information.

IBM provides a number of products to help you bridge the gap between XML and its relational database, DB2®. The DB2 engine itself provides support to generate XML fragments from relational data through the use of SQL/XML built-in functions. DB2 also provides the DB2 XML Extender. It allows you to perform XML composition, like SQL/XML, but also provides functionality to decompose XML documents, as well as store XML documents intact inside the database. In addition, XML Extender provides a set of transformation and validation functions. Another option to work with XML is to use the XML wrapper that is part of the set of non-relational wrappers of DB2 Information Integrator.

This redbook also looks at the IBM tools available to assist you when dealing with XML, more specifically WebSphere® Application Developer and DB2 Control Center.

To give the whole discussion a more practical angle, the use of these functions and products is illustrated through the development of a simple application.

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, San Jose Center.

Bart Steegmans is a DB2 Product Support Specialist from IBM Belgium currently on assignment at the ITSO in San Jose. He has over 15 years of experience in DB2. Before joining IBM in 1997, Bart worked as a DB2 system administrator within a banking and insurance group. His areas of expertise include DB2 performance, database administration, and backup and recovery.

Ronald Bourret is a freelance programmer, technical writer, and researcher. His work includes XML-DBMS, a set of Java™ packages for transferring data between XML documents and relational databases, an XML schema language (DDML), several widely read papers on XML and databases, and the XML Namespaces FAQ. He has lectured on XML and databases at both commercial and academic conferences and has contributed articles to both XML.com and xmlhack.

Owen Cline is a member of the IBM Software Services for the WebSphere team based in San Diego, CA. He earned a BS in Computer Science from the University of Pittsburgh and an MS from San Diego State University. He has over 20 years of experience in the software development field. He holds four software patents, has written IBM Redbooks™ and has presented at multiple technical conferences. For the past five years, Owen has specialized in J2EE application development and deployment with a special emphasis on the WebSphere platform. He has also worked on many high-profile Web sites over the past few years.

Olivier Guyennet is a Data Management IT Engineer at IBM Japan Systems Engineering with three years of experience in the DB2 family of products. He is an IBM Certified Solutions Expert on DB2 UDB Database Administration and provides technical support for DB2 UDB on UNIX® and Intel® platforms. His areas of expertise include DB2 UDB system administration as well as perfomance tuning, DB2 XML Extender and DB2 Content Manager system administration.

Shrinivas Kulkarni is a Software Architect working in IBM India Software Labs in Bangalore. His areas of interest include Business Integration, XML, software architecture and component-based solutions. He specializes in Object Technology and is an IBM certified OO analyst and designer. Shrinivas has been working with IBM for four and a half years. He holds a graduate degree in Computer Science from the University of Mysore.

Stephen Priestley is a Business Intelligence Technical Specialist based in Melbourne, Australia, with six years of experience within the IBM Software

Group. Prior to joining IBM, Stephen was a data architect with Dun and Bradstreet Information Services, where he worked on data enhancement and expansion projects. Stephen has been involved in many business intelligence and data management projects with many of the largest organizations in Australia. He has extensive experience in front-end BI Tools, OLAP, ETL tools and databases. Stephen's work with IBM makes him a regular presenter at IBM conferences.

Valeriy Sylenko is a Lead Software Developer for The Lowe-Martin Group, Ottawa, Ontario, Canada. He has eight years of experience in the field of information technology. Valeriy is a Sun Certified Programmer for the Java 2 Platform 1.4 and holds a PhD degree in Physics and Mathematics from the Institute of Semiconductor Physics, Ukraine, Kiev. His areas of expertise include object-oriented analysis and design, data modeling and J2EE application development using WebSphere Application Developer and WebSphere Application Server.

Ueli Wahli is a Consultant IT Specialist at the IBM International Technical Support Organization in San Jose, California. Before joining the ITSO 19 years ago, Ueli worked in technical support at IBM Switzerland. He writes extensively and teaches IBM classes worldwide on application development, object technology, WebSphere Application Server, and lately, WebSphere Studio products. Ueli holds a degree in Mathematics from the Swiss Federal Institute of Technology.

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

Cecilia Bardy
International Technical Support Organization, Raleigh Center

Emma Jacobs
International Technical Support Organization, San Jose Center

Yannick Barel
David Bertoni
Martin Breining
Seeling Cheung
Chris Holstrom
Susan Malaika
Dung Nguyen
Matthias Nicola
Mayank Pradhan
Angel Reyda
Hardeep Singh
Dave Salinero
Singgih Tanumihardjo
Kyla Town
Dirk Wollscheid
Patsy Yu
IBM Silicon Valley Laboratories, San Jose

Morgan Tong
IBM Austin

Jasmi John
Ernest Mah
Shili S Yang
IBM Toronto Lab, Canada

Tim Brown
IBM UK

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'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll 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 Internet note to:

  redbook@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. QXXE Building 80-E2
  650 Harry Road
  San Jose, California 95120-6099

Part 1

Overview

1

XML and databases

This chapter describes the following topics:

- Reasons to use XML with a database
- Whether XML is itself a database
- How XML is used with databases
- XML-enabled databases
- Native XML databases

1.1 Why use XML with a database?

There are a number of reasons to use XML with a database. Perhaps the most common of these is to publish data stored in an existing database as XML. For example, suppose a database contains information about stock prices. A Web service might return the current price of a stock as an XML document (Figure 1-1).

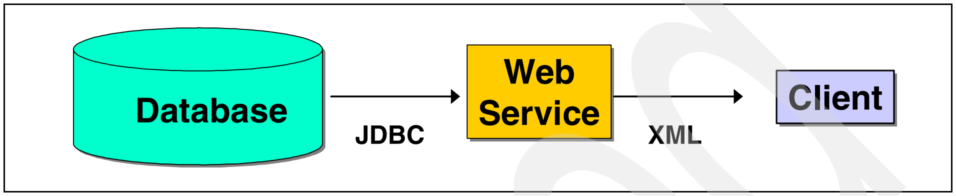


Figure 1-1 Publishing relational data as XML through a Web service

Similarly, data may be transferred from an application to a database in the form of an XML document. For example, an insurance agency may submit data for a new insurance policy as an XML document. As part of processing this document, an application within the insurance company extracts this data from the XML document and stores it in a database (Figure 1-2).

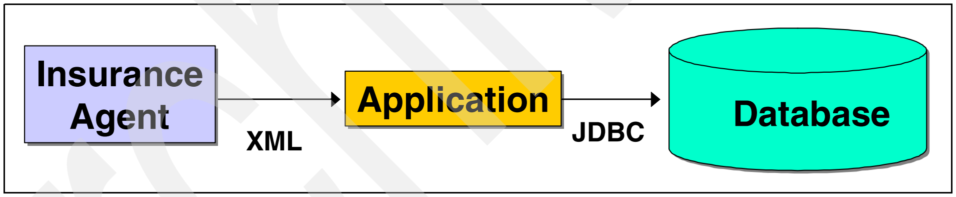


Figure 1-2 Storing data from an XML document in a database

Another use of XML (and one that is increasingly popular) is to model semi-structured data, especially in the life sciences area. One of the primary advantages of XML in this case is its extensibility. Because life sciences are such a rapidly growing field, it is impossible to predict all the kinds of data that may be available in the near future. Therefore, trying to determine a relational schema for life sciences data is an exercise in planned obsolescence. Furthermore, the volumes of data in this field are enormous, dictating the need for a database to store, manage, and query data in the form of XML documents.

Note: Semi-structured data has some structure, but the structure is not highly regular. For example, the white pages of a telephone book are structured; each entry has a name, an optional address, and a telephone number. Yellow pages are semi-structured. As in the white pages, each entry has a name, an optional address, and a telephone number, but entries might also have items such as fax numbers, maps, descriptions of services available, and license numbers. Thus, while it is possible to describe most item categories found in entries of the yellow pages, there may be considerable variation in how items use a given entry.

The XML documents in the previous examples are *data-centric*. That is, the XML documents generally contain discrete pieces of data with a fairly regular structure.
Another class of XML documents is that of *document-centric* documents. Examples of these are end-user documentation, marketing brochures, and Web pages. Document-centric documents are characterized by mixed content and irregular structure.

Note: *Mixed content* is a mixture of child text and elements. For example, the <p> element in XHTML has mixed content:

```
<p>This is an example of <i>mixed</i> content.</p>
```

In contrast to mixed content is *element content,* where an element's content consists of nothing but child elements, and *PCDATA-only content,* where an element's content is a single piece of text. Both are shown in the following fragment:

```
<Name>                  <!-- <Name> has element content -->
   <First>John</First>  <!-- <First> has PCDATA-only content -->
   <Last>Doe</Last>     <!-- <Last> has PCDATA-only content -->
</Name>
```

The Extensible HyperText Markup Language (XHTML) is a family of document types and modules that reproduce, subset, and extend HTML, but reformulate it in XML. XHTML document types are XML-based. XHTML can be seen as the successor of HTML.

The most common reason to use a database with document-centric documents is to manage and query them. For example, the amount of end-user documentation for large projects (such as for a commercial aircraft) can be enormous and being able to manage these documents in a database is critical. The advantage of using XML in this case is that it is easy to build new documents from fragments of existing documents, as well as to perform structured queries over the contents of the documents (Figure 1-3 on page 6).

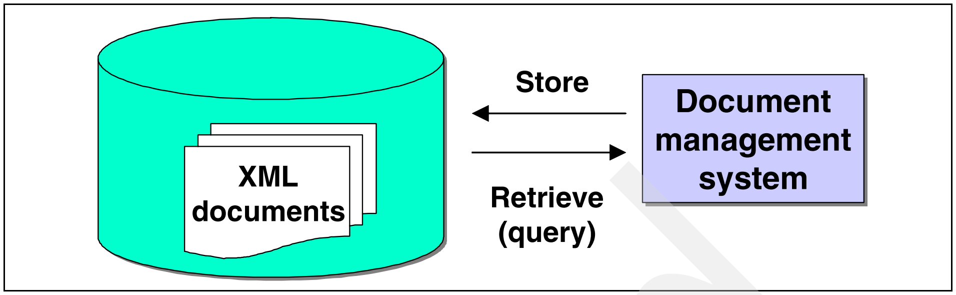


Figure 1-3 Managing XML documents in a database

A final example of using XML with databases is archiving XML documents, whether data-centric or document-centric. For example, suppose a stock brokerage company receives transaction requests from the Web in the form of XML documents. If the brokerage is required to retain these requests for a certain period of time, archiving them in a database increases their value, since they are secure and can be easily queried and retrieved.

1.2 A common misconception

Before we describe how XML interacts with databases, we need to dispel a common misconception, which holds that XML can be used directly as a database. This misconception arises from the observation that XML documents are a natural format for storing data. The question is, is this a good idea?

That is, *should you use XML as a database?*

There are actually several arguments in favor of this. The hierarchical model used by XML is ideal for describing a wide variety of data, XML's use of Unicode makes XML documents portable across many systems, and the text format used by XML makes XML documents easily readable by humans. In addition, the XML family includes schema languages (XML Schemas, DTDs), query languages (XQuery, XPath), APIs (SAX, DOM, JDOM), and many other technologies useful in a database setting.

> **Note:** By convention, the term *XML Schemas* (with an upper-case S) refers to the XML Schema language defined by the W3C. The term *XML schemas* (with a lower-case s) refers to an XML schema represented in any language, including DTDs and XML Schemas.

In spite of this, there are many more arguments against using XML as a database. The most important of these is that XML is an inefficient storage format. It is verbose and the need to parse it means that any data access will be slow, even for modestly large documents. But even if these problems could be solved, the XML family lacks many of the technologies commonly found in modern databases, such as indexes, transactions, multi-user access, security, logging, referential integrity, triggers, and so on.

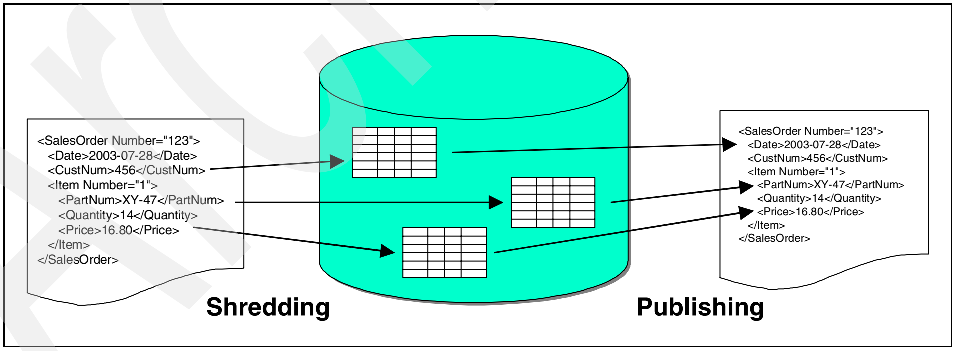
As a result, using XML documents and surrounding technologies as a database means that you will need to write a lot of the code already found in modern databases. So while it might make sense to use XML for small, single-user databases (configuration files, small contact lists, and so on) most production environments require a real database.

1.3 How is XML used with databases?

XML is used with databases in two ways:

- XML documents are used to exchange data between a database and an application or another database. For example, suppose a relational database contains information about stock prices. A Web service can return the current price of a stock as an XML document.

  The process of extracting data from a database and constructing an XML document (or XML fragments) is known as *publishing* or *composition*. The reverse process (extracting data from an XML document (or XML fragment) and storing it in the database) is known as *shredding* or *decomposition*. These processes are shown in Figure 1-4.



```
<SalesOrder Number="123">
  <Date>2003-07-28</Date>
  <CustNum>456</CustNum>
  <Item Number="1">
    <PartNum>XY-47</PartNum>
    <Quantity>14</Quantity>
    <Price>16.80</Price>
  </Item>
</SalesOrder>
```

```
<SalesOrder Number="123">
  <Date>2003-07-28</Date>
  <CustNum>456</CustNum>
  <Item Number="1">
    <PartNum>XY-47</PartNum>
    <Quantity>14</Quantity>
    <Price>16.80</Price>
  </Item>
</SalesOrder>
```

Figure 1-4 Shredding and publishing XML documents

- XML documents are stored in a database. For example, end-user documentation written in DocBook (an XML dialect) can be managed in a database, as shown in Figure 1-5 on page 8.

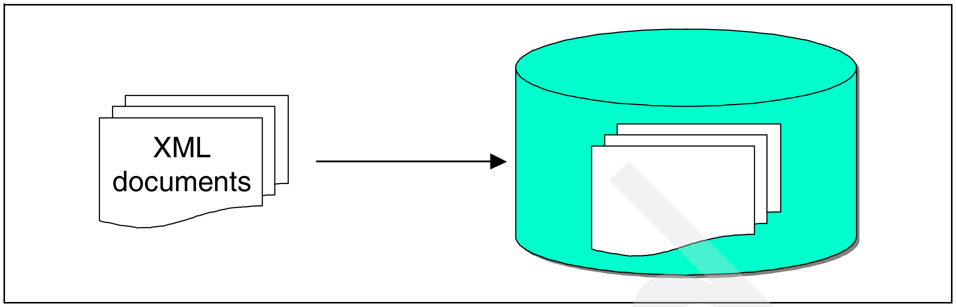


Figure 1-5 Storing XML documents in a database

To see the difference between these two uses, we will look "inside" the database and see what the stored data looks like. Consider the different ways to store the XML sales order document of Example 1-1.

Example 1-1 Sample sales order

```
<SalesOrder Number="123">
   <OrderDate>2003-07-28</OrderDate>
   <CustomerNumber>456</CustomerNumber>
   <Item Number="1">
      <PartNumber>XY-47</PartNumber>
      <Quantity>14</Quantity>
      <Price>16.80</Price>
   </Item>
   <Item Number="2">
      <PartNumber>B-987</PartNumber>
      <Quantity>6</Quantity>
      <Price>2.34</Price>
   </Item>
</SalesOrder>
```

If this XML document is used only as a way to exchange data, we can shred it into the database. That is, we can extract its data and store that data in Table 1-1 and Table 1-2 on page 9 shown below. Similarly, if the data was already in these tables, we can publish it as a sales order document.

Table 1-1 Sales order

Number	Date	Customer
123	2330-07-28	456
...	...	...

Table 1-2 Items table

SONumber	Number	PartNumber	Quantity	Price
123	1	"XY-47"	14	16.80
123	2	"B-987"	6	2.34
...	...	...	...	...

Regardless of whether we are shredding or publishing XML documents, there are two important things to notice here.
First, there is no XML "visible" inside the database. That is, the XML document is completely external to the database. It is constructed from data already in the database or is used as a source of new data to store in the database.
Second, the database schema matches the XML schema. That is, a different XML schema is needed for each database schema. Databases that use XML in this fashion are known as *XML-enabled databases*.

On the other hand, if the document itself is stored in the database, it can be stored in a set of tables (see Table 1-3 through Table 1-6 on page 10) designed especially to store XML documents. Note that the document ID, below, is a value generated by the database or assigned by the application. It does not necessarily contain data extracted from the document itself.

Table 1-3 Documents table

ID	Name
34	"SalesOrder123.xml"

Table 1-4 Elements table

DocumentID	ElementID	ParentID	Name	OrderInParent
34	1	NULL	"SalesOrder"	1
34	2	1	"OrderDate"	1
34	3	1	"CustomerNumber"	2
34	4	1	"Item"	3
34	5	4	"PartNumber"	4
34	6	4	"Quantity"	2
34	7	4	"Price"	3
34	8	1	"Item"	4
34	9	8	"PartNumber"	1

DocumentID	ElementID	ParentID	Name	OrderInParent
34	10	8	"Quantity"	2
34	11	8	"Price"	3

Table 1-5 Attributes table

DocumentID	AttributeID	ParentID	Name	Value
34	1	1	"Number"	123
34	2	4	"Number"	1
34	3	8	"Number"	2

Table 1-6 Text table

DocumentID	TextID	ParentID	Value	OrderInParent
34	1	2	"2003-07-28"	1
34	2	3	"456"	1
34	3	5	"XY-47"	1
34	4	6	"14"	1
34	5	7	"16.80"	1
34	6	9	"B-987"	1
34	7	10	"6"	1
34	8	11	"234"	1

In this case, the XML is "visible" inside the database. That is, the database contains information such as element type and attribute names. Furthermore, a single database schema can be used to store all XML documents. That is, the database schema models XML documents, not the data in those documents. Databases that use XML in this fashion are known as *native XML databases*.

A more technically correct view of the difference between XML-enabled databases and native XML databases entails that XML adds a new data model to the world of databases. That is, in addition to the existing hierarchical, relational, object-oriented, multi-valued, and other data models, XML adds its own model. The *XML data model* is an ordered tree with typed, labeled branch nodes and data stored in unlabeled leaf nodes.

In this view, an XML-enabled database is one that uses a non-XML data model and maps instances of this model (such as the tables needed to store sales order information) to instances of the XML data model (such as an XML schema for a sales order document). A native XML database is one that uses the XML data model directly.

1.4 XML-enabled database: using XML to exchange data

As we have seen, an XML-enabled database is one that has a data model other than XML. Most commonly, this is the relational model. Individual instances of this data model are mapped to one or more instances of the XML data model. For example, a relational schema for sales order data can be mapped to different XML schemas, such as a schema for a sales order document, or a report showing sales by region.

XML-enabled databases generally include software for transferring data between themselves and XML documents. This software can be integrated into the database engine or external to the engine. For example, DB2 XML Extender, the XML Wrapper, and SQL/XML can all transfer data between XML documents and the DB2 database. XML Extender and XML Wrapper are external to the database engine, while SQL/XML support is integrated into the DB2 database engine itself. For more information about these products, see Chapter 2, “XML services in DB2 and DB2 Information Integrator” on page 23.

> **Note:** The XML Extender is a DB2 extender for working with XML documents. The XML Wrapper is a wrapper in DB2 Information Integrator that treats XML documents as a relational data source. SQL/XML is a standard set of extensions to SQL for publishing data as XML.

As a general rule, the software used by XML-enabled databases cannot handle all possible XML documents. Instead, it can handle the subclass of documents that are needed to model the data found in the database. For example, data transfer software that works with relational databases rarely handles mixed content XML documents, since mixed content is not easily modeled using the relational model.

> **Important:** The primary advantage of using an XML-enabled database is that it keeps existing data and applications intact. That is, adding XML functionality to the database is simply a matter of adding and configuring the software that transfers data between XML documents and the database. There is no need to change existing data or applications.

1.4.1 XML as a data exchange format

As a general rule, XML-enabled databases use XML only as a data exchange format, and the documents used by the database have no permanent identity. For example, suppose XML is used to transfer temperature data from a weather station to a database. After the data from a particular document is stored in the database, the document is discarded.

"Retrieving" the document means querying the database for the desired data and constructing an XML document from the results. It is not possible to ask for the document by name, and there is no guarantee that the original document can be exactly reconstructed. Because of this, it is not a good idea to shred a document into an XML-enabled database as a way of storing the XML document. Instead, you should think in terms of publishing existing relational data (regardless of its source) as XML.

This is in contrast to native XML databases, which do retain document identity. For example, suppose a user's manual is written in XML and stored in a native XML database. In this case, XML is not just a way to exchange the chapters and sections in the user's manual with another application; the XML documents that comprise the user's manual are in the native format of the manual, and are retained as such by the database.

An important consequence of using XML as a data exchange format is that an XML-enabled database will only retain information that its own model considers important. In the case of an XML-enabled relational database, this means the data itself and the hierarchical relationships (parent, child, sibling) among the data. All other information, including entity references, CDATA sections, comments, processing instructions, and the DTD, are ignored. Even the order in which elements appear in their parent is lost, since relational databases have no concept of order among columns or rows.

1.4.2 Mapping a database schema to an XML schema

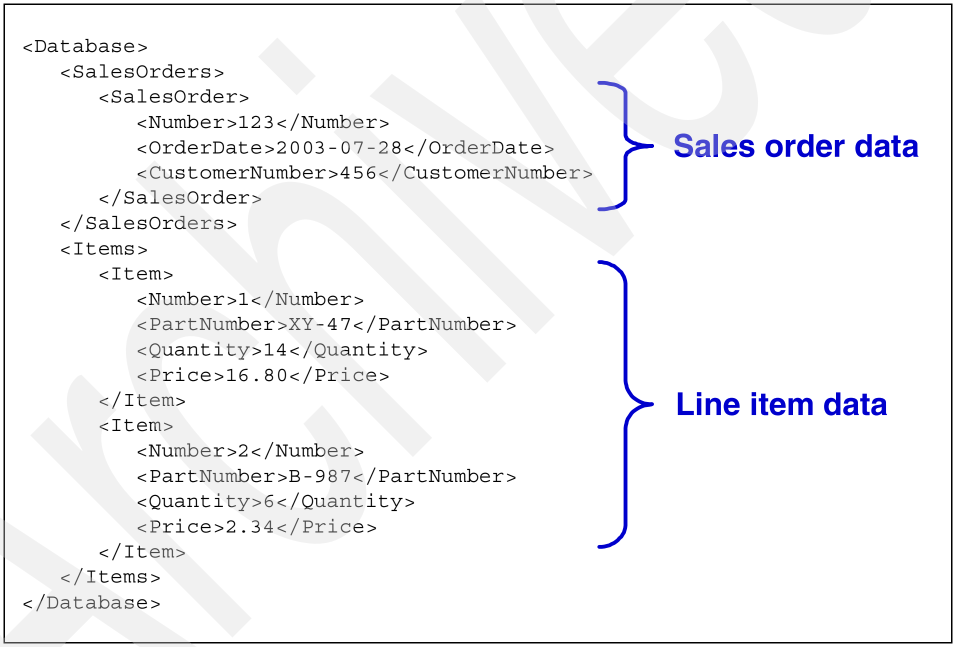
When using an XML-enabled database, it is necessary to map the database schema to the XML schema (or vice versa). Such mappings are many-to-many. For example, a database schema for sales order information can be mapped to an XML schema for sales order documents, or it can be mapped to an XML schema for reports showing the total sales by region.

Mapping database schemas to XML schemas, like writing queries, is generally a design time operation. Although it is possible to generate mappings from a given database schema to an XML schema and vice versa, such mappings are usually just a starting point, since the process that generates the mapping must make a number of assumptions about the schemas, not all of which are correct.

There are three important kinds of mappings, all of which are supported by the DB2 family of products: the *table-based mapping*, the *object-relational mapping*, and *query languages*. The table-based mapping and the object-relational mapping are important because they define bi-directional mappings. That is, the same mapping can be used to transfer data both to and from the database. While XML query languages only define a mapping in one direction (currently, from the database to XML), they are important because they are much more flexible than the other two mappings.

Table-based mapping

When using a table-based mapping, the XML document must have the same structure as a relational database. That is, the data is grouped into "rows," and rows are grouped into "tables." For example, the document in Figure 1-6 uses a table-based mapping to list the data in sales order 123.



```
<Database>
   <SalesOrders>
      <SalesOrder>
         <Number>123</Number>
         <OrderDate>2003-07-28</OrderDate>
         <CustomerNumber>456</CustomerNumber>
      </SalesOrder>
   </SalesOrders>
   <Items>
      <Item>
         <Number>1</Number>
         <PartNumber>XY-47</PartNumber>
         <Quantity>14</Quantity>
         <Price>16.80</Price>
      </Item>
      <Item>
         <Number>2</Number>
         <PartNumber>B-987</PartNumber>
         <Quantity>6</Quantity>
         <Price>2.34</Price>
      </Item>
   </Items>
</Database>
```

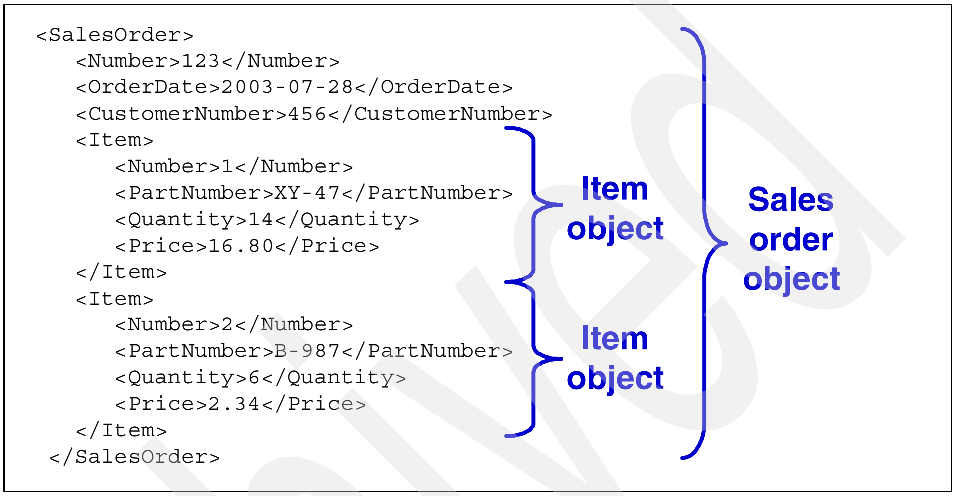
Figure 1-6 Sales order document that uses a table-based mapping

For more information about the table-based mapping, see “Table-based mapping” on page 582.

Object-relational mapping

When using object-relational mapping, an XML document is viewed as a set of serialized objects and is mapped to the database with an object-relational mapping. That is, objects are mapped to tables, properties are mapped to columns, and inter-object relationships are mapped to primary key / foreign key relationships.

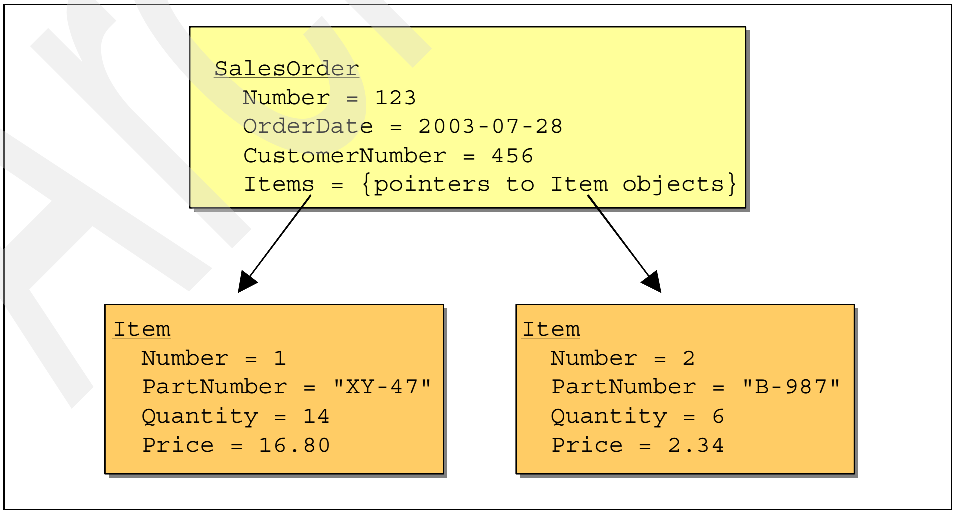
To see how the object-relational mapping works, consider the sales order document shown earlier (Figure 1-7).



```
<SalesOrder>
   <Number>123</Number>
   <OrderDate>2003-07-28</OrderDate>
   <CustomerNumber>456</CustomerNumber>
   <Item>
      <Number>1</Number>
      <PartNumber>XY-47</PartNumber>
      <Quantity>14</Quantity>
      <Price>16.80</Price>
   </Item>
   <Item>
      <Number>2</Number>
      <PartNumber>B-987</PartNumber>
      <Quantity>6</Quantity>
      <Price>2.34</Price>
   </Item>
 </SalesOrder>
```

Figure 1-7   Sales order document that uses an object-relational mapping

This document can be viewed as the serialization of the tree of objects shown in Figure 1-8.



```
SalesOrder
   Number = 123
   OrderDate = 2003-07-28
   CustomerNumber = 456
   Items = {pointers to Item objects}
```

```
Item
   Number = 1
   PartNumber = "XY-47"
   Quantity = 14
   Price = 16.80
```

```
Item
   Number = 2
   PartNumber = "B-987"
   Quantity = 6
   Price = 2.34
```

Figure 1-8   A tree of sales order objects

Using an object-relational mapping, this can be mapped to the sales order (Table 1-7) and line item table (Table 1-8).

Table 1-7 SalesOrder table

Number	Date	Customer
123	2003-07-28	456
...	...	...

Table 1-8 Items table

SONumber	Number	PartNumber	Quantity	Price
123	1	"XY-47"	14	16.80
123	2	"B-987"	6	2.34

Unlike the table-based mapping, where the data from different tables is listed in separate parts of the XML document, the object-relational mapping uses nesting to show the relationship between data from different tables. For example, the data from the line item table is nested inside the data from the sales order table.

For more information about the object-relational mapping, see “Object-relational mapping” on page 585.

Query languages

While the table-based mapping and the object-relational mapping require the XML schema to closely match the database schema, query languages provide more flexibility in constructing XML documents.

SQL/XML

For XML-enabled relational databases, the most important query language is SQL/XML, which is a set of extensions to SQL for creating XML documents and fragments from relational data. The main features of SQL/XML are an XML data type, a set of scalar functions for creating XML (XMLELEMENT, XMLATTRIBUTES, XMLFOREST, and XMLCONCAT), and an aggregate function for creating XML (XMLAGG).

For example, the following call to the XMLELEMENT function:

```
XMLELEMENT(NAME Customer,
      XMLELEMENT(NAME Name, customers.name),
      XMLELEMENT(NAME ID, customers.id))
```

constructs the following Customer element for each row in the customers table:

```
<Customer>
      <Name>customer name</Name>
      <ID>customer id</ID>
</Customer>
```

For more information about SQL/XML, see 2.1, “SQL/XML” on page 24 and Chapter 8, “Publishing data as XML” on page 209.

XQuery

XQuery is an XML query language being defined by the W3C. As of this writing, it is not yet finished. However, more than twenty implementations of XQuery already exist.

XQuery is designed to query XML documents, not relational databases. Thus, to implement it over a relational database, it is necessary to first map the relational database to one or more virtual documents. The easiest way to do this is to map each table to a separate document using the table-based mapping. Thus, an XQuery statement querying a virtual document can be mapped to a SELECT statement querying a table (Figure 1-9).

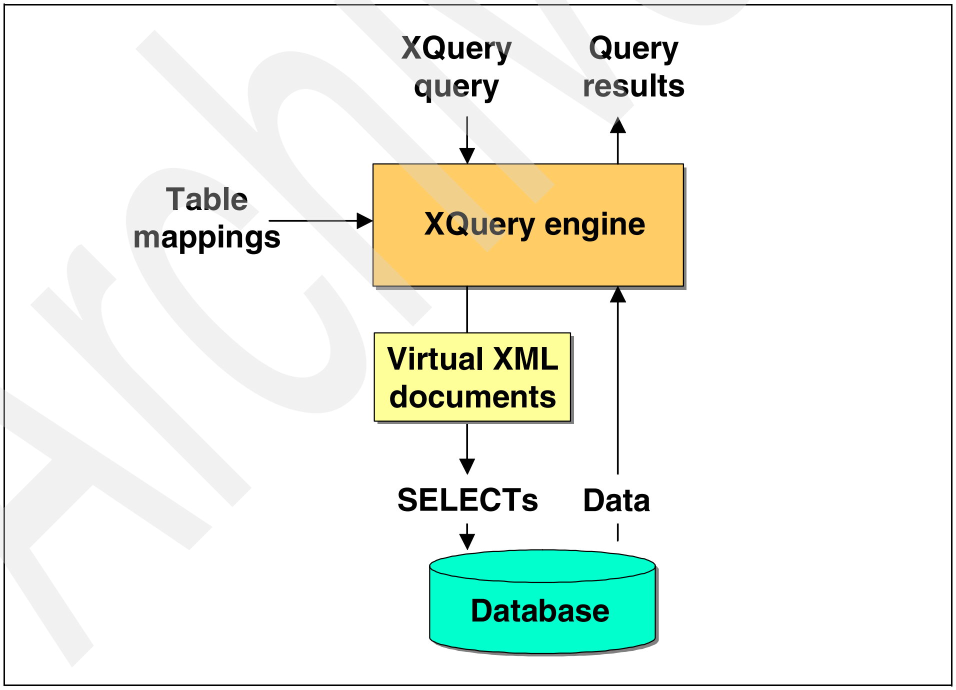


Figure 1-9 Implementing XQuery over a relational database

For example, the following XQuery statement is equivalent to the SQL/XML statement in the previous section:

```
FOR $c IN document("customers")/row/Customer
   RETURN
   <Customer>
         <Name>{$c/name}</Name>
      <ID>{$c/id}</ID>
   </Customer>
```

For a prototype implementation of XQuery over DB2, see the XML for Tables project on the alphaWorks® Web site:

http://alphaworks.ibm.com/tech/xtable

For more information about XQuery, see the W3C's XQuery Web site:

http://www.w3.org/XML/Query

1.5 Native XML DB: managing documents and semi-structured data

A native XML database is one that uses the XML data model directly. That is, it uses a set of structures that can store arbitrary XML documents. This makes native XML databases a good choice for two general situations:

- Those in which the schema of the documents to be stored is not known at design time.
- Those in which the data to be modeled does not fit well in other models, such as the relational model.

XML-enabled databases do not work well in the first case because they require that the database be configured for each XML schema for which data is to be stored. Although this can be done at runtime, this procedure is generally error-prone and results in less-than-optimal mappings. XML-enabled databases might work for the second case, but only for some classes of data. For example, although the XHTML schema can be mapped to a relational schema, the heavy use of mixed content in XHTML requires almost one table per element type. This results in poor performance when retrieving data.

In addition to providing XML-enabled storage in DB2, XML Extender provides simple native XML storage in DB2. It does this by storing complete XML documents in “text” columns, which can then be queried in a variety of ways. Net Search Extender also provides simple native XML storage in DB2 by providing XML-aware full-text searches and structural searches. For more information on both products, see Chapter 2, “XML services in DB2 and DB2 Information Integrator” on page 23.

> **Note:** Net Search Extender is a DB2 extender that provides full-text search capabilities for text documents stored in columns.

1.5.1 Use cases

The best way to understand native XML databases is to consider real-world use cases. For example:

- Document management, that is, managing things like end-user documentation, marketing brochures, Web pages, and so on. Native XML databases are particularly well suited for this case, because the XML data model fits this kind of document well.
- Semi-structured data. Semi-structured data fits very well into the XML data model for two reasons. First, the XML data model accommodates a sparsely populated schema without wasting any space. Second, the XML data model is arbitrarily extensible, so a fixed schema is not required.
- Long-running transactions. Many e-commerce transactions now use XML documents as a way to exchange state information between different parts of the application. Since these transactions can often last several weeks (for example, each part might require human approval), a native XML database is a good way to store the state of the transaction while it is being performed, even if the final state of the transaction is stored in a relational database. A native XML database allows the current state of the transaction to be queried with an XML query language, and may provide other XML tools, such as XSLT transformations and versioning.
- Archiving documents. Many companies, such as those in the pharmaceutical and financial industries, must archive documents for legal reasons. If these are XML documents, then a native XML database is a natural choice for archiving, since its XML query language support allows documents to be used as a source of historical data for trend analysis and, etc.

1.5.2 Technical definition of a native XML database

Before discussing the different IBM products, software, and software components that support XML in a database environment, we discuss the technical definition of a native XML database on a slightly more theoretical level.

Although there is no absolute definition, the following definition from the XML:DB Initiative (see `http://www.xmldb.org`)—a coalition of native XML database vendors—is widely quoted. A native XML database is one that:

- Defines a data model for XML. The minimal model includes elements, attributes, text, and document order.
- Uses an XML document as its fundamental (logical) unit of storage.
- Can use any physical storage strategy.

Let's now look at each part of this definition in more detail.

XML data model

An XML data model describes what parts of an XML document are logically significant. For example, all XML data models include elements, attributes, text, and document order. But some models also include things like entity references and CDATA sections, while others do not. Whether this makes a difference depends on the application; these are critical for XML editors and a nuisance for health data.

XML data models are used as the basis for XML query languages and define the minimum amount of information that a native XML database must store. Native XML databases are free to define their own XML data model because, at the time native XML databases were first created, there was no standard XML data model. As a result, there are native XML databases based on the info set, DOM objects, the XPath 1.0 data model, and the XQuery data model, among others. Fortunately, many native XML databases (including DB2 XML Extender) support XPath 1.0 and its data model today, and most of them will probably use the XQuery data model in the future.

For more information about the XPath 1.0 data model, see:

`http://www.w3.org/TR/xpath`

For more information on the XQuery (and XPath 2.0) data model, see:

`http://www.w3.org/TR/xpath-datamodel/`

Fundamental unit of storage

A *fundamental unit of storage* is the smallest grouping of data that logically fits together in storage. In a relational database, this is a row. In a native XML database, this is a document. Since any document fragment headed by a single element is potentially an XML document, the choice of what constitutes an XML document is purely a design decision. Here are some examples:

- A book written in XML. While most people would agree that the book should probably be split across multiple documents, and that it would be silly to have separate documents for each paragraph, what the ideal document size should be is not clear. For example, should each document contain a chapter? A section? A subsection? This depends entirely on the book and how it will be used.

- Medical data stored as XML documents. There might be a separate XML document for each patient, for each medical provider (doctor, hospital, clinic, and so on), and for each insurance company. Thus, the database schema is to a certain extent normalized, although this normalization is not as complete as might be achieved in a relational database.
- XML search engine documents. In this case, no attempt is made to find an ideal document size. Instead, each document is simply stored in the database, regardless of whether its size is ideal.

Implementation

A native XML database can use any implementation. For example, native XML databases have been implemented in all of the following ways:

- Store whole XML documents in CLOBs in a relational database, and index individual element and attribute values. This is what DB2 XML Extender does.
- Store parsed XML documents in a fixed set of tables (elements, attributes, text, and so on) in a relational database.
- Store parsed XML documents as DOM trees in an object-oriented database.
- Store parsed XML documents in an indexed set of hash tables.

How a native XML database is implemented can significantly affect performance. Perhaps the biggest single factor affecting performance is whether the document is stored as a whole, or in parsed form. For example, a document stored in an indexed CLOB is stored whole. A document stored as a DOM tree in an object-oriented database is stored in parsed form, since it is parsed at insert time, and stored as individual elements and attributes.

To see how this affects performance, consider the following classes of queries:

- Queries that retrieve whole documents based on indexed values. Databases that store whole documents provide the best performance on these queries, since there is no need to rebuild or re-serialize the document, as is the case when a document is stored in parsed form.
- Queries that retrieve indexed values. Given equivalent indexing schemes, all databases perform equally well on these queries, since they can be resolved simply by searching the indexes.
- Queries that retrieve unindexed values or document fragments. Databases that store pre-parsed documents perform best on these, since databases that store whole documents must parse each document to resolve these queries. For the latter databases, performance might be acceptable if the query needs to parse only a few documents. It will almost certainly be unacceptable if the query parses a large number of documents.

- Queries that update documents. Databases that store pre-parsed documents perform best on these, since they can update, insert, or delete individual nodes. Databases that store whole documents must not only parse and modify the document; they must also rewrite the entire document to storage after updating it.

1.6 Summary

In this chapter, we have looked at the two main ways XML is used with databases.

In an *XML-enabled database*, existing data can be used to create XML documents, a process known as *publishing*. Similarly, data from an XML document can be stored in the database, a process known as *shredding*. In an XML-enabled database, no XML is "visible" inside the database and the database schema must be mapped to an XML schema. XML-enabled databases are used when XML is used as a data exchange format and requires no changes to existing applications.

In a *native XML database*, a set of generic structures is used to store any and all XML documents. Because of this, XML is "visible" inside the database. No schema mapping is necessary since native XML databases use the XML data model directly. Native XML databases support an XML query language and can be built from scratch or on top of an existing database. They are best used for storing documents (such as user's manuals, static Web pages, and marketing brochures) and semi-structured data.

2

XML services in DB2 and DB2 Information Integrator

This chapter describes the XML services available in DB2 and DB2 Information Integrator:

- SQL/XML (DB2)
- XML Extender (DB2)
- Net Search Extender (DB2)
- XML Wrapper (DB2 Information Integrator)
- WebSphere MQ
- WebSphere Studio

In addition, this chapter discusses what products you should use to implement these services.

2.1 SQL/XML

For XML-enabled relational databases, the most important query language is SQL/XML, which is a set of extensions to SQL for creating XML documents and fragments from relational data. It is part of the ISO SQL specification (Information technology - Database languages - SQL - Part 14: XML-Related Specifications (SQL/XML) ISO/IEC 9075-14:2003). SQL/XML support can be found in DB2 UDB for Linux®, Unix and Windows®, and in DB2 for z/OS® V8.

SQL/XML adds a number of new features to SQL. The most important of these are a new data type (the XML data type), a set of scalar functions for creating XML (XMLELEMENT, XMLATTRIBUTES, XMLFOREST, and XMLCONCAT), and an aggregation function (XMLAGG) for creating XML. It also defines how to map database identifiers to XML identifiers.

The following sections briefly describe the main features of SQL/XML. For the complete specification, please refer to the official standard document mentioned above.

ftp://sqlstandards.org/SC32/WG3/Progression_Documents/Informal_working_drafts/

2.1.1 XML data type

The XML data type can have three different values: NULL, an XML document with an XML declaration, or element content. Element content is anything that is legal in an XML element, such as text, elements, comments, and processing instructions. All entries (*) in Example 2-1 on page 25 are "legal" values of the XML data type.

Example 2-1 Legal values for the XML data type

```
*  NULL                                                   1

*  <?xml version="1.0"?>                                  2
   <Book>
      <Title>XML Made Easy</Title>
      <Author>Jim Hu</Author>
   </Book>

*  XML Made Easy                                          3

*  <Title>XML Made Easy</Title>                           4
   <!-- A comment -->
   <Author>Jim Hu</Author>

*  This is a <b>good</b> book.                            5

*  <p>This is a <b>good</b> book.</p>                     6
```

> **Note:** The XML data type is *not* a text data type and XML values are not necessarily strings.
> For example, consider the string "<Database>DB2</Database>". While it is tempting to think of this as a database element whose value is "DB2", this is not the case; it is simply a string. Because this string uses the syntax specified in the XML 1.0 recommendation, we can parse it and create a database element whose value is "DB2", but that is different from the string itself.

Just as the SQL specification does not state how the DOUBLE data type should be implemented, the SQL/XML specification does not state how the XML data type should be implemented. For example, the XML data type could be implemented using DOM nodes, strings, or proprietary structures. (DB2 currently externalizes the XML data type using CLOBs. However, this is an implementation detail and you should not think of the XML data type as a CLOB, especially since attempts to treat XML values as CLOBs and vice versa will result in syntax errors.)

> **Note:** When an XML value is serialized as a string, the result is not necessarily well-formed and therefore might not be parseable by an XML parser. For example, in Example 2-1, only XML data values of examples 2 and 6 are well-formed.

The main reason that the XML data type supports non-well-formed values is so it can use these as intermediate values while executing queries. That said, there are good reasons to return non-well-formed values as well. For example, an application might use SQL/XML to create a forest of `<tr>` (table row) elements, then insert these into a `<table>` element inside an XHTML document it is constructing. While the values returned by SQL/XML are not well-formed, the document constructed by the application is well-formed.

2.1.2 SQL/XML functions

The functions defined by SQL/XML can be used to construct XML values from SQL expressions, such as column names or constants, or from other XML values. For example, the SELECT statement in Example 2-2 uses XMLELEMENT and XMLATTRIBUTE to construct Customer elements.

Example 2-2 Using XMLELEMENT and XMLATTRIBUTES

```
SELECT XML2CLOB(
         XMLELEMENT(NAME Customer, XMLATTRIBUTES(customers.id AS ID),
                  XMLELEMENT(NAME Name, customers.name),
                  XMLELEMENT(NAME Street, customers.street),
                  XMLELEMENT(NAME City, customers.city),
                  XMLELEMENT(NAME State, customers.state),
                  XMLELEMENT(NAME PostCode, customers.postcode),
                  XMLELEMENT(NAME Country, customers.country)
                  )
               )
   AS CustomerXML
FROM customers
```

For each row in the customers table, this statement returns the following value (see Example 2-3) in the CustomerXML column:

Example 2-3 CustomerXML column

```
<Customer ID="customer id">
      <Name>customer name</Name>
      <Street>street address</Street>
      <City>city name</City>
      <State>state name</State>
      <PostCode>postal code</PostCode>
      <Country>country</Country>
</Customer>
```

The other SQL/XML functions are:

- XMLFOREST. Creates a forest of XML elements from a list of SQL expressions, such as column names.
- XMLCONCAT. Concatenates a list of XML values into a forest of XML values.
- XMLAGG. Also concatenates a list of XML values into a forest. The difference between XMLCONCAT and XMLAGG is that the XML values used by XMLCONCAT must all be created from a single row, while the XML values used by XMLAGG are aggregated across rows.

2.1.3 XML2CLOB function

In the previous example, you may have noticed the use of the XML2CLOB function. This function casts an XML value (XML data type) —in this case, the value returned by the XMLELEMENT function— to a CLOB. This function is defined by DB2 and is needed because DB2 does not have a way to return XML values directly to the application.

More details on the SQL/XML functions, as well as examples, are provided in 8.1, “Publishing data using SQL/XML” on page 210.

Until DB2 UDB for Linux, UNIX, and Windows V8.2, XML2CLOB was the only supported operation to convert (serialize) an XML data type value to a string data type value. Serialization is the inverse operation of parsing; it is the process of converting a parsed XML value into a textual XML value.

A new standard SQL/XML function, XMLSERIALIZE (with the CONTENT option) will be introduced in DB2 UDB for Linux, UNIX and Windows V8.2 (still under development at the time of writing of this publication). It will allow you to convert an XML data type value into a result string data type that is appropriate for the length of the XML output. XMLSERIALIZE converts an XML expression into an SQL string value which, in turn, can be bound out to host character variables. With XMLSERIALIZE, you can specify a result type such as CHAR or VARCHAR, which might be more appropriate and result in better performance than CLOB.

2.2 DB2 XML Extender

The XML Extender is a DB2 Extender that provides both XML-enabled and native XML capabilities for DB2. XML-enabled capabilities are provided through *XML collections*, while native XML capabilities are provided through *XML columns*.

DB2 XML Extender consists of a number of user-defined data types, user-defined functions, and stored procedures. These must be installed in each database (for DB2 for z/OS that is a DB2 subsystem or a data sharing group) on which they are used. This process is known as *enabling* the database for XML use. DB2 XML Extender is shipped with DB2 UDB for Linux, Unix and Windows, V7 and later. In version 7, it is installed separately, while in version 8, it is installed as part of DB2 (although you still have to enable the database for XML use). XML Extender is also a free, separately installable component, of DB2 for z/OS V7 and later.

2.2.1 XML collections

XML collections are the way that DB2 XML Extender provides *XML-enabled capabilities* for DB2. An *XML collection* is a collection of tables that has been mapped from the database to an XML document by means of a DAD (Data Access Definition) document.

Note: There is nothing special about the data in an XML collection, and DB2 does not know which data is in an XML collection and which data is not. In particular, you should not think of an XML collection as a collection that is composed only of XML data, since the data can come from any source, not just from XML documents. Furthermore, because it is possible to map the same data using many different DAD documents, a given piece of data can reside in more than one XML collection.

There are two types of DAD documents used by XML collections: *SQL mapping* documents and *RDB node mapping* (relational database node mapping) documents. Both documents use a mapping roughly the same as an object-relational mapping, but have different techniques for specifying that mapping. The main differences between the two types of documents are:

- SQL statement mapping DAD documents contain a SELECT statement and information about how the columns returned by that statement are mapped to an XML document. They can only be used to publish (compose) relational data as XML.
- RDB node mapping DAD documents use a mapping language that specifies how nodes (elements and attributes) in the XML document are mapped to tables and columns in a DB2 database. This type of DAD document allows the XML Extender to construct both INSERT and SELECT statements, so RDB node documents can be used both to publish data to XML and to shred (decompose) XML into the database. That is, the mapping is *bi-directional*.

To publish data as XML using XML Extender, an application uses the dxxGenXML, dxxGenXMLClob, dxxRetrieveXML, or dxxRetrieveXMLClob stored

procedures.
The main difference between these procedures is whether or not you need to pass a DAD document to the procedure.

- dxxGenXML and dxxGenXMLClob accept a DAD document as an input argument.
- dxxRetrieveXML and dxxRetrieveXMLClob accept an XML collection name instead.
  The XML collection name is used to retrieve the DAD document from a special table (XML_USAGE) that is managed by XML Extender. The DAD is stored in this table when the collection is *enabled* (enabling a collection does nothing more than storing the DAD document in the XML_USAGE table, and create the tables if they do not yet exist).

To shred XML documents, an application calls the XML Extender through the dxxShredXML or dxxInsertXML stored procedures. As before, the main difference between both stored procedures is whether they use a DAD or XML collection name as an input parameter.

- dxxShredXML uses a DAD as an input parameter
- dxxInsertXML uses an XML collection as an input parameter

Let us now look at both types of mapping documents in more detail.

SQL statement mapping DAD documents

SQL mapping documents can only be used to publish data as XML. They contain a SELECT statement that specifies what data is to be published. This statement must meet the following conditions:

- Each column in the result set must have a unique name. If two columns have the same name, you can use the AS clause to assign unique names to one or both columns.
- The columns in the select list must be grouped by table.
- The first column in the select list from each table must uniquely identify a row of the table. This column may come from the table—for example, it can be a single-column primary key—or be a generated column.
- The order in which tables appear in the select list must match the nesting of the XML document.
- The result set must be ordered by the unique columns for each table.

These rules guarantee that the data is returned in the result set in the same order as it appears in the XML document. By checking when key column values change, DB2 XML Extender can determine when to create the elements that wrap the data from each table. For example, the following statement

(Example 2-4) can be used to select data for sales orders 123 and 124 from the sales order and line item tables.

Example 2-4 SQL statement for a DAD mapping file

```
SELECT Orders.Number AS SONumber,
       Orders.Customer AS CustNumber,
       Items.Number AS ItemNumber,
       Items.Part AS PartNumber
  FROM Orders, Items
  WHERE (SONumber = Items.SONumber) AND
        ((SONumber = 123) OR (SONumber = 124))
  ORDER BY SONumber, ItemNumber
```

The SQL statement mapping DAD document not only contains the SQL statement that provides the data content for the XML document, but also maps the result set of the query to the XML document structure. It contains a template in which element_node, attribute_node, and text_node elements outline the structure of the XML document. element_node elements may contain attribute_node, text_node, and other element_node elements. attribute_node and text_node elements specify the result set column from which an individual data value is to be retrieved. For example, the following document fragment maps the result set from the SELECT statement in Example 2-4:

Example 2-5 SQL statement DAD mapping

```
<element_node name="SalesOrder">
   <attribute_node name="Number">
         <column name="SONumber" />
   </attribute>
   <element_node name="CustomerNumber">
         <text_node>
            <column name="CustNumber" />
         </text_node>
   </element_node>
   <element_node name="Item" multi_occurrence="YES">
      <attribute_node name="Number">
            <column name="ItemNumber" />
      </attribute>
      <element_node name="PartNumber">
         <text_node>
               <column name="PartNumber" />
         </text_node>
      </element_node>
   </element_node>
```

```
</element_node>
```

When DB2 XML Extender is called with a SQL mapping document, it executes the SELECT statement and constructs one XML document for each unique value of the key column in the outermost table. For example, DB2 XML Extender constructs the following two XML documents (Example 2-6) from the preceding SELECT statement and mapping information:

Example 2-6 Resulting XML document

```
<?xml version="1.0"?>
<SalesOrder Number="123">
   <CustomerNumber>456</CustomerNumber>
   <Item Number="1">
      PartNumber>XY-47</PartNumber>
   </Item>
   <Item Number="2">
      <PartNumber>B-987</PartNumber>
   </Item>
</SalesOrder>


<?xml version="1.0"?>
<SalesOrder Number="124">
   <CustomerNumber>456</CustomerNumber>
   <Item Number="1">
      <PartNumber>XY-47</PartNumber>
   </Item>
   <Item Number="2">
       <PartNumber>B-987</PartNumber>
   </Item>
</SalesOrder>
```

One additional feature of SQL mapping documents is that an application can pass a different SELECT statement at runtime than that which appears in the DAD document. Because this statement uses the mapping information in the DAD document, it must return a result set with the same structure as that returned by the SELECT statement in the DAD document. However, the result set can contain different data. In other words, both SELECT statements should have the same select list, FROM clause, join conditions, and ORDER BY clause, but can have different criteria in their WHERE clauses.

RDB node mapping documents

RDB node mapping documents can be used both to publish relational data as XML, and to shred XML documents into the database. Instead of a SELECT statement, they contain information that maps elements and attributes to tables and columns. This allows DB2 XML Extender to create both SELECT statements (during composition) and INSERT statements (during decomposition or shredding) from the same information.

The structure of the mapping information is essentially the same as that in SQL mapping DAD documents; that is, it is a template that outlines the structure of the XML document. The main difference is that attribute_node and text_node elements contain RDB_node elements instead of column elements.

These RDB_node elements specify the *table and column* to which an attribute or text node is mapped, as well as any search conditions that apply to that column. In addition, a special RDB_node element in the first element_node element lists the names of the tables used by the mapping and the conditions used to join them.

For example, the following RDB node mapping document (Example 2-7) is equivalent to the SQL mapping in the previous section.

Example 2-7 RDB_node mapping

```
<element_node name="SalesOrder">
   <RDB_node>
      <table name="Orders" key="Number" />
      <table name="Items" key="SONumber Number" />
      <condition>Orders.Number=Items.SONumber</condition>
   </RDB_node>
   <attribute_node name="Number">
      <RDB_node>
         <table name="Orders" />
         <column name="Number" type="integer"/>
         <condition>(Orders.Number=123) OR (Orders.Number=124)</condition>
      </RDB_node>
   </attribute>
   <element_node name="CustomerNumber">
      <text_node>
         <RDB_node>
            <table name="Orders" />
            <column name="Customer" type="integer"/>
         </RDB_node>
      </text_node>
   </element_node>
   <element_node name="Item" multi_occurrence="YES">
      <attribute_node name="Number">
```

```
         <RDB_node>
            <table name="Items" type="integer"/>
            <column name="Number" />
         </RDB_node>
      </attribute>
      <element_node name="PartNumber">
         <text_node>
            <RDB_node>
               <table name="Items" />
               <column name="Part" type="varchar(10)"/>
            </RDB_node>
         </text_node>
      </element_node>
   </element_node>
</element_node>
```

An additional feature of RDB node mapping documents is that the application can pass different conditions at runtime than those that appear in the DAD document. An location path expression specifies the element or attribute to which the condition applies.

2.2.2 XML columns

XML columns are the way that DB2 XML Extender provides simple *native XML storage* in DB2. An *XML column* is a column in a DB2 table that is used to store a complete XML document. It must have a data type of XMLVARCHAR, XMLCLOB (XMLDBCLOB if your database is enabled for DBCS), or XMLFILE and all of the XML documents stored in a particular XML column should use the same XML schema. For example, Table 2-1 on page 34 shows how sales order documents can be stored in the OrderDocuments table.

Note: XMLVARCHAR, XMLCLOB, XMLDBCLOB, and XMLFILE are user-defined types defined by DB2 XML Extender. They are based on the VARCHAR, CLOB, DBCLOB and VARCHAR, DB2 built-in data types, respectively. In the case of XMLFILE, the name of the XML file, rather than its contents, are stored in the XML column.

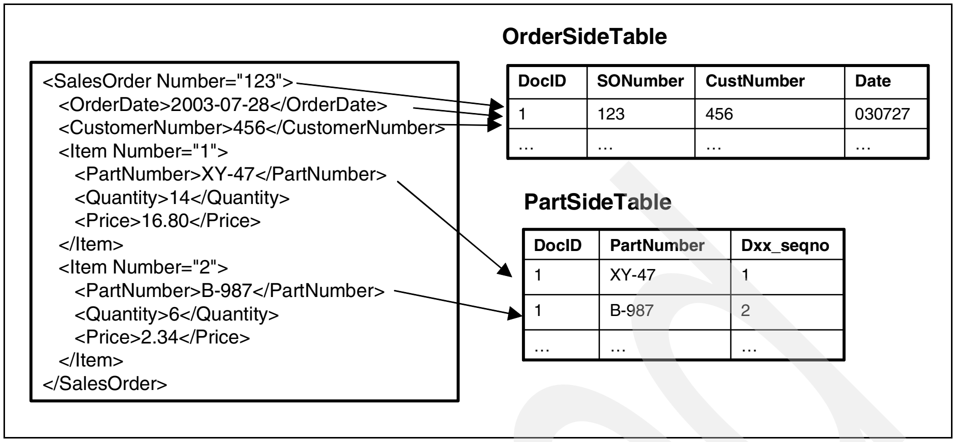
Table 2-1 OrderDocuments table

DocID	Document
1	<SalesOrder Number="123"> <OrderDate>2003-07-28</OrderDate> <CustomerNumber>456</CustomerNumber> <Item Number="1"> <PartNumber>XY-47</PartNumber> <Quantity>14</Quantity> <Price>16.80</Price> </Item> <Item Number="2"> <PartNumber>B-987</PartNumber> <Quantity>6</Quantity> <Price>2.34</Price> </Item> </SalesOrder>
2	......

The documents stored in an XML column can be indirectly indexed and queried by using *side tables*. Side tables are tables that are separate from the table that contains the XML column. They contain values extracted from the XML documents in the XML column, and a foreign key column (although the relationship is not explicitly defined by XML Extender) that points back to the table that contains the XML column. DB2 XML Extender automatically maintains the data in side tables.

A subset of XPath 1.0 is used to specify which values of the XML document should be stored in side tables.

For example, Figure 2-1 on page 35 shows how side tables may be built from a sales order XML document. The sales order number, date, and customer number are stored in one side table. The part numbers used in the sales order are stored in a different side table. In both side tables, the DocID column is a foreign key pointing back to the table containing the XML column.



```
<SalesOrder Number="123">
  <OrderDate>2003-07-28</OrderDate>
  <CustomerNumber>456</CustomerNumber>
  <Item Number="1">
    <PartNumber>XY-47</PartNumber>
    <Quantity>14</Quantity>
    <Price>16.80</Price>
  </Item>
  <Item Number="2">
    <PartNumber>B-987</PartNumber>
    <Quantity>6</Quantity>
    <Price>2.34</Price>
  </Item>
</SalesOrder>
```

OrderSideTable

DocID	SONumber	CustNumber	Date
1	123	456	030727
…	…	…	…

PartSideTable

DocID	PartNumber	Dxx_seqno
1	XY-47	1
1	B-987	2
…	…	…

Figure 2-1 Side tables for a sales order document

Applications can effectively query individual values in the XML document by querying the side tables. These queries are equivalent to queries that a database engine resolves by reading only indexes. They can also retrieve documents that match a particular "index" value (entry in the side table) or values, by querying the side tables and then retrieving the corresponding documents from the document table. The performance of these queries can be optimized by indexing the values in the side tables.

For example, the query in Example 2-8 can be used to retrieve the numbers of sales orders that contain part XY-47.

Example 2-8 Query retrieving info using side tables

```
SELECT OrderSideTable.SONumber
  FROM OrderSideTable, PartSideTable
  WHERE OrderSideTable.DocID = PartSideTable.DocID AND
        PartSideTable.PartNumber = 'XY-47'
```

And the following query (Example 2-9) can be used to retrieve the sales order documents that contain part XY-47:

Example 2-9 Retrieving XML documents from an XML column using side table info

```
SELECT OrderDocuments.Document
  FROM OrderDocuments
  WHERE OrderDocuments.DocID IN
             (SELECT PartSideTable.DocID
   FROM PartSideTable
   WHERE PartSideTable.PartNumber = 'XY-47'
)
```

Note: The XML Extender can create a default view that joins the table in which the document is stored with all of its side tables. Applications can query this view directly, which saves them from having to join the tables themselves. A sample default view can be found in Example 5-10 on page 128.

To specify which values in the XML document are to be extracted and stored in side tables, we also use a DAD document. This type of DAD uses a different format from the DAD documents used by XML collections. For example, the fragment of a DAD document in Example 2-10 shows how to specify the layout of the side tables in the preceding examples.

Example 2-10 XColumn DAD file

```
<Xcolumn>
   <table name="OrderSideTable">
      <column name="SONumber" type="integer"
                 path="/SalesOrder/@Number" multi_occurrence="NO" />
      <column name="CustNumber" type="integer"
                 path="/SalesOrder/CustomerNumber" multi_occurrence="NO" />
      <column name="Date" type="date"
                 path="/SalesOrder/OrderDate" multi_occurrence="NO" />
   </table>
   <table name="PartSideTable">
      <column name="PartNumber" type="varchar(10)"
              path="/SalesOrder/Item/PartNumber" multi_occurrence="YES" />
   </table>
</Xcolumn>
```

After you have created a DAD document for a given XML column, you must *enable* that column. This is done with the dxxadm command, the parameters of which include the table name, column name, and DAD file name.

Specific elements and attributes of XML documents stored in XML columns can be updated with the Update user-defined function. This function is defined by DB2 XML Extender and accepts the name of an XML column, a location path expression, and a new value as its parameters. For example, the SQL UPDATE statement in Example 2-11 on page 37 changes the customer number in sales order 123 (which has a DocID of 1) to 457 using the XML Extender Update UDF.

Example 2-11 Using the Update() UDF

```
UPDATE OrderDocuments
      SET Document=Update(Document, '/SalesOrder/CustomerNumber', '457')
   WHERE DocID = 1
```

It should be noted that a partial update of an XML document stored in an XML column is potentially an expensive operation. This is because the document must be parsed, the value changed, and the new document serialized and stored back into the XML column. (When you update the entire document by replacing it, parsing and reserialization is not required.) Thus, updates of XML columns should be done sparingly or only to small XML documents.

Location path expressions can also be used to extract individual values from an XML document using user-defined functions defined by DB2 XML Extender. There are two UDFs for each data type that can be extracted (varchar, integer, date, and so on).

- The first UDF for each type returns a scalar value, for example extractCLOB(), and is used with location path expressions that return single values.
- The second UDF returns a table with a single column, for example extractCLOB**s**(), and is used with location path expressions that return multiple values.

For example, the statement in Example 2-12 extracts the order date from sales order 123.

Example 2-12 Using the extractInteger() function

```
SELECT extractInteger(Document, '/SalesOrder/OrderDate') AS OrderDate
   FROM OrderDocuments
WHERE DocID = 1
```

Like updates, the location path-based extraction functions are expensive because the document must be parsed. Therefore, frequently-queried values should be extracted at insert time and stored in side tables.

2.3 Net Search Extender

DB2 Net Search Extender provides simple native XML functionality for DB2 in the form of XML-aware full-text searches and structural searches. An

XML-aware full-text search is one that only searches the text (element and attribute values) in an XML document, but not the markup (element and attribute names, comments, and so on). An *XML-aware structural search* is one that searches only the values of a particular element type or attribute.

You can use Net Search Extender with columns that have a data type of CHAR, VARCHAR, LONG VARCHAR, CLOB, DBCLOB, BLOB, GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC, or DATALINK, as well as with columns that have types that can be converted to these types, such as the XMLVARCHAR and XMLCLOB data types defined by the XML Extender.

Just like DB2 XML Extender, DB2 Net Search Extender contains a number of functions, stored procedures, as well as views of catalog information. These must be installed in each database on which they are used, a process known as *enabling* the database for use by Net Search Extender.

DB2 Net Search Extender requires version 8.1 or later of DB2 for Linux, Unix and Windows.

2.3.1 Creating indexes

To create a full-text index on a column, you use the CREATE INDEX statement. This allows you to specify the name of the table and column whose documents are to be indexed, the code page used by the documents, the document format, how often the index is to be updated, and so on.

> **Note:** This is not the same as the SQL CREATE INDEX statement used in DB2 to create an index on a table. It has a different syntax and is executed by Net Search Extender, not DB2.

When creating an index on a column used to store XML documents, you will need to specify the following information:

- Format. This is always XML.
- Code page. The Net Search Extender does not follow the rules of the XML 1.0 recommendation for determining the code page used by an XML document. In particular, it ignores the encoding declarations in XML declarations and text declarations. Instead, it requires that all XML documents stored in a column use the same code page. You can specify this code page in the CREATE INDEX statement. If you do not specify a code page, the default code page for the database is used.
- Conversion function. If the data type of the column in which the XML documents are stored is not one of the directly supported data types (CHAR, VARCHAR, and so on), you must specify the name of a function that can

convert the data in the column to one of the supported types. In the case of the UDTs defined by XML Extender, you can use the casting functions provided by the XML Extender, such as DB2XML.VARCHAR().

- Document model. This describes the set of elements and attributes whose values are to be indexed. If no document model is provided, all elements and attributes are indexed.

2.3.2 Full-text searches

DB2 Net Search Extender has three scalar functions (CONTAINS, NUMBEROFMATCHES, and SCORE), a table-valued function (TEXTSEARCH), and a stored procedure (also named TEXTSEARCH) for performing full-text searches. For each of these, a search argument specifies what terms to search for and how to search for them. For example, you can choose to search for a single word, any or all of a list of words, words in the same sentence or paragraph, synonyms of a word, words that match a wild card, and so on.

For example, suppose we store XML sales order documents in the Document column, a LONG VARCHAR column in the OrderDocuments table. We create an index on this column with the following statement (Example 2-13).

Example 2-13 Creating an index

```
CREATE INDEX OrderIndex
      FOR TEXT ON OrderDocuments (Document) FORMAT XML
```

To search for all documents that contain the word "wrench", we use the CONTAINS function, passing in the name of the column and a simple search argument (Example 2-14). The single quotes (') in the search argument delimit the search argument string within the SELECT statement. The double quotes (") delimit search values —as opposed to key words— within the search argument itself. The CONTAINS() function returns 1 if the document satisfies the search argument, and 0 if it does not.

Example 2-14 Using the CONTAINS() function

```
SELECT Document
   FROM OrderDocuments
WHERE CONTAINS(Document, '"wrench"') = 1
```

A more complex query might search for documents that contain synonyms for the word "wrench" or contain the phrase "Automotive Suppliers" (Example 2-15 on page 40). In this case, MechanicThesaurus is the name of a thesaurus we have created that contains words used by mechanics.

Note: DB2 Net Search Extender allows you to create custom thesauruses. To do this, you create a text file containing your synonyms, then pass this file to Net Search Extender, which compiles it into a thesaurus file.

Example 2-15 Using a thesaurus

```
SELECT Document
   FROM OrderDocuments
WHERE CONTAINS(Document,
                  'THESAURUS "MechanicThesaurus"
                EXPAND SYNONYM TERM OF "wrench" |
                   "Automotive Suppliers"') = 1
```

2.3.3 Structural queries

A structural query is one that queries a particular *section* of the document, where a section is defined by an XPath expression. (Only a subset of XPath is supported. In particular, only the child (/), attribute (@), and descendant-or-self (//) axes are supported. Predicates are not supported.) You can use the default document model or define your own sections in a *document model file*.

The default document model indexes all elements and attributes in the document. Each element or attribute is identified by its XPath. For example, to search for all documents in which the value of the PartNumber element contains XY-47, we again use the CONTAINS function. We use the name (XPath) of the section (element) to specify which part of the document to search, as shown in Example 2-16.

Example 2-16 Structural search query

```
SELECT Document
   FROM OrderDocuments
WHERE DB2TX.CONTAINS(Document,
                   'SECTION ("/SalesOrder/Item/PartNumber") "XY-47"') = 1
```

Tip: The CONTAINS function searches for strings that contain a particular token, not match it exactly. Thus, the preceding query also returns documents where the part number is "XY-47 subpart A", "This is part XY-47", and so on.

To define your own document model, you first construct a document model file. This file contains information about the parts of the document you want to index. For example, suppose you only want to index part numbers and customer numbers. You would specify the XPath expressions for these elements in the document model file, using a special XML syntax (Example 2-17).

Example 2-17   Document model file specification

```
<XMLModel>
    <XMLFieldDefinition name="PartNumber"
                       locator="/SalesOrder/Item/PartNumber" />
   <XMLFieldDefinition name="CustomerNumber"
                       locator="/SalesOrder/CustomerNumber" />
</XMLModel>
```

Next, you specify the logical name of your document model, as well as the path of the file containing it, in your CREATE INDEX statement, as shown in Example 2-18.

Example 2-18   Create structural search index

```
CREATE INDEX OrderIndex FOR TEXT ON Orders (OrderDoc)
                FORMAT XML
             DOCUMENTMODEL MyDocumentModel IN C:\MyModels\MyModel.xml
```

With this index, you can only search for values in the PartNumber and CustomerNumber elements. For example, to search for all documents in which the value of the PartNumber element contains XY-47 and the value of the Customer element contains 456, we can use the following statement (Example 2-19).

Example 2-19   Selecting documents using a structural search index

```
SELECT OrderDoc
   FROM Orders
WHERE DB2TX.CONTAINS(OrderDoc,
                     'SECTION ("PartNumber") "XY-47" &
                       SECTION ("CustomerNumber") "456"'
                     ) = 1
```

Note that we use the section names we defined in our model document rather than the XPath expressions we used in the previous statement. This is because the XPath expressions in the previous statement were not really XPath expressions —they were just default section names that happened to look like XPath expressions.

2.4 XML wrapper

The XML wrapper provides XML-enabled capabilities for DB2 Information Integrator by treating an XML document as a source of relational data. In XML terms, it shreds a portion of the XML document according to an object-relational mapping and returns the data as a table. Note that XML Wrapper queries are potentially expensive because the XML Wrapper must parse each document it queries. Thus, if a query queries a large number of documents, or if you frequently query the same XML document, you may want to shred these documents into tables in your database, assuming this is possible.

Wrapper: A *wrapper* is a component in a federated database management system. That is a system that can query data from multiple sources in a single SQL statement. Each wrapper knows how to communicate with a particular type of data source and, if necessary, converts the data to the model used by the federated system. (Most federated systems today are built on top of relational databases and therefore use the relational model.) For example, a wrapper for a particular e-mail system might treat each e-mail message as a separate row in a table, with separate columns for to, cc, from, subject, and the body of the message.

The XML Wrapper is shipped with version 8.1 or later of DB2 Information Integrator.

2.4.1 Registering a wrapper and server

Before using the XML Wrapper, you must register it with the federated server. Registering the XML wrapper tells DB2 where the wrapper library (DLLs) can be found. You must then register a server for the XML Wrapper. For many data sources, a *server* represents a specific data source, such as the Informix® database in the accounting department, or the Microsoft® SQL Server database in the shipping department. This is not true for the XML Wrapper, for which you can use a single server for all your XML documents.

2.4.2 Mapping an XML schema

To use the XML Wrapper with a set of documents that use a particular XML schema, you must map the element types and attributes in that schema to one or more tables. Each table is mapped separately using a CREATE NICKNAME statement. (A *nickname* is a set of data in a data source that is treated as a table. It is easiest to think of a nickname as being a remote table and you may want to substitute the word "table" for the word "nickname" while reading the rest of this section.)

A CREATE NICKNAME is similar to a CREATE TABLE statement, in that it allows you to specify nickname and column names, column data types, column nullability, and primary and foreign key columns. Unlike a CREATE TABLE statement, CREATE NICKNAME (on a server that refers to an XML wrapper) allows you to specify an XPath expression that points to an element type that corresponds to a row in the nickname, the XPath expressions that point to element types or attributes that correspond to columns, and the location of the XML document.

For example, suppose we have the following XML document (Example 2-20):

Example 2-20   XML document

```
<SalesOrder Number="123">
   <OrderDate>2003-07-28</OrderDate>
   <CustomerNumber>456</CustomerNumber>
   <Item Number="1">
      <PartNumber>XY-47</PartNumber>
      <Quantity>14</Quantity>
      <Price>16.80</Price>
   </Item>
   <Item Number="2">
      <PartNumber>B-987</PartNumber>
      <Quantity>6</Quantity>
      <Price>2.34</Price>
   </Item>
</SalesOrder>
```

We can map the schema of this document to a set of nicknames with the following CREATE NICKNAME statements (Example 2-21). We use the suffix "NN" on these nicknames to distinguish them from the Orders and Items tables already in the database.

Example 2-21   Creating nicknames for XML documents

```
CREATE NICKNAME OrdersNN
       (ID        VARCHAR(16) NOT NULL OPTIONS(PRIMARY_KEY 'YES'),
        Number    INTEGER     NOT NULL OPTIONS(XPATH './@Number'),
        OrderDate DATE        NOT NULL OPTIONS(XPATH './OrderDate'),
        CustNum   INTEGER     NOT NULL OPTIONS(XPATH './CustomerNumber'))
   FOR SERVER xml_server
   OPTIONS(DIRECTORY_PATH 'c:\OrderDocs\', XPATH '/SalesOrder')

CREATE NICKNAME ItemsNN
      (ParentID  VARCHAR(16)  NOT NULL OPTIONS(FOREIGN_KEY 'OrdersNN'),
       Number    INTEGER      NOT NULL OPTIONS(XPATH './@Number'),
       PartNum   VARCHAR(10)  NOT NULL OPTIONS(XPATH './PartNumber'),
       Quantity  INTEGER      NOT NULL OPTIONS(XPATH './Quantity'),
       Price     DECIMAL(8,2) NOT NULL OPTIONS(XPATH './Price'))
   FOR SERVER xml_server
   OPTIONS(XPATH '/SalesOrder/Item')
```

Like CREATE TABLE statements, these statements specify a name (nickname), followed by a set of column definitions. However, the other parts of the statement are different, and we will look at each of them in turn.

- The FOR SERVER clause specifies the name of the server that was registered for the XML Wrapper. The name used in the CREATE SERVER statement is the same for all CREATE NICKNAME statements for all XML documents.
- The XPATH clause at the end of the statement specifies the *row element type* —that is, the element type that corresponds to a row in the nickname. The number of elements returned by the expression determines the number of rows in the nickname. For example, the expression /SalesOrder in the first statement always returns a single element for sales order documents. Thus, the OrdersNN nickname will always have one row for a particular document. On the other hand, the /SalesOrder/Item expression in the second statement can return multiple elements —one for each Item element— and the ItemsNN nickname can contain multiple rows, depending on the actual XML document.
- The XPATH clause used in the column definitions specifies the location of the element type or attribute from which the column's data is to be retrieved. Notice that this XPath expression is relative to the row element type. For example, the XPath expression ./@Number is used in both CREATE NICKNAME statements. In the first statement, it specifies the Number attribute of the SalesOrder element type; in the second statement, it specifies the Number attribute of the Item element type.
- The PRIMARY_KEY 'YES' clause in the definition of the ID column in the OrdersNN nickname, specifies that this column contains a primary key generated by the XML Wrapper.

- The FOREIGN_KEY 'OrdersNN' clause in the definition of the ParentID column in the ItemsNN nickname specifies that this column contains a foreign key that points to the OrdersNN nickname.

Primary and foreign keys are used to join different nicknames defined over the same set of XML documents. This is necessary when querying data in any nickname other than the root nickname —that is, the nickname whose row element is highest in the XML document. In more deeply nested documents, a nickname can have both a foreign key pointing to a parent nickname and a primary key pointing to one or more child nicknames.

2.4.3 Identifying an XML document

There are several ways to specify which XML document or documents are to be queried through a nickname. In the CREATE NICKNAME statement for the OrdersNN nickname (Example 2-21 on page 43), the *DIRECTORY_PATH* option specifies the directory containing one or more XML documents. If this option is used, the XML Wrapper will query all documents with a .xml extension in the specified directory. The *FILE_PATH* option is similar to DIRECTORY_PATH, except that it specifies a single XML file.

While the DIRECTORY_PATH and FILE_PATH options require you to specify the XML document or documents to be queried at design time, the *document column* options allow you to specify the XML document or documents to be queried at query execution time.

The document column options require you to define an additional column in the nickname. This column does not correspond to any element types or attributes in the XML document. Instead, it is a placeholder whose name is used in the WHERE clause of a SELECT statement to specify the location of the XML document or documents to be queried.

For example, suppose we want to query a sales order document that exists on the Web. Instead of specifying the DIRECTORY_PATH option for the root nickname, we create a document column (which we name Doc) with the *URI* option, as shown in Example 2-22:

Example 2-22 Using a URI on the CREATE NICKNAME

```
CREATE NICKNAME OrdersNN
      (Doc       VARCHAR(255)          OPTIONS(DOCUMENT 'URI'),
       ID        VARCHAR(16)  NOT NULL OPTIONS(PRIMARY_KEY 'YES'),
       Number    INTEGER      NOT NULL OPTIONS(XPATH './@Number'),
       OrderDate DATE         NOT NULL OPTIONS(XPATH './OrderDate'),
       CustNum   INTEGER      NOT NULL OPTIONS(XPATH './CustomerNumber'))
```

```
   FOR SERVER xml_server
      OPTIONS(XPATH '/SalesOrder')
```

When we write a query using this nickname, we also have to specify the URI of the document in the WHERE clause of the SELECT statement. For example:

```
SELECT * FROM OrdersNN WHERE Doc='http://samples.ibm.com/order123.xml'
```

Other document column options are *FILE* (a file path), *DIRECTORY* (a directory path), and *COLUMN* (an SQL expression that resolves to a column, such as a column name).

2.4.4 Querying an XML document through the XML Wrapper

Root nickname queries

The root nickname is queried like any table in the database. For example, to retrieve the numbers of all sales orders for customer 456, we can use the following SELECT statement, assuming we use our original CREATE NICKNAME statements, which specifies that we are to search all documents with a .xml extension in the c:\OrderDocs directory (Example 2-21 on page 43).

```
SELECT Number FROM OrdersNN WHERE CustNum = 456
```

Non-root nickname queries

Querying non-root nicknames is slightly more complex, since we must join all ancestor nicknames. This is necessary for two reasons. First, the location of the XML document or documents to be queried is only specified in the CREATE NICKNAME statement for the root nickname. Second, it allows us to specify additional filtering conditions on columns in the ancestor nicknames. For example, to find the total number of part XY-47 ordered by customer 456, we use the SELECT statement in Example 2-23.

Example 2-23 Querying non-root nicknames

```
SELECT SUM(Quantity)
  FROM OrdersNN, ItemsNN
  WHERE OrdersNN.ID = ItemsNN.ParentID AND
        CustNum = 456 AND
        PartNum = ‘XY-47‘
```

Notice that we use the primary key of the OrdersNN nickname (the ID column) and the foreign key of the ItemsNN nickname (the ParentID column) to join the nicknames. Should we define nicknames for more deeply nested documents, we would need to join each of the nicknames in the path leading to the lowest level nickname.

2.4.5 Shredding an XML document using the XML Wrapper

In addition to querying XML documents, we can use the XML Wrapper to shred XML documents into relational tables. To do this, we use an INSERT INTO ... SELECT statement. For example, to insert rows into the Orders table, we use the following INSERT statement (Example 2-24).

Example 2-24 Insert into Orders

```
INSERT INTO Orders (Number, Date, Customer)
      SELECT Number, OrderDate, CustNum FROM OrdersNN
```

This statement retrieves the values of the Number attribute of the Order element, the OrderDate element, and the CustomerNumber element and inserts them into the Orders table.

Similarly, we can use the following INSERT statement to insert rows into the Items table (Example 2-25).

Example 2-25 Insert into Items

```
INSERT INTO Items (SONumber, Number, PartNumber, Quantity, Price)
      SELECT OrdersNN.Number, ItemsNN.Number, PartNum, Quantity, Price
        FROM OrdersNN, ItemsNN
        WHERE OrdersNN.ID = ItemsNN.ParentID
```

There are two important things to notice here. First, we retrieve the sales order number from the OrdersNN nickname and use this to populate the SONumber column of the Items table. This is because there is no sales order number in the Items nickname. Second, we still use the OrdersNN.ID and ItemsNN.ParentID columns to join the nicknames, even though we join the corresponding tables in the database with the Orders.Number and Items.SONumber columns.

Finally, if you do use the XML Wrapper to shred XML documents into relational tables, you must be careful to populate the tables in the correct order to maintain referential integrity. In our case, we inserted data into the Orders table first and the Items table second, since the Items table has a foreign key column (SONumber) that points to the primary key column (Number) of the Orders table.

Note that because you control the insert order yourself, you can use the XML Wrapper to shred documents that cannot be shredded by the XML Extender. In particular, you can shred documents in which a child element is mapped to a table that contains the primary key in the relationship with the table to which its parent element is mapped.

For example, suppose you map the following XML document (Example 2-26) to the database in the obvious way, with the SalesOrder element type mapped to the Orders table, the Customer element type mapped to the Customers table, and the Item element type mapped to the Items table.

Example 2-26 Sales order XML document

```
<SalesOrder Number="123">
   <OrderDate>2003-07-28</OrderDate>
   <Customer Number="456">
      <Name>ITSO Insurance, Inc.</Name>
      <Street>123 Main St.</Street>
      <City>Chicago</City>
      <State>IL</State>
      <PostCode>60609</PostCode>
      <Country>US</Country>
   </Customer>
   <Item Number="1">
      <PartNumber>XY-47</PartNumber>
      <Quantity>14</Quantity>
      <Price>16.80</Price>
   </Item>
   <Item Number="2">
      <PartNumber>B-987</PartNumber>
      <Quantity>6</Quantity>
      <Price>2.34</Price>
   </Item>
</SalesOrder>
```

DB2 XML Extender cannot shred this document because it inserts data into the database in the order the data occurs in the XML document. That is, it attempts to insert data into the Orders table, then into the Customers table, and finally into the Items table. Unfortunately, this fails when inserting the row into the Orders table unless the customer row already exists in the Customer table. If you use the XML Wrapper to shred this document, you can insert rows in the correct order: Customers, then Orders, then Items.

2.5 WebSphere MQ

DB2 XML Extender can be used with WebSphere MQ (formerly known as MQSeries®) to send XML documents to and retrieve XML documents from message queues. In particular, XML Extender includes a number of user-defined functions and stored procedures for publishing XML documents to a publication service, sending XML documents to message queues, and reading (read but not remove) or receiving (read and remove) XML documents from message queues. It also includes stored procedures for using XML collections with message queues — reading XML documents from message queues and then shredding them, as well as publishing relational data as XML documents and then sending them to message queues. However, the use of these functions is beyond the scope of this publication. More information can be found in *DB2 XML Extender Administration and Programming.*

2.6 WebSphere Studio

The WebSphere Studio family of products is a set of development tools for enterprise e-business Java-based applications. Besides programming tools, it also enables the developers to test and deploy their application from the common environment. The rich set of utilities and wizards helps developers to simplify common tasks so that they can concentrate on more creative parts of their jobs and be more productive.

WebSphere Studio provides these tools for development of XML and DB2 applications:

- Editors for XML files, DTDs, XML Schemas, and XSL files
- Generation of XML from SQL statements
- Run XSL transformations
- Editors for DB2 data descriptors (database, schema, tables, keys)
- Import and export of data descriptors from and to DB2 databases
- DDL generator
- Creation of SQL statements for Web application generation
- Creation of stored procedures and user-defined functions (UDF)
- Creation of document access definitions (DAD) for XML Extender
- Execute DAD files against DB2 XML Extender
- Creation of Web services from SQL statements, stored procedures, and DAD
- Usage of Web services in UDFs

We will explore most of these functions in Part 3 and Part 4 of this document.

2.7 What products should I use?

In this section, we will try to help you decide which products to use when working with your XML documents.

Product overview

Table 2-2 summarizes what products you can use to accomplish various tasks. These are explained further in the sections that follow.

Table 2-2 Product overview

Task	Product	Comments
Publishing data as XML (composition)	SQL/XML	Best choice due to flexibility.
	XML Extender	Most useful if you want bi-directional mappings. (Use RDB node mapping).
	Write your own code	Realistically limited to table-based mapping.
Shredding XML documents (decomposition)	XML Extender	Use the RDB node mapping (and XML collections).
	XML Wrapper	Avoids minor limitations of RDB node mapping. Application must do more work.
	Write your own code	Realistically limited to table-based mapping.
Storing complete XML documents	XML Extender	Automatically maintains side tables that can be indexed and queried when using XML columns.
Querying XML documents	XML Extender	Can query side tables or use a location path expression to query document.
	Net Search Extender	Full-text and structural searches.
	XML Wrapper	Treats document as if they are a relational source. Document can be on the Web.

Task	Product	Comments
Updating XML documents	XML Extender	Location path expression-based updates.
	Write your own code	More flexible, more work.

2.7.1 XML-enabled storage or native XML storage?

Before you decide what products to use, you need to decide what kind of storage you want to use: XML-enabled storage or native XML storage. The best way to answer this question is to look at your XML documents. Here are some possibilities:

- Your XML documents will contain data that is currently stored in your database. For example, you have historical stock data that you want to publish as XML. By definition, you are using XML-enabled storage and need a way to publish your current data as XML. You may also want to shred incoming XML documents into your existing tables.
- Your XML documents represent a new source of data that is similar to data currently stored in your database. For example, a new business partner is sending you sales orders as XML documents. You will probably want to shred these documents into your existing tables; this is XML-enabled storage. You may also want to store whole (intact) incoming documents in a single column, such as for legal reasons; this is native XML storage.
- Your XML documents represent an entirely new form of data. For example, you are moving from paper invoices to computerized invoices and want to use XML to send and receive invoices. Or you are migrating your documentation from a word processor to DocBook (an XML dialect for writing books). What to do in this case usually depends on what kind of documents you have and how you intend to use them.
  - If your documents are data-centric, then you will usually want to store the data in them in an XML-enabled database. That is, you will shred incoming documents into relational tables and construct outgoing documents from the data in those tables. By storing your data in relational tables, you will be able to work with it using non-XML applications.
  - The main exception to this is when your data is semi-structured. Semi-structured data is sparsely populated; that is, there are many fields, most of which are not used at any one time. Such data is not easily stored in a relational database. The main choices in that case are, either to use many tables (requiring many joins to retrieve data), or a single table with many columns (most of which are null for a given row). Because of this, documents containing semi-structured data are often best stored in a single column; that is, using native XML storage.

- On the other hand, if your documents are document-centric, then you will usually want to store them in a single column; that is, using native XML storage. There are two reasons for this.
  - First, you will preserve document order, entity usage, comments, and so on, something that XML-enabled storage cannot do.
  - Second, while it is possible to map document-centric XML schemas to relational schemas, the mapping is very inefficient (too many tables are involved) and does not scale well.

2.7.2 Products for XML-enabled storage

There are two XML operations you can perform on data stored in XML-enabled storage; publishing data in the storage as XML, and shredding XML documents into that storage.

Local and global XML schemas

One issue with using XML-enabled storage is that you are often constrained as to what XML schemas your documents can use. This is because the object-relational mapping used by the XML Extender and the XML Wrapper requires the XML schema to roughly match the database schema. (SQL/XML does not impose this requirement.)

When your XML schema does not match your database schema —that is, when you cannot map the XML schema to the database schema using the DAD mapping language (XML Extender) or CREATE NICKNAME statements (XML Wrapper) — then you will need two XML schemas: a *local XML schema* and a *global XML schema*. The local XML schema matches the database schema and can be mapped to that schema. The global XML schema is the schema that your applications use. That is, it is the schema of XML documents that you exchange with other applications and databases.

To convert between these two schemas, your application generally uses XSLT. That is, it uses XSLT to convert incoming documents from the global XML schema to the local XML schema. Similarly, it converts outgoing documents from the local XML schema to the global XML schema. This is shown in Figure 2-2.

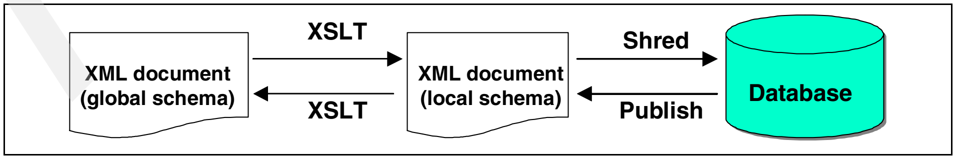


Figure 2-2 Transforming XML documents between local and global schemas

Publishing relational data as XML

If you want to publish relational data as XML, the best choice is to use SQL/XML. SQL/XML is extremely flexible and allows almost arbitrary XML documents to be constructed. This saves you from having to use local (database-specific) and global XML schemas — you can usually publish directly to the global XML schema.

The second choice is to use the publishing capabilities of DB2 XML Extender. There are two possibilities here: SQL statement mapping and RDB node mapping. Of these two, SQL statement mapping is probably easier to use for simple documents— you just construct the SELECT statement you want and map it to an XML document. However, since SQL statement mapping DAD files have a number of restrictions (see “SQL statement mapping DAD documents” on page 29), it makes them less suited for more complex documents. RDB node mapping may more difficult to get your arms around because XML Extender constructs the SELECT statement(s) for you (based on the RDB nodes in your DAD file). For example, you cannot use GROUP BY or ORDER BY clauses in RDB node mapping.
On the other hand, you must use RDB node mapping if you also want to shred XML documents that use the same XML schema. Regardless of which type of mapping you use, you will often need to use XSLT to transform the resulting XML document to one that uses your global XML schema.

The third choice is to write your own code. There are two choices here: writing code that is specific to a particular XML schema and database schema, and writing generic code that is driven by external information, such as a mapping file. The first allows you to easily handle complex XML documents, but can be expensive to maintain. The second is easy if you use the table-based mapping, but that limits the XML documents you can handle. It is significantly harder if you use more complex mappings, such as an object-relational mapping. Perhaps the best compromise when you decide to write code yourself, is to write generic code that uses the table-based mapping and use XSLT to transform between a local, table-based XML schema and a global XML schema.

Shredding XML documents into relational tables

If you want to shred XML documents into relational tables, the best choice is to use DB2 XML Extender with an RDB node mapping (there is no SQL based mapping when shredding). You can also use the XML Wrapper to shred XML documents. This requires you to insert the data for each table separately, but does allow you to shred some documents that cannot be handled by the XML Extender due to referential integrity constraints.

You can also write your own code to shred XML documents into relational tables. As with writing your own code to publish relational data as XML, the best compromise when writing all the code yourself, is to write generic code that uses the table-based mapping and use XSLT to transform between a local, table-based XML schema and a global XML schema.

2.7.3 Products for native XML storage

There are three XML operations you can perform on documents stored in native XML storage; storing the documents, querying them, and updating them.

Storing complete XML documents

If you want to store complete (intact) XML documents, the best choice is to use DB2 XML Extender, and store the documents in an XML column of type XMLVARCHAR, XMLCLOB (XMLDBCLOB if the database allows DBCS), or XMLFILE. This allows you to use side tables to index the documents.

You can also store complete documents in another type of text column, such as a VARCHAR or CLOB column. However, there seems little reason to do this, especially since you can use the XMLVARCHAR, XMLCLOB, XMLDBCLOB, and XMLFILE data types without having to enable an XML column (although you still must enable the database for XML Extender).

Querying XML documents

There are several ways to query complete XML documents. The kind of query you want to run determines how you should store and query the XML document.

If you want to query individual values —that is, element or attribute values — in the XML document, you can:

- Store the document in an XML column, construct side tables for the data you want to query, and query the side tables. This is the best choice for several reasons.
  First, by querying the side tables, you avoid re-parsing the XML document.
  Second, the side tables can use any data type (date, integer, float, and so on), so extracted data values can be stored in the type that makes most sense for them.
  Third, the side tables are always up to date, as the XML Extender updates them each time a document is inserted (or re-inserted) into an XML column.
- Store the document in a text column or an XML column and query it with one of the location path-based extraction functions that come with XML Extender. The main advantage of this is that you can perform ad hoc queries, rather than having to decide what data you want to query when you create the table. In addition, the extraction functions can convert data in the XML document to the type of your choice, such as an integer, date, or float. The disadvantage is that the XML document must be parsed, so performance may be an issue.

- Store the document in a text column or an XML column and perform a structured query with Net Search Extender. While structured queries are similar to XPath queries or querying side tables, there are three significant differences.
  First, Net Search Extender only tests for the existence of a value; it does not return the value itself.
  Second, all comparisons are text based. Thus, "2", "02", and "2.0" are different values.
  Third, comparisons are whether an element or attribute *contains* a given value, rather than *strictly equalling* that value.
- Wrap the document using XML Wrapper and query a subset of it as a table. The advantage of this is that it allows you to store the document in a column or in the file system, or to use documents retrieved from the Web. The disadvantage is that the XML document must be parsed, so performance may be an issue.

If you want to perform a full-text search:

- Store the document in a text column or an XML column, create a full-text index on the column, and perform a full-text query with Net Search Extender.

Updating XML documents

If you want to update XML documents, you can:

- Store the document in an XML column and update the column with the location path-based update function defined by the XML Extender. The advantage of this is that you can easily specify the value you want to change. There are also several disadvantages.
  First, the XML document must be parsed, so performance may be an issue.
  Second, even though only a single value (element or attribute) is changed, the entire document is re-written to the column, again affecting performance.
  Third, the document may be changed. Because the document is converted into its internal representation, updated, and serialized again, some of its content may have changed during the processing of the document.
  Fourth, not all XPath functionality is supported in location path expressions.
- Store the document in a text or XML column, retrieve the document, update it locally, and re-write the document to the column. The advantage of this is that you can make any changes you want. There are several disadvantages as well.
  First, the XML document must be parsed, so performance may be an issue.
  Second, the entire document is re-written to the column, again affecting performance.
  Third, writing the code to update the document may be a lot of work.

3

Designing XML and database schemas

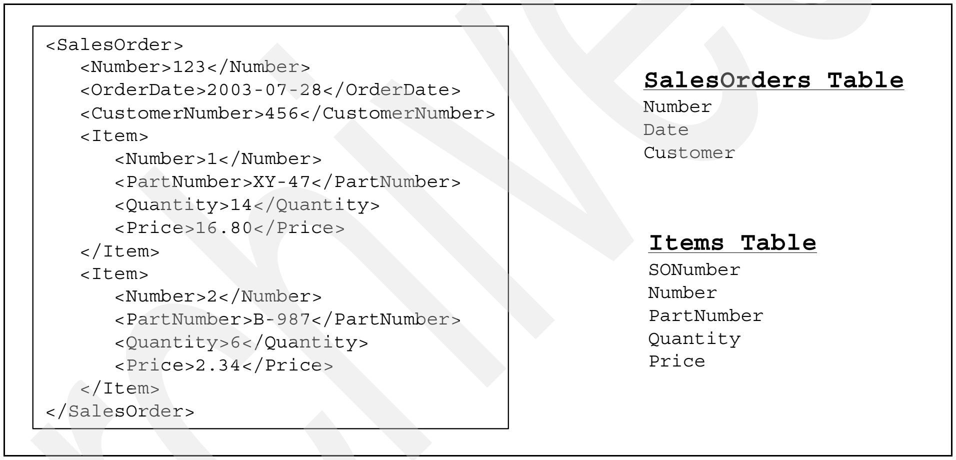
This chapter discusses what XML and/or database schemas you need to design. It is divided into the following sections:

- Local and global XML schemas
- Determining what schemas you need to design
- Whether your XML schema matches your database schema
- Creating an XML schema from a database schema
- Creating a database schema from an XML schema
- Designing an XML schema

3.1 Local and global XML schemas

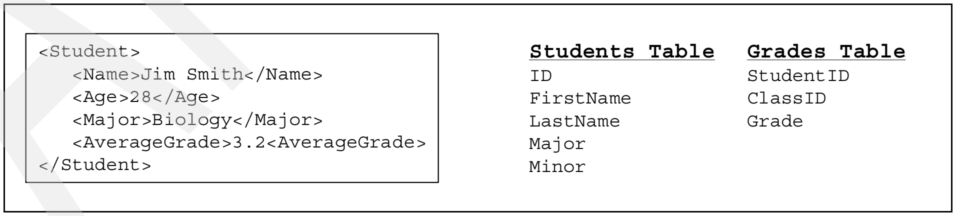
As was briefly discussed in 2.7.2, “Products for XML-enabled storage” on page 52, one issue with using XML-enabled storage is that you are often limited as to what XML schemas your documents can use. This is because the object-relational mapping used by DB2 XML Extender and the XML Wrapper requires the XML schema and the database schema to roughly "match.” This is rarely a problem with SQL/XML, since that language is flexible enough to construct almost any XML document. However, SQL/XML can only be used to publish relational data as XML. It cannot be used to shred XML documents.

For example, the schemas of the XML document and relational tables in Figure 3-1 match while the schemas of the XML document and relational tables in Figure 3-2 do not match.



```
<SalesOrder>
   <Number>123</Number>
   <OrderDate>2003-07-28</OrderDate>
   <CustomerNumber>456</CustomerNumber>
   <Item>
      <Number>1</Number>
      <PartNumber>XY-47</PartNumber>
      <Quantity>14</Quantity>
      <Price>16.80</Price>
   </Item>
   <Item>
      <Number>2</Number>
      <PartNumber>B-987</PartNumber>
      <Quantity>6</Quantity>
      <Price>2.34</Price>
   </Item>
</SalesOrder>
```

Figure 3-1 XML document and relational tables with matching schemas



```
<Student>
   <Name>Jim Smith</Name>
   <Age>28</Age>
   <Major>Biology</Major>
   <AverageGrade>3.2<AverageGrade>
</Student>
```

Figure 3-2 XML document and relational tables with non-matching schemas

It is common for the XML schema to not match the database schema, because the database schema and the XML schema are often defined by two different groups of people. For example, the XML schema could be an industry-standard schema, while the database schema is designed to work with local applications. Or the XML schema could be designed to share data among many departments, while each department has their own database schema.

But even when a single group of people has control over both the XML schema and the database schema, the schemas might not match. For example, the XML schema might be normalized for ease of use, while the database schema might be denormalized for better performance. Or the XML schema is designed to work with newer applications that use XML's hierarchical data structures, while the database schema was be designed to work with older applications that use relational structures.

When your XML schema does not match your database schema, you will need two XML schemas: a *local XML schema* and a *global XML schema*.
The local XML schema is used when transferring data to and from the database, and must match the database schema.
The global XML schema is used by your applications, as well as to exchange data with other applications or databases. It might be an industry-standard schema, or a schema that all external users of your XML documents have agreed upon.

When you are using local and global XML schemas, your application must transform incoming documents from the global schema to the local schema before storing the data in those documents in the database. Your application must also transform outgoing documents from the local schema to the global schema after those documents have been constructed from data in the database (Figure 3-3).

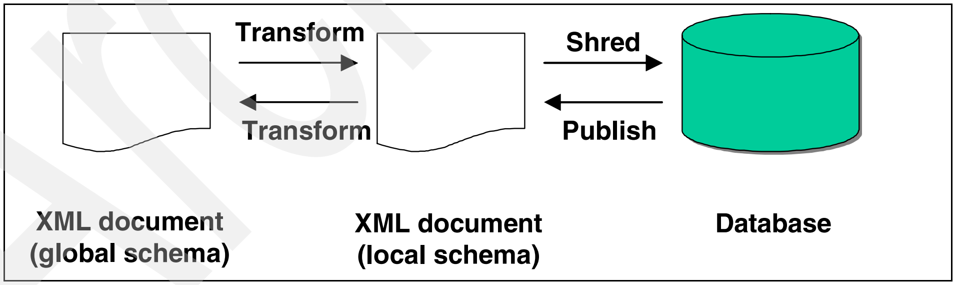


Figure 3-3   Transforming XML documents between local and global schemas

There are several ways to transform XML documents. These are:

- XSLT. This is the most common way to transform XML documents.

  http://www.w3.org/TR/xslt

  The advantage of XSLT is that it is a standard technology and is widely available. Furthermore, it only requires you to write XSLT stylesheets, not code, in order to transform documents. The disadvantage of XSLT is that it can be slow and may need to read the entire document into memory. The latter problem prohibits its use with very large XML documents.

- Custom SAX applications. If your transformation is simple and can be performed while reading through the document from start to finish, then you might be able to write a simple SAX program to perform the transformation. The advantage of SAX is that it is generally faster than XSLT. Furthermore, it does not read the entire document into memory, so it can be used with arbitrarily large documents. For more information about custom SAX applications, see 9.1.3, “Transforming XML documents with SAX” on page 256.
- Third-party transformation packages. There are some third-party packages available for performing specific types of transformations. For example, the Regular Fragmentations package uses regular expressions to create multiple elements from a single element. For example, you can use this to create Year, Month, and Day elements from a Date element.

  http://regfrag.sourceforge.net/

Of course, if you can use a single XML schema, as is generally the case when using SQL/XML, and sometimes the case when using the XML Extender or the XML Wrapper, then you should use only a single XML schema. The reason is that transformations can be expensive, so your application will generally perform better without them.

3.2 What schemas do you need to design?

In this section, we discuss what XML schemas and database schemas you need to design. In most cases, you will already have a database schema and need to design one or more XML schemas. In a few cases, you will have an XML schema and need to design a corresponding database schema.

3.2.1 On using industry-standard XML schemas

One of the first design decisions you will need to make is whether to use an industry-standard XML schema like ebXML (the XML version of EDI), FIXML (the XML version of the Financial Information eXchange protocol), or ACORD XML for Life Insurance.

The primary advantage of industry-standard XML schemas is interoperability. If you decide to use one of these schemas, you will (hopefully) be able to exchange data with other companies that use the same schema. A secondary advantage is that somebody else has already designed the schema —designing a robust XML schema is a potentially complex task and can take months or years.

Unfortunately, industry-standard XML schemas also have drawbacks. For example:

- The schema might not have everything you need. One solution to this problem is to add the element types and attributes that you and your trading partners need. Unfortunately, this means that your schema will no longer be standard, so it might not work with standard software (if any exists) and you might not be able to exchange data with new partners. If you do add new element types and attributes, you should always add element types and attributes that are in your own XML namespace.

  > **Note:** *XML namespaces* are a way to create universally unique element types and attribute names. Each name has two parts: a URI and a local name, such as `http://www.w3.org/1999/xsl/transform` and `stylesheet`. For more information, see http://www.w3.org/TR/REC-xml-names.

- The schema requires element types or attributes that you do not need. When shredding incoming XML documents, you should be able to ignore the unneeded element types and attributes. When publishing data as XML, you will need to add the elements or attributes to the outgoing document, perhaps using default values. This is easy to do with SQL/XML, as you can use a constant value. If you are using DB2 XML Extender, you will need to transform the document to add the element or attribute.
- You are using DB2 XML Extender or an XML Wrapper, and the industry-standard schema does not match your database schema. This will be true except in the uncommon case that you have designed your database schema specifically to match the industry-standard XML schema. The solution to this is to use a local (database-specific) schema as was described earlier in this chapter. This means you will have to transform documents between the local schema and the industry-standard (global) schema.

In spite of these drawbacks, you should still consider an industry-standard XML schema. Even if you need to modify the schema to work with specific trading partners, just using an industry-standard schema will often create more opportunities than problems.

For a further discussion of these issues, see "Interoperability Without Authority: Loosely Coupled XML Processing without Shared Schemas" by Michael Champion. The paper is available on the Web at:

http://idealliance.org/papers/xml02/dx_xml02/papers/04-04-04/04-04-04.html

For a list of industry-standard XML schemas, see the XML.org Web site at:

http://www.xml.org

or the Cover Pages at:

http://xml.coverpages.org

The next sections will consider what you need to do in specific situations.

3.2.2 You have a database schema

Perhaps the most common starting point for building XML/database applications is that you have an existing database, and now want to use XML as a way to exchange data. You have three choices:

- Use an industry-standard XML schema. The advantages and disadvantages of this were discussed in the previous section. Because you have an existing database schema, you will almost certainly need to create a local schema if you want to use XML Extender or XML Wrapper.
- Create an XML schema that matches your relational schema. While this is the simplest option and gives the best performance, you should consider this option carefully before doing it. The main problem is that the resulting XML schema may not be the best XML schema for your users. For example, suppose your database uses lookup tables. This will result in an extra (and unnecessary) element in your XML document corresponding to the lookup table.
- Create an XML schema from scratch. If there are no industry-standard XML schemas that apply to your data, then this is probably the best option, as the resulting schema is most likely to meet the needs of your users. If you are only using SQL/XML, then the only disadvantage of this option is that designing an XML schema takes time. If you are using DB2 XML Extender or an XML Wrapper, then an added disadvantage is that you may not be able to map the resulting schema to your database schema directly. As a result, your application will have to transform documents between the new (global) schema and a local (database-specific) schema.

3.2.3 You have an XML schema

A less common situation is that you have an XML schema and want to build structures in your database to store the data in documents conforming to that schema. If you have a set of XML documents you want to store in your database, you need to decide if you really want to store the data from these documents into a set of schema-specific tables, or you are simply interested in storing the documents themselves. In other words, are you interested in XML-enabled storage or native XML storage?

If you want to use native XML storage —that is, you simply want to store the XML documents— you should consider using the XML column feature of DB2 XML Extender. This allows you to store whole documents in a single column. The documents can then be indexed and searched using side tables. For more information about XML columns, see Chapter 4, “Storing whole XML documents” on page 93.

If you want to use XML-enabled storage —that is, you want to shred the documents into a set of relational tables— you will need to design those tables. Although it is possible to design a database schema that exactly matches your XML schema, this is often a bad idea. The reason is that XML schemas often include structure that is not strictly necessary. This results in inefficient storage of the data. For example, consider the following customer element type in Example 3-1.

Example 3-1 Customer element type

```
<Customer>
   <Number>456</Number>
   <Address>
      <Street>123 Main St.</Street>
      <City>Chicago</City>
      <State>IL</State>
      <PostCode>60609</PostCode>
      <Country>US</Country>
   </Address>
</Customer>
```

Creating a database schema that exactly matches the XML schema for this element type means that you will need two tables: one for customers and one for addresses. However, it generally makes more sense to store the customer's address directly in the customer table.

A better solution is to start with a database schema that matches your XML schema and then modify it to create a more efficient database schema. Depending on your modifications, you may then need to create a local XML schema that matches your database schema and write XSLT stylesheets to transform documents between your original (global) schema and this local schema.

3.2.4 You have both database and XML schemas

This is another common case. For example, when you have an existing database and someone external to your organization requests data in the form of a specific

XML document, such as one that uses an industry-standard XML schema. Or you gather a certain type of data, and a new source of that data becomes available in the form of XML documents.

If you are only publishing data to XML, you should be able to write SQL/XML statements that create documents conforming to the desired XML schema.

If you are shredding XML documents into your database, then it is very likely that your XML schema will not match your database schema. In this case, you will need to write a local XML schema that matches your database schema and write XSLT stylesheets to transform documents between the global XML schema and this schema.

3.3 Does my XML schema match my database schema?

If you are using DB2 XML Extender or the XML Wrapper, your XML schema must match your database schema. This is because these products use an object-relational model that defines a fairly rigid mapping between the XML schema and the database schema. Although the languages used to specify these mappings —DAD documents in the XML Extender and CREATE NICKNAME statements in the XML Wrapper— support minor transformations, they are not general transformation languages.

Note: If you are using SQL/XML to publish data to XML, your XML schema does not need to match your database schema. This is because SQL/XML is very flexible and, assuming the necessary data is in your database, you should be able to construct almost any document you need.

The following procedure may be used to determine if an XML schema matches a database schema. Note that this procedure is sufficient, but not necessary. That is, if an XML schema and a database schema satisfy this procedure, it will be possible to map the XML schema to the database schema. However, it may still be possible to map an XML schema to a database schema even if they do not satisfy this procedure.

We will illustrate this procedure with the DTD shown in Example 3-2,

Example 3-2 Sample DTD

```
<!ELEMENT SalesOrders (SalesOrder+)>

<!ELEMENT SalesOrder (OrderDate, Customer, Item+)>
<!ATTLIST SalesOrder
```

```
          Number CDATA #REQUIRED>

<!ELEMENT Customer (Name, Address)>
<!ATTLIST Customer
          Number CDATA #REQUIRED>

<!ELEMENT Address (Street, City, State, PostCode, Country)>

<!ELEMENT Item (Part, Quantity, Price)>
<!ATTLIST Item
          Number CDATA #REQUIRED>

<!ELEMENT Part (Name, Description)>
<!ATTLIST Part
          Number CDATA #REQUIRED>

<!-- Element type declarations for
        PCDATA-only element types not shown. -->
```

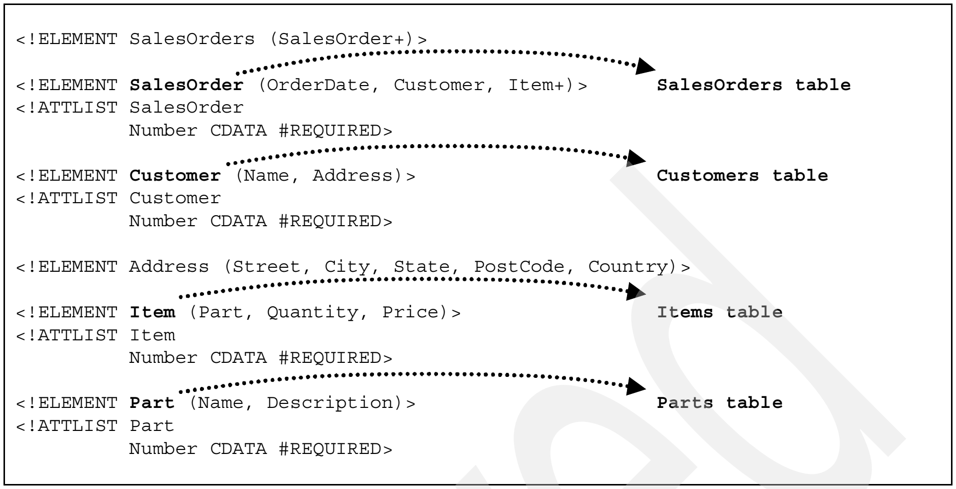
and the database schema shown in Example 3-3.

Example 3-3 Sample database schema

```
SalesOrders (Number, Date, CustomerNumber)
Items       (SONumber, ItemNumber, PartNumber, Quantity, Price)
Customers   (Number, Street, City, State, PostCode, Country)
Parts       (Number, Name, Description)
```

To determine if your XML schema matches your database schema, first perform the following procedure:

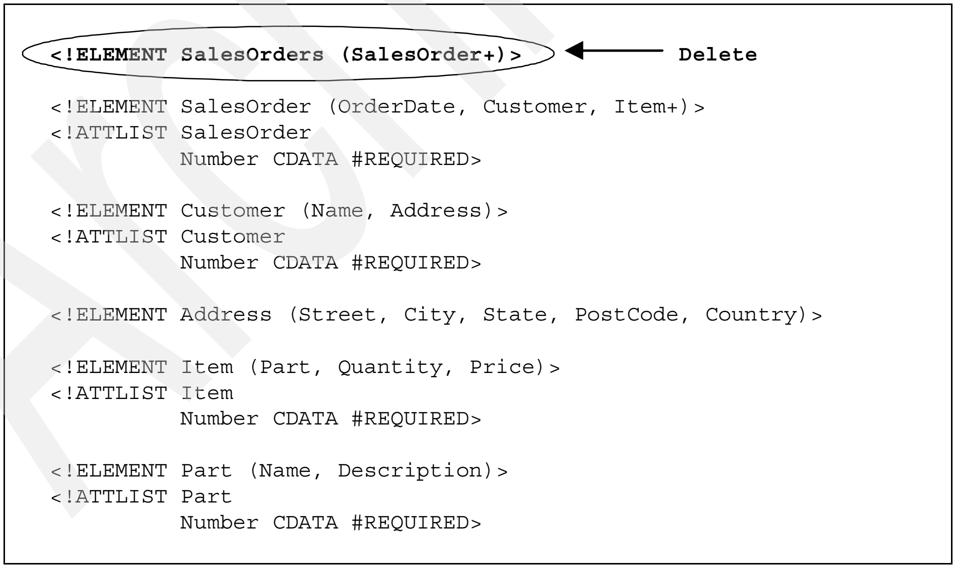
1. Map complex element types to tables as desired. For example, map the SalesOrder element type to the SalesOrders table, the Item element type to the Items table, and so on (Figure 3-4 on page 66).



```
<!ELEMENT SalesOrders (SalesOrder+)>

<!ELEMENT SalesOrder (OrderDate, Customer, Item+)>
<!ATTLIST SalesOrder
          Number CDATA #REQUIRED>

<!ELEMENT Customer (Name, Address)>
<!ATTLIST Customer
          Number CDATA #REQUIRED>

<!ELEMENT Address (Street, City, State, PostCode, Country)>

<!ELEMENT Item (Part, Quantity, Price)>
<!ATTLIST Item
          Number CDATA #REQUIRED>

<!ELEMENT Part (Name, Description)>
<!ATTLIST Part
          Number CDATA #REQUIRED>
```

Figure 3-4 Map complex element types to tables

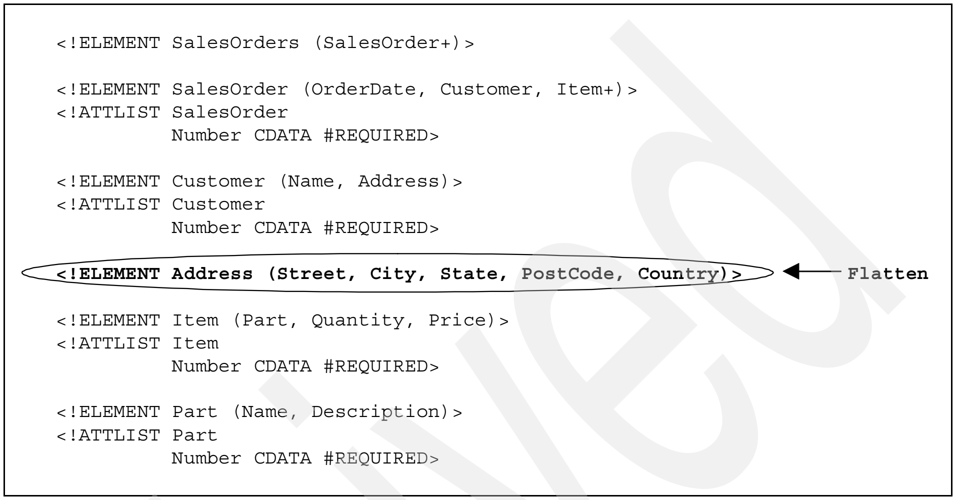
2. If the root element type is not mapped and does not have any attributes, and only has complex element types as children, then remove it. Repeat this step until: (a) the root element type has one or more attributes, (b) the root element type has one or more children that only contain PCDATA, or (c) the root element type is mapped to a table (Figure 3-5).



```
<!ELEMENT SalesOrders (SalesOrder+)>

<!ELEMENT SalesOrder (OrderDate, Customer, Item+)>
<!ATTLIST SalesOrder
          Number CDATA #REQUIRED>

<!ELEMENT Customer (Name, Address)>
<!ATTLIST Customer
          Number CDATA #REQUIRED>

<!ELEMENT Address (Street, City, State, PostCode, Country)>

<!ELEMENT Item (Part, Quantity, Price)>
<!ATTLIST Item
          Number CDATA #REQUIRED>

<!ELEMENT Part (Name, Description)>
<!ATTLIST Part
          Number CDATA #REQUIRED>
```

Figure 3-5 Remove unmapped root elements

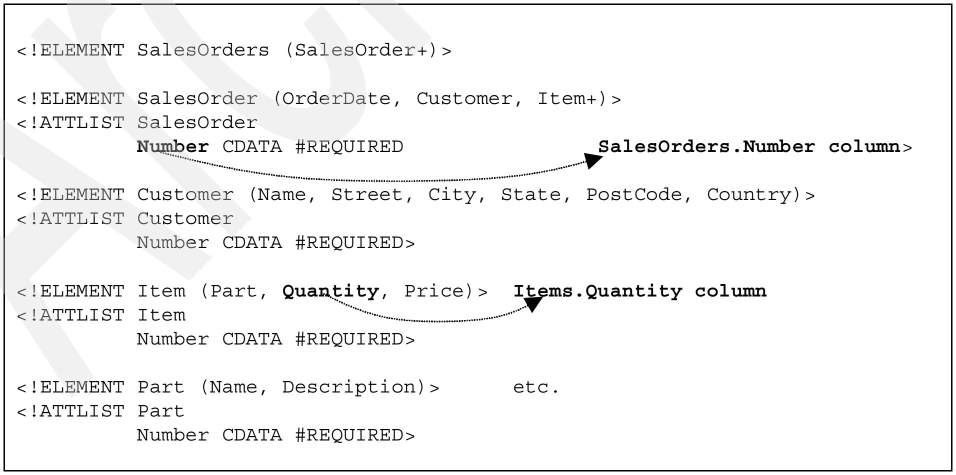
3. Recursively flatten any remaining complex element types. That is, move the attributes and child elements of the unmapped element type to its parent element type. For example, you can flatten the Address element type (Figure 3-6).



```
<!ELEMENT SalesOrders (SalesOrder+)>

<!ELEMENT SalesOrder (OrderDate, Customer, Item+)>
<!ATTLIST SalesOrder
          Number CDATA #REQUIRED>

<!ELEMENT Customer (Name, Address)>
<!ATTLIST Customer
          Number CDATA #REQUIRED>

<!ELEMENT Address (Street, City, State, PostCode, Country)>  <—— Flatten

<!ELEMENT Item (Part, Quantity, Price)>
<!ATTLIST Item
          Number CDATA #REQUIRED>

<!ELEMENT Part (Name, Description)>
<!ATTLIST Part
          Number CDATA #REQUIRED>
```

Figure 3-6 Flatten unmapped complex element types

4. Map attributes and references to simple child elements to columns as desired. For example, map the Number attribute of the SalesOrder element type to the Number column of the SalesOrders table and the Quantity child element to the Quantity column of the Items table (Figure 3-7).



```
<!ELEMENT SalesOrders (SalesOrder+)>

<!ELEMENT SalesOrder (OrderDate, Customer, Item+)>
<!ATTLIST SalesOrder
          Number CDATA #REQUIRED             SalesOrders.Number column>

<!ELEMENT Customer (Name, Street, City, State, PostCode, Country)>
<!ATTLIST Customer
          Number CDATA #REQUIRED>

<!ELEMENT Item (Part, Quantity, Price)>  Items.Quantity column
<!ATTLIST Item
          Number CDATA #REQUIRED>

<!ELEMENT Part (Name, Description)>      etc.
<!ATTLIST Part
          Number CDATA #REQUIRED>
```

Figure 3-7 Map attributes and references to simple child elements to columns

Your XML schema matches your database schema if the modified XML schema meets *all* of the following criteria:

- Attributes and child element types are mapped to columns in the table of their parent element type.
- No flattened element type can occur more than once in its parent. Element types that occur more than once in their parent must be mapped to separate tables.
- No mapped element types or attributes have an unmapped ancestor.
- No two element types are mapped to the same table and each element type mapped to a table is mapped only once.
- No two element types or attributes are mapped to the same column and each element type or attribute mapped to a column is mapped only once.
- Any child element type mapped to a column can occur at most once in its parent.
- If you are publishing data as XML, then all unmapped element types and attributes are optional. If you are shredding XML documents, then all unmapped columns are nullable or have a default.
- If you know the data type of an element type or attribute (such as when you are using XML Schemas), then it must be possible to convert this type to the data type of the column to which the element type or attribute is mapped and vice versa.
- If a child element type is mapped to a table, then there must be a candidate key / foreign key relationship between the table of the parent element type and the table of the child element type.

  In addition, if you are shredding XML documents with the XML Extender, then:

  - The candidate key must be a primary key,
  - The candidate key must be in the table of the parent element type, and
  - All columns in the primary key must be mapped.

3.4 Creating an XML schema from a database schema

This section explains how to create an XML schema from a database schema. The resulting schema will match the structures in the database. For example, if the database schema is non-normal, the XML schema will also be non-normal. Whether you want to use this schema as your global XML schema —that is, the schema you use when exchanging data with other applications— is a separate design decision.

3.4.1 The algorithm

To create an XML schema from a database schema, you need to do the following steps. Note that WebSphere Studio can do this for you automatically; see Chapter 12, “XML and database tools in Application Developer” on page 337 for more on this.

1. Determine what tables you want to use and arrange these tables into a hierarchy. That is, choose a root table and determine which keys you will use to link to the other tables in the hierarchy.
2. For each table, create a complex element type.
3. From each table, decide which data columns you need. Data columns are columns that are not primary key columns and are not part of a foreign key that is used to join the tables in your hierarchy.
4. For each data column you choose, create an attribute or a simple element type. Add references to these element types to the content model (a sequence) of the table's element type. If the column is nullable, then the attribute or reference is optional. If you are using XML Schemas, be sure to specify the data type of the attribute.
5. For each primary key column, decide whether to add the column to the XML schema. Primary keys fall into two categories: object identifiers and keys with business significance. Object identifiers simply identify a row in the database. They might be unique on a per-table or per-database basis. Keys with business significance are things like part numbers, flight numbers, and employee IDs.

   As a general rule, you should include object identifiers in the XML schema only if you will need them at a later time to identify the row, such as for an update. (Note that the XML Extender does not support updates for XML collections, so you will need to write your own code to do this.) Otherwise, there is no point in including them as they have no significance outside the database.

   You should always include keys with business significance in your XML schema.

   For each primary key column you choose to add to the XML schema, follow the procedures in step 4.
6. For each table in the hierarchy, add a reference to the content model of the element type corresponding to the parent table. If the relationship between the parent table and the child table is one-to-one, the reference may be required or optional. If the relationship is one-to-many, the reference may be zero-or-more or one-or-more.

   For information about how to handle many-to-many and many-to-one relationships, see the section “Third normal form” on page 82.

3.4.2 Using the generated XML schema with DB2 XML Extender

If you want to use the XML schema created by this algorithm to shred documents with XML Extender, then you must be sure that the following is true:

- No two leaf nodes (attributes or PCDATA-only element types) may have the same name.
- All non-nullable columns that do not have defaults must have corresponding element types or attributes in the XML schema. The only exception to this is foreign key columns used to join tables in your hierarchy.
- The tables in the hierarchy must be arranged so that the primary key used to join parent and child tables is always in the parent table.

3.5 Creating a database schema from an XML schema

This section explains how to create a database schema from an XML schema. The resulting database schema will match the structures in the XML schema. For example, if the XML schema is non-normal, the database schema will also be non-normal.

3.5.1 Creating a local XML schema from a global XML schema

It is often a bad idea to create a database schema directly from your global XML schema —that is, the schema for the documents that you use to exchange data with other applications. The reason is that the global schema is designed for exchanging data in a particular situation and its structure might not represent the best structure for storing data in the database. For example, the global XML schema might use wrapper elements to improve human readability but that structure does not need to be duplicated in the database.

Therefore, before you create a database schema from your global XML schema, you should first create a local XML schema. In many ways, this is equivalent to designing your database schema, since the structure of the local XML schema will match your database schema. Here are some things to do when creating a local XML schema from a global XML schema:

- Remove outer wrapper elements types. An *outer wrapper element type* is a wrapper element type whose ancestors are all wrapper element types. The most common use of an outer wrapper element type is to satisfy the rule that an XML document has a single root element. For example, in the following XML schema (Example 3-4 on page 71), SalesOrders is an outer wrapper element type:

Example 3-4 Outer wrapper element type

```
<!ELEMENT SalesOrders (SalesOrder+)>
<!ELEMENT SalesOrder (OrderDate, Customer, Item+)>
<!ATTLIST SalesOrder
             Number CDATA #REQUIRED
```

Similarly, you might have two or more outer wrapper element types that serve only to group the actual data. For example, in the following XML schema (Example 3-5), Orders, SalesOrders, and PurchaseOrders are all outer wrapper elements types:

Example 3-5 More outer wrapper element types

```
<!ELEMENT Orders (SalesOrders, PurchaseOrders)>
<!ELEMENT SalesOrders (SalesOrder+)>
<!ELEMENT PurchaseOrders (PurchaseOrder+)>
```

Outer wrapper element types generally do not have any attributes or child elements that can be used as primary keys in a database. In other words, the collection of data that they represent —a group of sales orders, a set of employees, and so on— does not have any identity in and of itself. It merely represents a set of data that happened to be grouped together for some non-significant purpose, such as replicating a database.

Because of this, outer wrapper element types should be removed from the local XML schema, as they do not have a corresponding structure in the database. Because outer wrapper element types are often root element types, this may result in the XML schema being split into two or more XML schemas.

- Remove unneeded data. Global XML schemas sometimes contain data that is not relevant to your company. This is particularly true for industry-standard XML schemas, which are designed to satisfy the needs of many companies. Remove any element types or attributes that contain data not needed by your company.
- Add needed data. Global XML schemas also sometimes do not contain data that is relevant to your company. Again, this is particularly true with industry-standard XML schemas. If you need data that is not in the global XML schema, add element types or attributes for this data to your local XML schema. (You can also add it later to the database schema.)
- Flatten inner wrapper element types. *Inner wrapper element types* are wrapper element types that occur inside a data structure. They exist only for

clarity and ease of processing and do not have a corresponding structure in the database. Inner wrapper element types can occur either zero or one time in their parent and can often be recognized because they have no attribute or child element that can be used as a primary key in a database. For example, the Address element type in the following XML schema is an inner wrapper element type:

```
<!ELEMENT Customer (Number, Address)>
   <!ELEMENT Address (Street, City, State, PostCode, Country)>
```

Inner wrapper element types should be flattened. That is, they should be removed from the schema, their attributes moved to their parent element type, and their children moved to the content model of their parent element type. (To prevent naming conflicts, you may need to change the names of the attributes or element types that you move.) For example:

```
<!ELEMENT Customer (Number, Street, City, State, PostCode, Country)>
```

- Think again. Remember that your local XML schema has the same structure that your database schema will have. Before constructing a database schema from it, you should be sure that the resulting database schema is the best way to store your data. Remember, complex element types will be converted to tables and attributes and simple element types will be converted to columns.

3.5.2 Creating a database schema from a local XML schema

To create a database schema from a local XML schema, you need to do the following. Note that WebSphere Studio can do this for you automatically. See 12.2.3, “XML Schema, table DDL, and DDT” on page 344 for details.

1. For each complex element type, create a table.
2. For each attribute, create a column in the table of the parent element type. Set the column metadata as follows:
   - If the attribute is optional, the column is nullable.
   - If the attribute has a default, the column should have the same default.
   - If you are using XML Schemas, set the column data type to the SQL data type that is closest to the attribute's data type. If you are using DTDs, set the column data type to VARCHAR or CLOB, depending on the expected values of the attribute.
3. For each simple child element that can occur zero or one time in its parent element type, create a column in the table of the parent element type. (If a child element appears in a subgroup, this may affect the number of times it can appear in its parent.) Set the column metadata as follows:
   - If the child element can occur zero times, the column is nullable.

   - If you are using XML Schemas and the child element has a default, the column should have the same default.
   - If you are using XML Schemas, set the column data type to the SQL data type that is closest to the child element's data type. If you are using DTDs, set the column data type to VARCHAR or CLOB, depending on the expected values of the child element type.

4. For each pair of complex element types that are a parent and child, determine the primary key and foreign key used to link the tables corresponding to these element types. Note that the primary key can be in either table. There are two potential problems here:
   - If there are no columns that can be used as the primary key, then DB2 XML Extender cannot be used to shred documents corresponding to the local XML schema. The solution to this is to add a child element or attribute to the local XML schema that can be used as the primary key. The value of this element or attribute can be generated before the document is passed to XML Extender, such as with an XSLT extension function.
   - If the primary key is in the table corresponding to child element type, then XML Extender cannot be used to shred documents corresponding to the local XML schema. There are two possible solutions to this.

     First, rearrange the document so that the child element type is in a separate part of the document or in a separate document. (For more information, see "Third normal form" on page 82.)

     Second, use the XML Wrapper to shred the document. (For more information, see 2.4, "XML wrapper" on page 42.)
5. For each uniqueness constraint in the XML schema, such as an ID attribute in a DTD or a unique element in an XML Schema, consider adding a uniqueness constraint to the database schema. Note that uniqueness constraints in XML schemas only guarantee uniqueness within a single document or part of that document, not across all documents that correspond to the XML schema. Therefore, they might not correspond to uniqueness constraints in the database.
6. For each referential constraint in the XML schema, such as ID/IDREF attributes in a DTD or key/keyref elements in an XML schema, consider adding a candidate key / foreign key constraint to the database schema. Note that referential constraints in an XML schema corresponds to referential integrity in the database only if the key value is unique across all documents that correspond to the XML schema.

3.6 Designing XML schemas

In this section we describe some general rules for designing XML schemas. It applies primarily to designing global XML schemas -- that is, schemas for XML documents that will be used by applications other than your own. (Because local XML schemas must match your database schema, they may not follow all of these rules.)

This section is not meant to be a complete discussion of designing XML schemas. For additional information, see any guidelines for designing object-oriented systems, as the structures found in data-centric XML schemas are essentially objects (a recommended Web site is http://www.ambysoft.com).

Finally, remember that designing XML schemas, like designing database schemas or object-oriented schemas, is a difficult process and may take a significant amount of time. Not only are you trying to characterize your data and provide future migration routes, you are trying to resolve political and technical differences among a wide variety of users.

Next, we discuss the following topics:

- Who will use your XML schema?
- What XML schema language should you use?
- General guidelines for designing XML schemas
- Normalizing your XML schema
- XML schema styles to avoid
- XML schema structures not supported by the XML Extender

3.6.1 Who will use the XML schema?

Before you design your XML schema, you should consider who will be using your XML documents. Will they be local to a single application? Used within your department? Used with current trading partners? Exposed to the entire world through the Web?

With each wider set of potential users, you bring in a larger community and a larger set of requirements. Hence, the design of an XML schema becomes more complex and more political as competing requirements make technical solutions more difficult. This is best illustrated by the prediction that industry-standard XML schemas would arrive a year or two after the XML 1.0 recommendation was published in 1998. In fact, most are only just now [2004] arriving.

One thing to remember when you are trying to resolve differences between multiple user communities is that you might not be limited to a single XML schema for a single set of data. This is particularly true when publishing data as XML, where an XML document is just a query result. It is less true when shredding XML documents, as constraints in your database might limit the set of XML documents you can use to populate that database.

3.6.2 What XML schema language should you use?

Another thing to consider is what XML schema language you should use. While a number of these have been proposed, only four seem to be in widespread use today:

- DTDs. These are simple, standard, and widely supported, although some products are replacing DTD support with XML Schema support. Their major disadvantage is that DTDs do not support data types. For more information, see the XML 1.0 recommendation:

  http://www.w3.org/TR/REC-xml

- XML Schemas. These are also standard and widely supported. They support data types and have a variety of reusable constructs, such as complex types, attribute groups, and model groups. They also include support for documentation and application-specific extensions. Their major disadvantage is that they are difficult to learn and read, although editing tools may reduce these problems. For more information, see the XML Schema recommendation:

  http://www.w3.org/TR/xmlschema-1/

- RELAX NG. This is an XML schema language developed through OASIS and is being standardized by ISO. It is generally considered to be easier to learn and more flexible than XML Schemas. However, far fewer products support it. For more information, see the RELAX NG home page:

  http://relaxng.org/

  Note: OASIS is a global consortium that develops e-business standards. For more information, see http://www.oasis-open.org and http://www.xml.org.

- Schematron. This is not a schema language in the sense of the other languages in that it doesn't explicitly define element types and attributes. Instead, it is better thought of as a constraint language. For example, it is possible to specify constraints such as, "When the value of the Sex element is 'M', the value of the Title element must be 'Mr.'" Thus, Schematron is best used as a supplemental language for specifying constraints that cannot be expressed in any of the other languages. For more information, see the Schematron home page:

  http://www.ascc.net/xml/resource/schematron/schematron.html

3.6.3 General guidelines for designing XML schemas

The following are a set of general guidelines for designing XML schemas.

Names

The following are guidelines for constructing XML element type and attribute names.

- Avoid abbreviations. Database schemas often use abbreviations, such as FLTTM and PRTNO. Unfortunately, abbreviations often make sense only to their inventor and may be impossible for non-native speakers to understand. Fortunately, you are not required to use the same names in your XML schema that you use in your database schema. Therefore, you should avoid abbreviations in XML schemas and use understandable names like FlightTime and PartNumber instead. This will make your XML schema easier to read, especially by an international audience. And don't worry about name length: XML compresses well due to repeating structures.
- Use context to distinguish names. If an element type or attribute can logically appear in more than one place, use it in both places rather than creating two different element types or attributes. For example, if you are creating an XML schema for a book, use a single Title element type, rather than creating separate BookTitle, ChapterTitle, and SectionTitle element types.

  Note that this does not apply to different uses of the same data type. For example, if you have separate ship-to and bill-to addresses, you will need to different element types —ShipToAddress and BillToAddress— as context alone cannot distinguish between these uses.
- Use one or more XML namespaces. You should always use XML namespaces, even if there are not any current collisions between your element type names and other element type names. This will make it easier for other people to combine your documents with other documents and give you a good migration path for the future. You should use multiple XML namespaces only if you expect your schema to be modular. That is, if you expect people to reuse subsets of your schema in a well-defined manner.

  > **Note:** DB2 XML Extender and XML Wrapper do not support XML namespaces, so the local (database-specific) schemas used with those products cannot use them. (SQL/XML does support XML namespaces.)

- Do not encode values in names. For example, do not construct names like PartXY-47or Transaction-47-01120-17-6. This confuses data and metadata and is difficult to process. Instead, construct general names like Part or Transaction and use a child element or attribute like Number to provide specific values.

Structures

The following are guidelines for constructing structures in XML schemas.

- Use complex types. If two different element types have the same complex type, then this should be defined separately. For example, BillToAddress and ShipToAddress can both use an Address complex type. This can be done with parameter entities in DTDs, and complex types in XML Schemas.

  This also applies to sets of attributes and elements that do not constitute a complete type, such as a subgroup found in the content models of many different element types. (Note that XML Schemas have separate constructs for such sets: attribute groups and model groups.)

  There are several good reasons for doing this: it reduces the chance for errors as a schema evolves, it promotes schema consistency, and it may allow the reuse of software designed to process a particular complex type.
- Reuse element types. As was mentioned in the section on names, whenever an element type or attribute can appear in more than one context, it should be reused, rather than constructing separate element types or attributes for each context.
- Wrap related elements. The hierarchical structure of XML makes it natural to wrap related elements. In some cases, this simply enhances readability. For example, an Address element can be used to group elements that represent the different parts of an address. (see Example 3-6).

Example 3-6 Using wrap related elements

```
<Customer ID="456">
   <Name>ITSO Insurance, Inc.</Name>
   <Address>
      <Street>123 Main St.</Street>
      <City>Chicago</City>
      <State>IL</State>
      <PostCode>60609</PostCode>
      <Country>US</Country>
   </Address>
</Customer>
```

In other cases, it may make processing easier. For example, items in a sales order can form a repeating subgroup inside a SalesOrder element type:

Example 3-7 Item as repeating subgroup

```
<SalesOrder Number="123">
   <OrderDate>2003-07-28</OrderDate>
   <CustomerNumber>456</CustomerNumber>
   <Item Number="1">
      ...
   </Item>
   <Item Number="2">
         ...
   </Item>
</SalesOrder>
```

or be grouped inside an Items element (Example 3-8).

Example 3-8 Items wrapper

```
<SalesOrder Number="123">
   <OrderDate>2003-07-28</OrderDate>
   <CustomerNumber>456</CustomerNumber>
   <Items>
      <Item Number="1">
            ...
      </Item>
      <Item Number="2">
         ...
      </Item>
   </Items>
</SalesOrder>
```

The latter is not only easier to read, it is easier to process. (When shredding an XML document, the XML Extender requires repeating subgroups to be placed inside a wrapper element.)

- Elements or attributes? A perennial question in XML schema design is whether to use simple element types or attributes. There is no absolute answer for this question and the newer schema languages are removing many earlier reasons. Here are some of the issues:
  - Order. Attributes are always unordered. Elements are always ordered. However, not all applications treat elements as ordered. In fact, for many data-centric documents, sibling order is only significant during validation. That said, XML Schemas have limited support for unordered sibling elements and RELAX NG has full support for unordered sibling elements.

> **Note:** *Sibling order* is the order in which child elements and PCDATA appear in their parent element.

- Data types. DTDs support very limited data typing for attributes and no data typing for simple element types. Both XML Schemas and RELAX NG support data typing for attributes and simple element types.
- Multiple values. DTDs support multi-valued attributes —that is, values separated by white space. Both XML Schemas and RELAX NG support multiple values for attributes and simple element types.
- Repeatability. Attributes can never be repeated. Child elements can always be repeated.
- Structure. Attributes are always scalar-valued. Element types may be scalar-valued or have structure. This is commonly cited as a reason to use element types, since a simple element type may be changed to a complex element type without disrupting the content model of the parent element type. Whether this is actually any less disruptive than changing from an attribute to an element type is not clear.

For more details, see `http://xml.coverpages.org/elementsAndAttrs.html`.

Constraints

The following are guidelines for constraints in XML schemas. The main reasons to add constraints to your XML schemas —even if parallel constraints do not appear in your database schemas— are that they help application developers to better understand your data and map it to the database and they help applications to automate constraint checking. As a general rule, automated constraint checking is better than coding constraints inside your application, since the latter is hidden from view, more difficult to change, and can only be used inside that application.

- Use simple data types. Both XML Schemas or RELAX NG support a large number of simple data types and have facilities for adding additional constraints to these types.
- Use unique and referential constraints. Unique constraints are supported in DTDs with ID attributes and in XML Schemas with the unique and key elements. Referential constraints are supported in DTDs with IDREF attributes and in XML Schemas with the keyref element.

  Unfortunately, both constructs are less than ideal for use with databases. In both cases, uniqueness is limited. ID values must be unique within the document and key and unique values must be unique within a document fragment defined by an XPath expression. Neither guarantees uniqueness across documents, which is what is needed to guarantee uniqueness once data is transferred from an XML document to the database.

ID attributes are further restricted in that their values must match the Nmtoken production in the XML 1.0 recommendation. This requires that values start with a letter or underscore and consist of only letters, underscores, hyphens, and periods. ID attributes therefore cannot be used to represent numeric key values from the database.

- Consider using Schematron to define additional constraints.

Documentation

The following are guidelines for documentation in XML schemas.

- Document your schema. The declarations in an XML schema are not enough to fully describe the element types and attributes in that schema. For example, what are the legal units for a Price element type? Documentation will both improve the quality of your schema and reduce your support load, especially in cases where your XML documents are widely used, such as with Web services. You can document your schema with comments in DTDs and annotation elements in XML Schemas.

3.6.4 Normalizing your XML schema

Normalization is formally defined in terms of relational schemas, not XML schemas. In spite of this, the rules of normalization can be applied to XML schemas, although not always with the technical precision that is used in normalizing relational schemas. For example, most people would agree that the schema for the following XML document (Example 3-9) is not normalized, since the sales order number, order date, and customer number are repeated in each item.

Example 3-9 Unnormalized XML document

```
<SalesOrder>
   <Item SONumber="123" ItemNumber="1">
      <OrderDate>2003-07-28</OrderDate>
      <CustomerNumber>456</CustomerNumber>
      <PartNumber>XY-47</PartNumber>
      <Quantity>14</Quantity>
      <Price>16.80</Price>
   </Item>
   <Item SONumber="123" ItemNumber="2">
      <OrderDate>2003-07-28</OrderDate>
      <CustomerNumber>456</CustomerNumber>
    = <PartNumber>B-987</PartNumber>
      <Quantity>6</Quantity>
      <Price>2.34</Price>
   </Item>
</SalesOrder>
```

This section assumes that you have a basic understanding of normalization. If you do not, you can find articles about normalization in database textbooks or on the Web, such as at:

http://www.utexas.edu/cc/database/datamodeling/rm/rm7.html

First normal form

First normal form requires two things:

- Column values are atomic (single values)
- A primary key is assigned.

While the XML data model does not require atomic values —child elements can occur multiple times— the first part of this form does suggest that there should only be one (repeating) child element type for a given type of data. For example, instead of using multiple, similarly named child elements like the following:

```
<!ELEMENT Book (Title, Author1, Author2, Author3, ..., Content)>
```

use a single child element that can occur multiple times, as this does not artificially restrict the number of values:

```
<!ELEMENT Book (Title, Author+, Content)>
```

The second part of first normal form —assigning a primary key— does apply to XML schemas. A primary key uniquely identifies a row and implies that the data in a row represents a single, indivisible set of data. (This is not necessarily true for tables in first normal form —later forms may remove some columns— but it is true for tables in third normal form.)

While a primary key cannot always be assigned to an XML document —for example, there might not be any element or attribute value in an XHTML document that uniquely identifies it— the idea that an XML document should represent a single, indivisible set of data does makes sense. For example, an XML document should contain a single invoice or Web page, rather than a set of invoices or Web pages.

Documents that violate this form are easily identified because the root element type exists only to meet the requirement that an XML document have a single root element. That is, the root element type has no attributes and only has repeating child elements or child elements that are wrapper elements surrounding repeating child elements, as shown in Example 3-10 on page 82.

Example 3-10 Root element with no attributes

```
   <!ELEMENT SalesOrders (SalesOrder+)>
or:
   <!ELEMENT Database (SalesOrders, Items, Parts, Customers)>
   <!ELEMENT SalesOrders (SalesOrder*)>
   <!ELEMENT Items (Item*)>
   <!ELEMENT Parts (Part*)>
   <!ELEMENT Customers (Customer*)>
```

Thus, to conform to first normal form, the document should contain a single set of data.

Note that documents that do not conform to first normal form are still useful in some cases. For example, when replicating a database, it is useful to send multiple sets of data in a single document, since this allows the receiving database to optimize processing by using bulk loading.

Second normal form

Second normal form splits a table into two tables when there is a one-to-many relationship between some columns, such as between sales order header information and sales order items. This form does not apply to the XML data model, as a parent and child can have a one-to-many relationship.

As a matter of good design, you should wrap the "many" children in a wrapper element, rather than using repeating subgroups. This makes the intent of the schema clearer and may make processing easier. (DB2 XML Extender and XML Wrapper both require this to handle repeating children.) For example, use:

```
<!ELEMENT SalesOrder (Number, OrderDate, Customer, Item+)>
<!ELEMENT Item (Number, Part, Quantity)>
```

instead of:

```
<!ELEMENT SalesOrder (Number, OrderDate, Customer,
                      (ItemNumber, Part, Quantity)+)>
```

Third normal form

Third normal form splits a table into two tables when there is a many-to-one relationship between some columns, such as between sales order header information and customer information in a sales order. While this form does apply to XML schemas, there is no clear best choice here due to the inability of the XML data model to easily handle many-to-one relationships.

There are three possible ways to handle many-to-one relationships:

- Ignore the problem and repeat the "many" data in each place it logically appears (Example 3-11).

Example 3-11 Handling many to one relationships - Ignore

```
<SalesOrder Number="123">
      <OrderDate>2003-07-28</OrderDate>
   <Customer Number="456">
      <Name>ITSO Insurance, Inc.</Name>
      <Street>123 Main St.</Street>
      <City>Chicago</City>
      <State>IL</State>
      <PostCode>60609</PostCode>
      <Country>US</Country>
   </Customer>
   <Item Number="1">
         ...
   </Item>
      <Item Number="2">
      ...
      </Item>
</SalesOrder>
```

- Place the "many" data in a separate part of the document and reference it as needed, such as with ID/IDREF attributes (Example 3-12).

Example 3-12 Handling many to one relationships - ID/IDREF

```
<SalesOrderDoc>
      <SalesOrder Number="123">
      <OrderDate>2003-07-28</OrderDate>
      <Customer IDREF="A">
      <Item Number="1">
            ...
      </Item>
         <Item Number="2">
         ...
         </Item>
      </SalesOrder>
      <Customer ID="A" Number="456">
         <Name>ITSO Insurance, Inc.</Name>
         <Street>123 Main St.</Street>
         <City>Chicago</City>
         <State>IL</State>
         <PostCode>60609</PostCode>
         <Country>US</Country>
      </Customer>
</SalesOrderDoc>
```

- Place the "many" data in a separate document and reference it through an XLink or external entity reference (Example 3-13).

Example 3-13 Handling many to one relationships - Xlink

```
<SalesOrder Number="123">
   <OrderDate>2003-07-28</OrderDate>
   <Customer xmlns:xlink="http://www.w3.org/1999/xlink"
                xlink:href="customer456.xml">
   <Item Number="1">
         ...
   </Item>
   <Item Number="2">
      ...
   </Item>
</SalesOrder>

<Customer Number="456">
   <Name>ITSO Insurance, Inc.</Name>
   <Street>123 Main St.</Street>
   <City>Chicago</City>
   <State>IL</State>
   <PostCode>60609</PostCode>
   <Country>US</Country>
</Customer>
```

Which choice to use depends on your situation. The first and second choices are equivalent, as they keep all of the data in a single document. The main difference is that the first choice is easier for humans to read, while the second choice will never have any repeated data. These choices are commonly used in two situations: query results and business documents.

Business documents present a special case, as these are often stored intact for historical or legal reasons. Such storage is usually secondary. That is, the document is stored intact, but its data is also extracted and stored separately, such as in a set of relational tables. Because the document storage is secondary, duplicate data is not an issue, as the documents are generally read-only.

The third choice is most commonly used when the XML documents are used for primary storage of the data, such as in a native XML database. The advantage of this choice is that no data is repeated across documents. For example, a medical database might place information about patients in one set of documents, information about insurance companies in another set of documents, and information about doctors in a third set of documents, with cross-references between documents as needed.

> **Note:** If you are shredding XML documents with DB2 XML Extender, then you cannot use the first choice. This is because the XML Extender requires the primary key in parent / child relationships to be in the table corresponding to the parent element type.

Furthermore, the second choice may cause problems because the XML Extender does not have update-or-insert or soft insert semantics. That is, it can only insert data into the database. It cannot check if that data is already in the database. Because of this, if the "many" data is already in the database, an insert error will occur when the XML Extender tries to re-insert the data.

The solution to this problem is to use XSLT to transform incoming documents into many documents (the third choice). One document (inserted first) will contain the "one" data. The other documents will each contain a single row of "many" data. The data from these can be inserted one document (row) at a time and any insert errors are trapped separately.

3.6.5 XML schema styles to avoid

XML is extremely flexible, making it possible to define almost any data structure. Some of those that are best avoided are listed in the sections that follow.

Generic schemas

It is possible to define an XML schema that is 100% generic. That is, an XML schema that can store any data that can be stored in any other XML schema. For example:

```
<!ELEMENT Structure (Structure | Property)+>
<!ATTLIST Structure Name CDATA #REQUIRED>
<!ELEMENT Property (#PCDATA)>
<!ATTLIST Property Name CDATA #REQUIRED>
```

Such a schema is a bad idea because:

- It subverts the idea of using element and attribute names as metadata — that is, to label data.
- It is difficult for humans to read, making it difficult to debug.
- The code to process it is extremely generic, making it unnecessarily complex and error prone.

Furthermore, neither XML Extender, nor XML Wrapper can process this in the expected way. That is, neither product will map table and column names to structure and property names. Instead, both products can only use this kind of XML schema with a similarly generic database schema.

Schemas with role attributes

A role attribute is an attribute whose value affects how another element type or attribute is interpreted. That is, the value of the attribute effectively casts another element type or attribute. For example, suppose you have an XML schema for bibliographic information.

```
<!ELEMENT Entry (Title, Author+, Publisher, Identifier)>
<!ATTLIST Entry
          Type "Book | Article | Paper">
```

In this schema, all entries use an Entry element type, but the meaning of the Identifier element type depends on the value of the Type attribute. For example, if Type is "Book", then Identifier is an ISBN. If Type is "Article" or "Paper", then Identifier is the name of the journal containing the article or paper.

This kind of schema is confusing for the same reasons that generic schemas are. And like generic schemas, most tools designed to work with data-centric XML will not be able to make decisions based on the value of the Type attribute. For example, XML data binding products cannot instantiate different classes based on the value of the Type attribute, nor can DB2 XML Extender store data in different tables or columns based on its value.

That said, attributes that simply provide status information and do not change the way that other elements or attributes are interpreted are not a problem. For example, entries in a membership list might have entries that indicate if the member is active or inactive.

Schemas that encode data as order

Encoding data as order means that the order in which child elements appear is interpreted as a data value. For example, in the sales order document of Example 3-14, the item number is calculated from the position of the Item element.

Example 3-14 Schema that encodes data as order

```
<SalesOrder Number="123">
   <OrderDate>2003-07-28</OrderDate>
   <CustomerNumber>456</CustomerNumber>
   <Item>
      ...
   </Item>
   <Item>
         ...
   </Item>
</SalesOrder>
```

The problem with this is that many tools for processing XML documents cannot perform this calculation. This is particularly true for tools designed to process data-centric XML documents, as such tools often ignore the order in which child elements appear. For example, if you are using DB2 XML Extender or the XML Wrapper to shred an XML document, neither can calculate the item number from the position and store this value in a column.

The prohibition on encoding data as order is not absolute. For example, suppose an XML document represents a chapter in a book (Example 3-15).

Example 3-15 XML document for a book chapter

```
<Chapter>
   <Title>Using XSLT</Title>
   <Section>
         ...
   </Section>
   <Section>
      ...
   </Section>
   <Section>
         ...
   </Section>
</Chapter>
```

Explicitly assigning numbers to each section, such as with a Number attribute on the Section element, makes editing the document more difficult, as each time you insert a new section you must renumber all subsequent sections. In this case, it makes more sense to leave sections unnumbered and let the publishing software calculate section numbers. For example, this could be done by an XSLT stylesheet that published the chapter as XHTML.

Schemas that contain actions

It is possible to write XML schemas in which some elements represent actions to be taken. The most well-known of these is SOAP, where the elements in the envelope (outer) part of the document describe how the data elements in the body (inner) part of the document are to be processed.

While there is nothing wrong with such schemas, they all have one thing in common; they are specific to a particular product or specification. (In particular, neither DB2 XML Extender, nor the XML Wrapper will be able to interpret these actions.) As a result, such schemas are not a good way to represent data that is not tied to a specific processing model.

A better solution is to develop a schema that represents only your data. This can be used by any consumer and, should they need that data in a form that can be interpreted by a specific product, they can transform it as necessary.

Schemas that use element types for data types

It is possible to use element types to indicate data types. For example, in the following schema, the Money element type is used to tell the application that a particular element type is a monetary value:

```
<!ELEMENT Money (#PCDATA)>
<!ELEMENT FullPrice (Money)>
<!ELEMENT SalePrice (Money)>
```

While this might have made sense when DTDs were the only way to define XML schemas, it is no longer necessary. A better solution is to assign simple or complex types with XML Schemas.

3.6.6 Unsupported XML schema structures by DB2 products

There are a number of structures that are not supported by DB2 XML Extender. Some of these are also not supported by the XML Wrapper. Just because these structures are not supported does not mean you should not use them; in fact, many of them are reasonable choices for use in a global XML schema. All it means is that your local (database-specific) XML schema cannot use them.

The unsupported structures are:

- Duplicate names. When shredding XML documents, DB2 XML Extender does not allow any two leaf nodes (attributes or simple element types) to have the same name. This restriction has been removed by DB2 V8.1 FixPak 3 or DB2 V7.2 Fixpak 11.
- Recursive element types. These are element types that have themselves as children. For example, it is possible to represent a family tree with the following element type, where a child of a Person element literally represents that person's child:

  ```
  <!ELEMENT Person (Name, Birthday, Sex, Person*)>
  ```

  While XML Extender and XML Wrapper can handle these structures, they cannot do so automatically. That is, it is not sufficient to map the Person element a single time — it must be mapped each time it occurs. As a result, a different map (DAD document or set of nicknames) is needed for each possible document depth.

- Primary key in table of child element type. When shredding XML documents, DB2 XML Extender requires the primary key in a primary key/foreign key relationship to be in the table corresponding to the parent element type. For example, suppose the primary key in the relationship between the customer and sales order tables is in the customer table. DB2 XML Extender can shred the document in Example 3-16.

Example 3-16 Primary key in parent element

```
<Customer>

   <!-- The following element types map to
           columns in the customer table. -->

   <Number>456</Number>
   <Street>123 Main St.</Street>
   <City>Chicago</City>
   <State>IL</State>
   <PostCode>60609</PostCode>
   <Country>US</Country>

   <!-- SalesOrder maps to the sales order table. -->

   <SalesOrder Number="123">
         <OrderDate>2003-7-28</OrderDate>
   </SalesOrder>
   <SalesOrder Number="124">
      <OrderDate>2003-7-30</OrderDate>
   </SalesOrder>
   <SalesOrder Number="125">
         <OrderDate>2003-8-1</OrderDate>
   </SalesOrder>
</Customer>
```

but cannot shred the document in Example 3-17.

Example 3-17 Primary key in child element

```
<SalesOrder Number="123">

   <!-- OrderDate and the Number attribute map
           to columns in the sales order table. -->

   <OrderDate>2003-7-28</OrderDate>
   <!-- Customer maps to the customer table. -->

   <Customer>
      <Number>456</Number>
      <Street>123 Main St.</Street>
      <City>Chicago</City>
      <State>IL</State>
      <PostCode>60609</PostCode>
      <Country>US</Country>
   </Customer>
</SalesOrder>
```

There are two ways to solve this problem with the latter document.
First, it can be transformed into two separate documents —one that contains customer information and one that contains sales order information, including the customer number.
Second, you can use the XML Wrapper to shred the document. (For more information, see 2.4.5, “Shredding an XML document using the XML Wrapper” on page 47).

- Missing primary keys. When shredding XML documents, DB2 XML Extender requires the document to contain a primary key for any element type that has a child element type that is mapped to a table. In particular, it cannot use keys generated by the database.
- Multi-valued attributes and elements. A multi-valued attribute or element is one that contains a number of values separated by spaces. For example:

  ```
  <WinningNumbers>1 3 7 12 36 38</WinningNumbers>
  ```

  Both the XML Extender and the XML Wrapper treat such values as a single string.
- More than 10240 rows. The XML Extender cannot insert more than 10240 rows into any table when shredding an XML document.

Part 2

Processing XML documents

4

Storing whole XML documents

This chapter introduces one of the storage and access methods provided by XML Extender, namely the XML column. The following topics are covered:

- Storing XML data outside of DB2
- Storing XML documents in DB2 without DB2 XML Extender
- Storing XML documents in DB2 using DB2 XML Extender
- Considerations when using XML columns to store XML documents

4.1 Storing outside of DB2 in the file system

An XML document is just a text file. You have many options as to how and where to store an XML document. In this section, we briefly discuss how you can store an XML document in the file system (local or remote) and how you can reference those XML documents kept in the file system. It is only mentioned here for completeness since there are many disadvantages to this approach.

4.1.1 File system storage considerations

If you have a very simple set of XML documents, the easiest way to store them is in the file system. You can use editing tools provided by the operating system, or any applications to query and/or modify those XML documents.

Storing data outside of DB2 is fairly easy, but has some disadvantages:

- The integrity of the data in your XML document is not guaranteed. No log is created when modifications are made to the XML documents.
- You cannot rely on DB2 to manage concurrent accesses (locking) to your XML documents.
- You cannot benefit from utilities provided by DB2 UDB for Linux, UNIX, and Windows, such as BACKUP or RESTORE (the DB2 for z/OS equivalent utilities are COPY and RECOVER). You need to take action at the operating system level to ensure that proper backups of the data/XML documents are taken.
- You need to consider the security requirements of your files. A DBMS usually provides more granular security control mechanisms than files that are stored in the file system

For a more detailed discussion on XML and databases, see 1.1, “Why use XML with a database?” on page 4.

4.1.2 Accessing data stored in the file system

Accessing XML documents stored in a file system can be done using:

- Home grown applications
- DB2 Information Integrator XML wrapper

Home grown applications

To access XML data stored in the file system, you can write your own applications that retrieve the XML document, parse it (for example using SAX or DOM) and process the document. Such an application can combine information from the XML document with other information stored in a database (Figure 4-1 on page 95). Because this book focuses on XML and databases, we will not explore this option any further. In our opinion, you should only revert to this approach if everything else (that follows in this publication) fails.

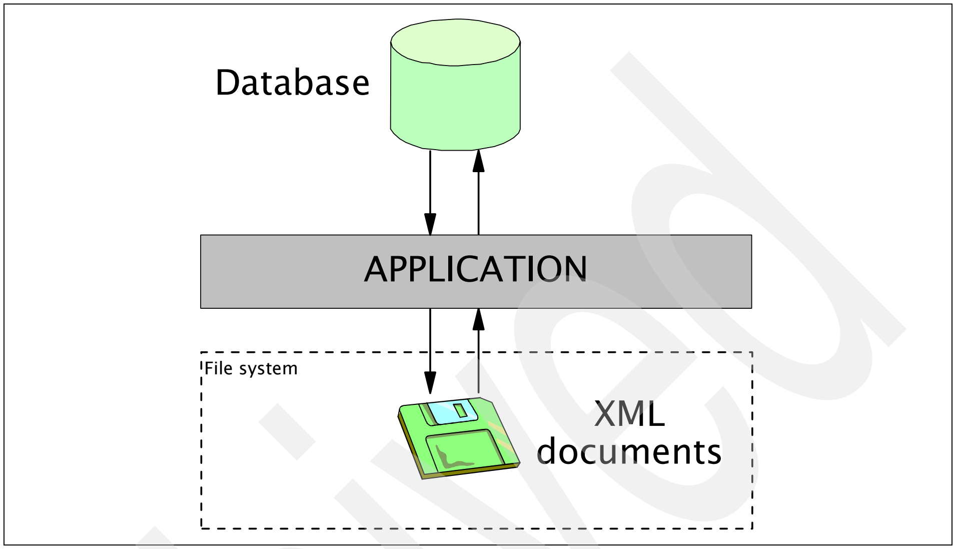


Figure 4-1 XLM documents stored outside of DB2

Accessing XML documents via the XML wrapper

While the base DB2 for Linux, Unix and Windows product only provides wrappers to access members of the DB2 Family (through DRDA®) and Informix data sources, the DB2 Information Integrator (II) product provides additional relational and non-relational wrappers enabling access to all sorts of non-DB2 data sources, such as Oracle databases, Excel spreadsheets, as well as XML documents.

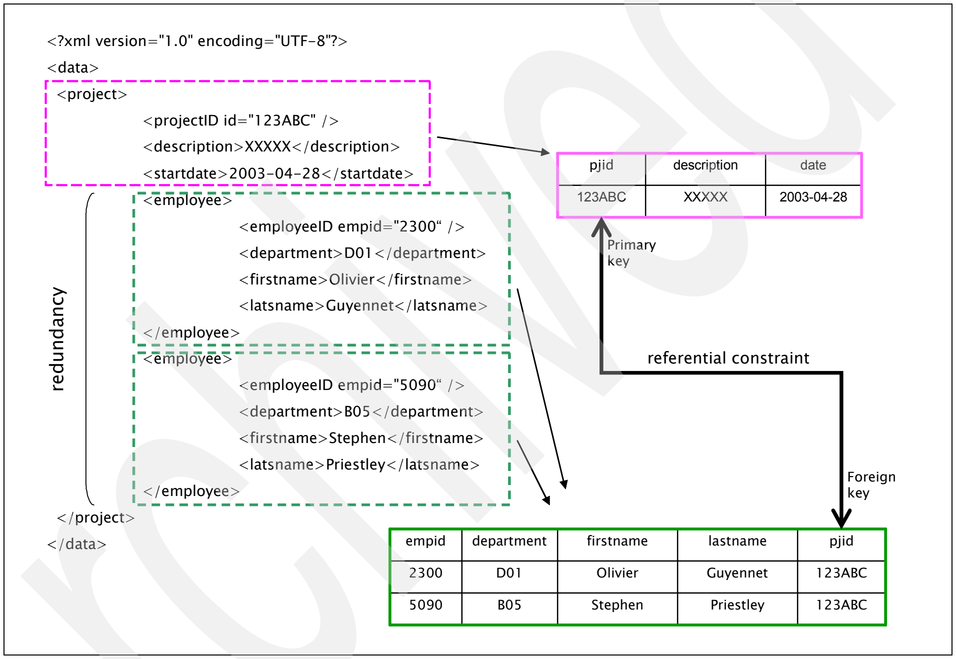
The DB2 II product allows you to create a so-called XML wrapper to access XML documents. For each XML document (or set of documents adhering to the same DTD), you need to create a nickname (or a number of nicknames). The nickname allows you to present the XML document as a table (or a set of tables). Users can then access these nicknames (and the underlying XML document(s)) as if they were (a) DB2 table(s), for example by querying them using a standard SQL SELECT statement.

In this section, we only provide a brief introduction to the XML wrapper. More details can be found in Chapter 11, “XML wrapper” on page 305.

The creation of a nickname with the XML wrapper involves the following tasks:

- Appropriate setup of the FEDERATED parameter in the database manager configuration file (should be set to `YES`)
- Generation of the XML *wrapper*
- Definition of the *server* for the XML wrapper
- Creation of the *nickname* itself, by defining the mapping between the XML elements or attributes and the columns in relational tables

Figure 4-2 shows a possible mapping between the XML document on the left and the relational tables (nicknames) on the right.



```
<?xml version="1.0" encoding="UTF-8"?>
<data>
 <project>
          <projectID id="123ABC" />
          <description>XXXXX</description>
          <startdate>2003-04-28</startdate>
          <employee>
                    <employeeID empid="2300“ />
                    <department>D01</department>
                    <firstname>Olivier</firstname>
                    <latsname>Guyennet</latsname>
          </employee>
          <employee>
                    <employeeID empid="5090“ />
                    <department>B05</department>
                    <firstname>Stephen</firstname>
                    <latsname>Priestley</latsname>
          </employee>
 </project>
</data>
```

pjid	description	date
123ABC	XXXXX	2003-04-28

empid	department	firstname	lastname	pjid
2300	D01	Olivier	Guyennet	123ABC
5090	B05	Stephen	Priestley	123ABC

Figure 4-2 Mapping of an XML document to relational tables

4.2 Storing XML in DB2 without using XML columns

In this section, we briefly discuss the capability that DB2 has to store XML documents, though without using the functionality offered by DB2 XML Extender, that is, without using the user-defined types (UDT) provided by DB2 XML Extender.

However, not using the DB2 XML Extender UDTs does not mean that you cannot use the DB2 XML Extender stored procedures and user-defined functions (UDF) that are created when the database is XML-enabled.

4.2.1 Using DB2 data types without XML Extender functions

Because an XML document is nothing more than a text file, it is possible to store its content using the following DB2 data types (the list below is not exhaustive).

- VARCHAR (up to 32,672 byte, 32,704 on DB2 for z/OS)
- VARGRAPHIC (up to 16,336 double-byte characters long, 16,352 on z/OS)
- CLOB (up to 2Gb -1 byte)

You can use a regular SQL INSERT statement to store the document in a column of your DB2 table.

Example 4-1 shows a DDL statement to create a simple table, and shows how to insert two records (XML documents) into it using an SQL INSERT statement.

Example 4-1 An INSERT statement to store an XML document

```
CREATE TABLE my_tab_clob
(
   id INTEGER NOT NULL,
   myxmldata CLOB NOT LOGGED,
   PRIMARY KEY (ID)
)

INSERT INTO my_tab_clob (id, myxmldata) VALUES
(1,‘<root><text>Hello World</text><type>information</type></root>‘),
(2,‘<root><text>Network will go down</text><type>alert</text></root>‘)
```

Administering whole XML documents in DB2 shows ease in management and more security granularity, since it is easy to protect XML documents at the document (DB2 row) level. However, relying on just DB2 capabilities to store XML data still has weaknesses, such as:

- Identification of the XML document itself is difficult just by looking at the catalog metadata.
- The validity of the XML documents is not checked by DB2.
- No indexing is possible on actual data stored within the XML document.

Even with the disadvantages described above, there might be a case to use normal DB2 data types to store XML documents.

When you enable your DB2 database for XML, a number of user-defined data types (UDT) provided by the XML Extender are created. These are:

- XMLVARCHAR
- XMLCLOB
- XMLDBCLOB
- XMLFILE

When you define a UDT (or in this case when XML Extender defines the UDT), you must specify a length for it. This means that, for example, all XMLVARCHAR columns in all tables that use that UDT have the same (maximum) length. For DB2 UDB, this is 3K. For this reason, you may want to avoid these fixed length restrictions of XML Extender XMLCLOB and XMLVARCHAR data types. The XMLDBCLOB data type is only created when the database is enabled for DBCS.

> **Note:** If 3K is not enough, you can pre-create the XMLVARCHAR, XMLCLOB, and XMLDBCLOB UDTs with the desired size prior to enabling the database.

The good thing about XML Extender is that a lot of the functionality is also available to you when the column that harbors the XML document is not defined as an XML Extender UDT, or is not an XML Extender enabled column. The section below shows how you can still use XML Extender user-defined functions (UDFs) with columns that are not XML Extender enabled or columns that are not defined as an XML Extender UDT. The database, however, has to be enabled for XML to be able to use these UDFs.

4.2.2 Using XML Extender UDFs to insert XML documents into a normal DB2 column

DB2 XML Extender provides four types of functions for storing, searching, updating XML documents, and for extracting XML elements or attributes. You use XML user-defined functions (UDFs) to perform those operations.

> **Note:** XML UDFs can be used for XML columns or non-XML columns, but cannot be used with XML collections. XML columns are discussed in more detail in 4.3, “DB2 XML Extender storage methods” on page 100 and 4.4, “Storing intact XML documents with XML Extender” on page 101.

In this section, we will only deal with the storage UDFs that allow you to insert XML documents into “regular” DB2 relational tables.

One of the first steps when using the DB2 XML Extender is to enable the database for XML. This is done via the administration command **`dxxadm`**. At the time of enablement, XML Extender UDTs, UDFs and stored procedures are created.

The storage related UDFs created by the XML Extender are shown in Table 4-1.

Table 4-1 XML Extender storage UDFs

Storage UDF	Return type	Description
XMLVarcharFromFile()	XMLVARCHAR	Reads an XML document from a file, and returns the document as an XMLVARCHAR type
XMLCLOBFromFile()	XMLCLOB	Reads an XML document from a file, and returns the document as an XMLCLOB type
XMLFileFromVarchar()	XMLFILE	Reads an XML document from memory as VARCHAR, writes it to an external file, and returns the file name and path as an XMLFILE type
XMLFileFromCLOB()	XMLFILE	Reads an XML document from memory as CLOB locator, writes it to an external file, and returns the file name and path as an XMLFILE type

To use the UDF for storing XML documents, you just have to make sure that there is no problem with data type compatibility. For example, you cannot use **`XMLVARCHARFromFile`** to store data into a CLOB. However, the database itself has to be XML-enabled; otherwise, the storage UDFs have not been created.

Example 4-2 shows how to insert an XML document called data.xml in the `myxmldata` column of a DB2 table my_tab. We use the XLM Extender storage UDF `XMLVarcharFromFile()`.

Example 4-2 An INSERT using the XMLVarcharFromFile() UDF

```
CREATE TABLE my_tab
(
   id INTEGER NOT NULL,
   myxmldata VARCHAR(200),
   PRIMARY KEY (id)
)

INSERT INTO my_tab (id, myxmldata) VALUES
(123, db2xml.XMLVarcharFromFile(‘C:\xmldata\data.xml’))
```

Figure 4-3 illustrates the storage of an entire XML document in a DB2 relational table column. Note that the type of the column can either be a DB2 built-in data type, or an XML Extender data type (that is, XMLVARCHAR, XMLCLOB).

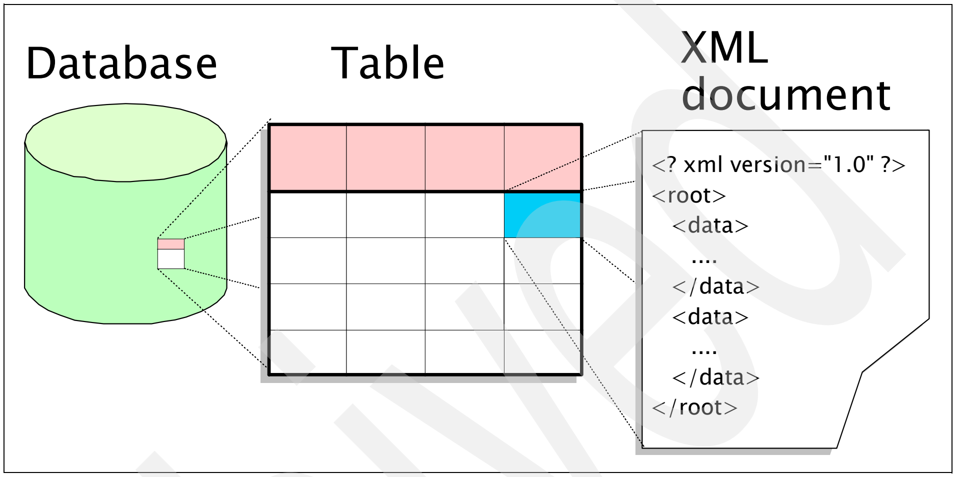


Figure 4-3 An XML document stored intact

4.3 DB2 XML Extender storage methods

DB2 XML Extender provides two storage methods to use DB2 as an XML repository:

- XML column
- XML collection

Deciding which of these methods best matches your needs for accessing and manipulating XML data is an important first step. Both methods are described below.

XML column

This method allows you to store an entire XML document as it is, in DB2. Documents are inserted into columns enabled for XML, and can then be updated, retrieved and searched. Elements and attributes data can be mapped to so-called side tables, which can be indexed for fast searches.

XML collection

Using this method allows you to map XML document structures to DB2 tables, so that you can decompose XML documents into DB2 tables, and compose XML documents from the existing DB2 data.

The nature of your application determines which access and storage method is most suitable, as well as how to structure your XML data.

In this chapter, we have seen, so far, how to store an XML document outside DB2, or in DB2 using only DB2 functionality.

The rest of this chapter discusses the usage of DB2 XML Extender to store XML documents intact and "as is" into DB2 relational tables.
Chapter 6, "Shredding XML into relational tables" on page 143 shows how to decompose (shred) XML documents into one or more relational tables using DB2 XML Extender.

4.4 Storing intact XML documents with XML Extender

As discussed previously, when you enable a database for XML usage, a number of UDTs, UDFs and stored procedures are created. After the database is XML-enabled, you can enable an XML column. Once the column is enabled, you can start inserting complete XML documents into the XML column, using the UDFs provided by XML Extender.

4.4.1 Using the XML Extender data type

DB2 XML Extender provides XML user-defined types (UDTs) that you can use to define a column to hold XML documents. These UDTs are created when the database is XML-enabled by the administration command:

```
dxxadm enable_db database_name
```

The data types generated by XML Extender are:

XMLVARCHAR The XMLVARCHAR data type is based on DB2's VARCHAR built-in data type. XMLVARCHAR is mostly used to store small documents in DB2. The maximum length of the XMLVARCHAR data type is 3K.

XMLCLOB Stores an entire XML document as a CLOB data type within DB2. XMLCLOB is normally used for large documents stored in DB2. The maximum length of the XMLCLOB data type is 2Gb -1.

XMLDBCLOB Stores an entire XML document as a DBCLOB data type within DB2. XMLDBCLOB is normally used for large documents stored in DB2. The maximum length of the XMLDBCLOB data type is 2Gb -1. XMLDBCLOB only applies when the database is enabled for double byte character set data.

XMLFILE Stores the file name of an XML document in DB2, and keeps the XML document itself in the file system, local to the DB2 server. This data type can be used for XML documents stored outside DB2. The maximum length of a XMLFILE data type (to represent the external file name that harbors the XML document) is 512 bytes.

4.4.2 When to use an XML column to store data

You may use XML columns in the following situations:

- The XML documents already exist or come from an external source and you prefer to store the documents in the native XML format. You want to store them in DB2 for integrity, archival, and auditing purposes.
- The XML documents are read frequently, but rarely or not updated.
- You want to use file name data types to store the XML documents external to DB2 in the local or remote file system and use DB2 for management and search operations.
- The documents have elements with large text blocks, and you want to use DB2 Net Search Extender for structural text searches while keeping the entire documents intact.

Keep in mind also that white space (blanks and blank sections) is important. By storing the XML document intact, you can preserve the white space.

You can use so-called “side tables” to perform range searches based on the values of XML elements or attributes, if you know what elements or attributes will frequently be the search arguments. In addition, you can build normal DB2 indexes on those side tables to speed up access even more.

When you want to store data in an XML colum, you need to do the following things:

1. Enable the database for XML using the `dxxadm enable_db` *`database_name`* command.
2. Build a DAD file. In the case of storing XML documents in an XML column, the DAD file specifies whether or not validation of the XML document needs to be performed, as well as which elements and attributes to use to build side tables.

3. Create the table in which you want to store the XML documents.
4. If the DAD file specifies that validation is required, insert the DTD into the DTD_REF table.
5. Enable the XML column.
6. Create indexes on the side tables to provide faster access.
7. Insert data into the XML column.

4.4.3 Building the DAD file

The intent of this section is not to cover all the details about creating a DAD file, but rather to try to help you understand what you need to do to build the most appropriate DAD file.

When you enable an XML column, you have to specify a Document Access Definition (DAD) file. The DAD file is used for many different things, but when used with an XML column, it specifies whether or not validation is required, as well as what side tables and side table columns to create. The DAD file is an XML document. For a detailed description of what you can code in a DAD and what is allowed according to the DAD's DTD, see Appendix D, "DAD DTD reference" on page 599.

As an example, we use the following project XML document (Figure 4-3).

Example 4-3 Project.xml document

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE data SYSTEM "C:\Documents and
Settings\resident\Desktop\chap3\project.dtd">
<data>
   <project>
      <projectID id="123ABC"/>
      <description>XXXXX</description>
      <startdate>2003-04-28</startdate>
      <employee>
         <employeeID empid="2300"/>
         <department>D01</department>
         <firstname>Olivier</firstname>
         <lastname>Guyennet</lastname>
      </employee>
      <employee>
         <employeeID empid="5090"/>
         <department>B05</department>
         <firstname>Stephen</firstname>
         <lastname>Priestley</lastname>
      </employee>
   </project>
</data>
```

The DAD file shown in Example 4-4 can be used to store the XML document in an XML column, and populate the appropriate side tables for easy access.

Example 4-4 An example DAD file for the Project XML document

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "C:\DB2\SQLLIB\samples\db2xml\dtd\dad.dtd">
<DAD>
   <dtdid>c:\xmldata\project.dtd</dtdid>
   <validation>YES</validation>
   <Xcolumn>
      <table name="XProject">
         <column name="pjid" type="char(6)"
                              path="/data/project/projectID/@id"
                              multi_occurrence="NO"/>
         <column name="description" type="varchar(40)"
                                    path="/data/project/description"
                                    multi_occurrence="NO"/>
         <column name="date" type="DATE"
                              path="/data/project/startdate"
                              multi_occurrence="NO"/>
      </table>
      <table name="XEmployee_ID">
         <column name="empid" type="char(4)"
                           path="/data/project/employee/employeeID/@empid"
                           multi_occurrence="YES"/>
      </table>
      <table name="XEmployee_Name">
         <column name="lastname" type="varchar(20)"
                                 path="/data/project/employee/lastname"
                                 multi_occurrence="YES"/>
      </table>
   </Xcolumn>
</DAD>
```

Note: The structure of a DAD file varies depending on whether you are using XML columns or XML collections, and/or whether you are shredding or composing XML documents.

For XML columns, the DAD file specifies a Document Type Description (DTD) file (`<dtdid>c:\data\project.dtd</dtdid>`) to use for automatic validation

(`<validation>YES</validation>`) of the documents inserted into the XML column. It also specifies how documents that are stored in XML columns are to be ”indexed.” To do so, the XML Extender creates so-called *side tables*.

Side tables are standard DB2 tables that are used to store parts of an XML document (elements and attributes) that will be searched frequently. You can use the side tables as an “index” to look up an XML document based on the value of certain attributes and elements stored in the side tables. When the XML document that is stored in the XML column is updated, the values in the side tables are automatically updated as well.

Before you create the DAD file, you need to:

- Decide which elements or attributes you expect to use often in your searches. The elements or attributes that you specify are extracted from the XML document at insert time and stored into the side tables for fast searches by the XML Extender (or SQL statements that your write yourself).
- Define the location path for each element or attribute that is to be stored in a column of a side table. To specify the location path, DB2 XML Extender uses the XPATH notation. For information about the subset of XPath that is supported in XML Extender location path expressions, see “Working with an XML Extender location path” in the *XML Extender Administration and Programming* manual.
- Specify the DB2 column name and data type that you want the element or attribute to have when it is stored in the side table by DB2 XML Extender.
- An important planning decision when using XML columns, is whether to index the side tables for XML column documents. This decision should be made based on how often you need to access the data, how big the side tables are, and how critical performance is during structural searches.

You must keep the following considerations in mind when creating a side table:

- You can create multiple side tables on a single XML column.
- For each element or attribute in an XML document that can occur multiple times, you must create a separate side table, and specify the `multi_occurrence="YES"` keyword in the DAD file (in Example 4-4 on page 104, this is the case for the XEmployee_ID and XEmployee_Name tables). This is due to the complex structure of XML documents.
- You can associate side tables with the table storing the XML document in the XML column:
  - Use the primary key in the table that contains the XML column. When you enable the XML column, you can specify the primary key as the ROOT ID via the -r option. This way, the column that you specified as the ROOT ID is also created (and populated) for each of the side tables. This method is

recommended. However, if you decide to use the primary key of the application table containing the XML column to be the ROOT ID, it cannot be a composite key.

- If the single primary key does not exist in the application table, or for some reason you don't want to use it, you can also have XML Extender create a DXXROOT_ID column for you (in the table containing the XML column, as well as all side tables). XML Extender will add a column, DXXROOT_ID, to the table containing the XML column. This column contains a unique ID that is generated at insertion time. All side tables will also have the same DXXROOT_ID column with same value as the row that is inserted into the table containing the XML column.

See also 4.4.6, "Enabling the XML column" on page 108 for additional information about side tables.

4.4.4 Creating the table that will contain the XML column

You can create or add a column to an existing table that will contain the XML document data in an XML column. We create the XML_PROJECT table to store our project XML documents in the PROJ_XML XML column (see Example 4-5 and Figure 4-4).

Example 4-5 Create table to store XML column

```
create table PROJ_XML_TB
      (PROJ_ID integer not null,
       PROJ_XML db2xml.xmlclob not logged,
       primary key (PROJ_ID)
      )
```

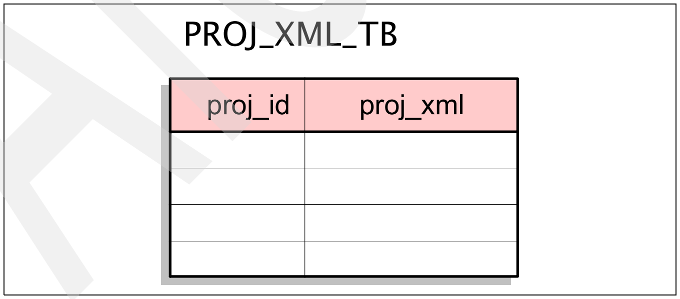


Figure 4-4 The PROJ_XML_TB table

4.4.5 Inserting the DTD into the DTD_REF table

If, in the DAD file, you request validation of the XML documents that you insert into the XML column, you need to “register” the Data Type Definition (DTD) with XML Extender. This is done by inserting the DTD in the DTD_REF table. The DTD_REF table is created when you enable the database for XML. You can use the following INSERT statement (Example 4-6) to do so.

Example 4-6 Insert into DTD_REF

```
INSERT INTO db2xml.DTD_REF
       VALUES(‘c:\xmldata\project.dtd’,
              db2xml.XMLCLOBFromFile(‘c:\xmldata\project.dtd’),
              0,
              ‘bart’,
              ’bart’,
              ‘bart’
             )
```

Example 4-7 shows the project.dtd information that we are inserting into the DTD_REF table that will be used to validate the XML documents when they are inserted into the XML column later on (see 4.4.8, “Using XML Extender UDFs to insert XML documents” on page 110).

Example 4-7 Project.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT data (project)>

   <!ELEMENT project (projectID, description, startdate, employee*)>
      <!ELEMENT projectID EMPTY>
      <!ELEMENT description (#PCDATA)>
      <!ELEMENT startdate (#PCDATA)>
      <!ATTLIST projectID id CDATA #REQUIRED>

   <!ELEMENT employee (employeeID, department, firstname, lastname)>
      <!ELEMENT employeeID EMPTY>
      <!ELEMENT department (#PCDATA)>
      <!ELEMENT firstname (#PCDATA)>
      <!ELEMENT lastname (#PCDATA)>
      <!ATTLIST employeeID empid CDATA #REQUIRED>
```

4.4.6 Enabling the XML column

Since we only have one administration command for DB2 XML Extender, the command to enable an XML column remains **dxxadm**, but this time we use the enable_column option (Example 4-8). The required parameters are:

- The database name
- The name of the table containing the XML column
- The name of the XML column in that table
- The DAD file to be associated with this XML column

There are also some optional arguments:

- Tablespace (-t): The table space where the side tables are created. You specify this option only if you want to put the side tables into a specific table space.
- Default_view (-v): The name of the default view joining the user table containing the XML column and the side tables.
- Root_id (-r): The name of the single primary key in the user table that is added to the side tables. The root_id is the way to tie the side tables together with the user table. We recommend that you use the primary key that is defined on the user table as the root_id. If you do not specify a root_id, or if your application table has no primary key, a DXXROOT_ID column is added to the user table containing the XML column (at enable column time) for storing a unique ID generated when you insert a row into the XML column. All side tables will then have that same column, DXXROOT_ID.

Example 4-8 Enabling an XML column

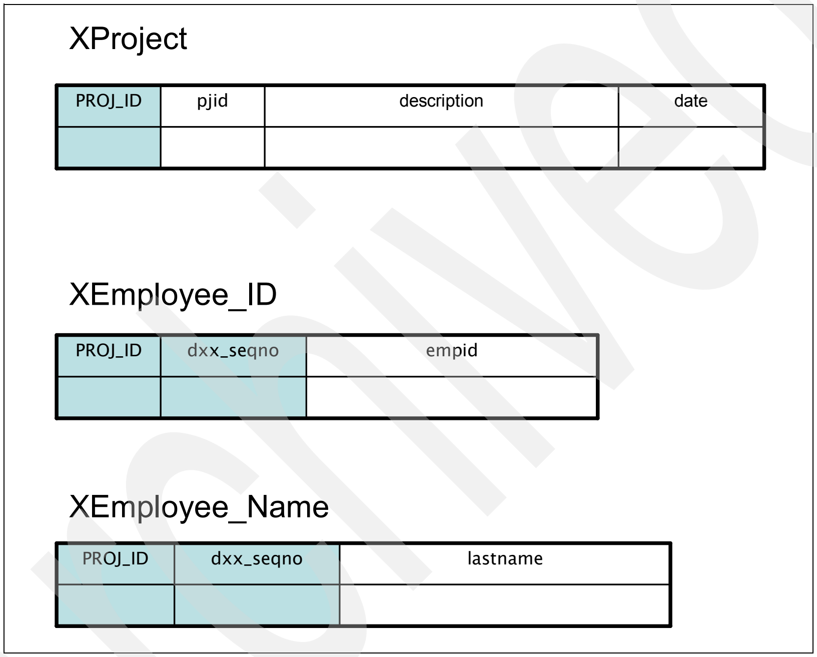
```
dxxadm enable_column PROJ_XML_TB
                       PROJ_XML
                       “c:\xmldata\project.dad”
                       -r PROJ_ID
```

When you enable an XML column, DB2 XML Extender performs the following actions:

- Add a new row to the XML_USAGE table (this table is created when the database is enabled for XML). This new entry keeps information about the relation between the user table, the XML column in that table, the DTDID and the DAD file. The DAD file is stored as a CLOB in the XML_USAGE table.
- Create the side tables with the desired columns corresponding to the elements and attributes from the XML document based on information in the DAD file.

- Create (three) triggers on the user table containing the XML column to maintain the side tables.
- If you decide to validate the XML document, the USAGE_COUNT column in the DTD_REF table is increased for the relevant row. You cannot delete the entry in the DTD_REF table when the USAGE_COUNT for this DTD is not zero (enforced by a trigger on the DTD_REF table).

To illustrate the side table concept (Figure 4-5), we use the Project XML document and its DAD file again. The side tables are created when you enable the XML column.



XProject

PROJ_ID	pjid	description	date

XEmployee_ID

PROJ_ID	dxx_seqno	empid

XEmployee_Name

PROJ_ID	dxx_seqno	lastname

Figure 4-5 Columns created in the side tables

Note that all side tables contain the `PROJ_ID` column (-r option during dxxadmin enable_column). Also note that a `dxx_seqno` column has been added to the Xemployee_ID and Xemployee_Name table. When an attribute or element can occur multiple times in a single XML document (`multi_occurrence="YES"` in the DAD file), a sequence number is automatically provided to identify each occurrence. This allows you to access individual entries of the attributes and elements of a single XML document using the side tables in the same order as they occur in the actual XML document.

Important: DB2 XML Extender populates the side tables at insertion or update time with the triggers that are created when the XML column is enabled. This means that if your column already contains XML documents before being enabled, the content of these XML documents will not be reflected in the side tables. The same is true if you decide to add side tables (to provide additional “fast” search criteria) after the table is populated with XML documents.

4.4.7 Creating indexes on the side tables

After you have enabled an XML column, and the side tables are available, you can create indexes on columns of the side tables using the DB2 `CREATE INDEX` statement. Indexing the side tables helps you improve the performance of the queries against the side tables when retrieving information from the side tables or retrieving XML documents.

4.4.8 Using XML Extender UDFs to insert XML documents

As explained in 4.2.2, “Using XML Extender UDFs to insert XML documents into a normal DB2 column” on page 98, DB2 XML Extender provides functions for storing XML documents in relational tables. Those storage functions are:

- **`XMLVarcharFromFile`**
- **`XMLCLOBFromFile`**
- **`XMLFileFromVarchar`**
- **`XMLFilefromCLOB`**

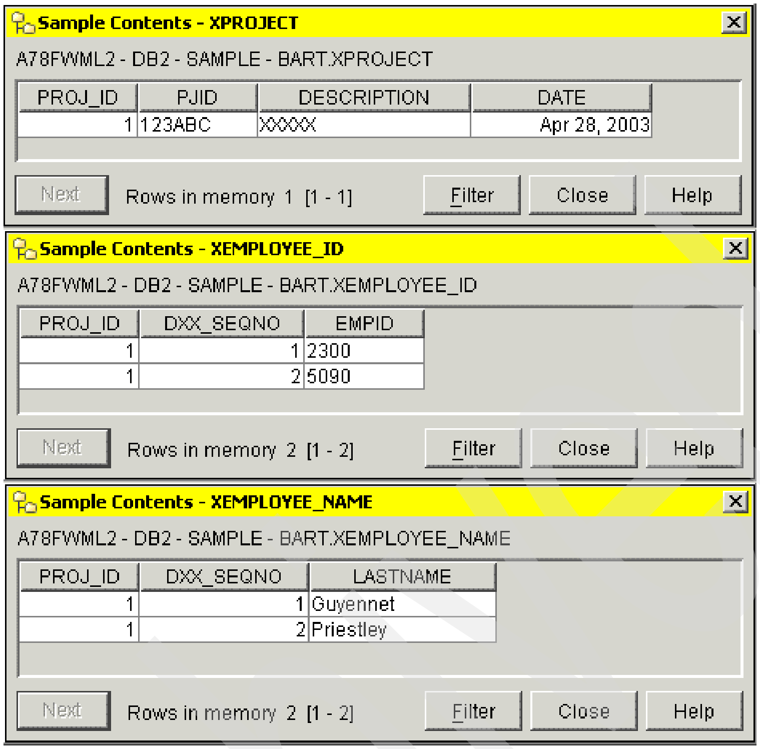
Note: These UDFs are created in the DB2XML schema.

In Example 4-9, we insert an XML document (project.xml) into the PROJ_XML_TB table that contains an XML column (PROJ_XML) defined as XMLCLOB.

Example 4-9 Insert data into the XML column

```
INSERT INTO PROJ_XML_TB
       VALUES (1,db2xml.XMLCLOBFromFile(’c:\xmldata\project.xml’))
```

After inserting the project.xml document into the PROJ_XML column, the XML column should contain the XML document, and the side tables should look like Figure 4-6 on page 111.



Sample Contents - XPROJECT

A78FWML2 - DB2 - SAMPLE - BART.XPROJECT

PROJ_ID	PJID	DESCRIPTION	DATE
1	123ABC	XXXXX	Apr 28, 2003

Rows in memory 1 [1 - 1]

Sample Contents - XEMPLOYEE_ID

A78FWML2 - DB2 - SAMPLE - BART.XEMPLOYEE_ID

PROJ_ID	DXX_SEQNO	EMPID
1	1	2300
1	2	5090

Rows in memory 2 [1 - 2]

Sample Contents - XEMPLOYEE_NAME

A78FWML2 - DB2 - SAMPLE - BART.XEMPLOYEE_NAME

PROJ_ID	DXX_SEQNO	LASTNAME
1	1	Guyennet
1	2	Priestley

Rows in memory 2 [1 - 2]

Figure 4-6 Side table contents

5

Working with XML documents stored in XML columns

This chapter describes how to work with XML documents that have been stored in an XML column:

- We first look at how to retrieve complete XML documents stored in an XML column
- Next, we describe how to extract parts (elements and attributes) of an XML document
- We also describe how to filter which XML documents we want to retrieve
- Then we look at how we can use an XML wrapper to retrieve information that is stored in an XML column
- We describe how to update and delete XML documents
- In conclusion, we give some words of advice on how XML columns are best used

5.1 Retrieving, extracting, updating, and deleting XML columns

In the previous chapter, we focused on *storing* XML documents in XML columns. In this chapter, we show how to perform other operations like *searching, retrieving and updating* XML documents that are stored in XML columns.

DB2 XML Extender provides user-defined functions (UDFs) for these operations.

Note: XML Extender does not use stored procedures when working with XML columns. Operations are done through UDFs.

For a detailed description concerning the syntax and arguments of these functions, refer to *DB2 XML Extender Administration and Programming V8*, SC27-1234.
In addition, we also look at how to use the XML wrapper to access information stored in an XML column.

In the following subsections, we describe these functions and illustrate some of them using simple examples.

5.2 Retrieving XML documents stored in XML columns

There are various ways of retrieving the XML document you stored in the database as an XML column.

First, we focus on retrieving the document itself, either the entire document, or parts of it; that is, we focus on the SELECT clause of the SQL statement.

In the next part of this section, we investigate how to filter the documents we want to retrieve, that is, the WHERE clause of the SQL statement.

5.2.1 Retrieving the entire XML document

When we want to retrieve an entire XML document that is stored in an XML column, we can use normal SQL SELECT statements. We just specify the XML columns that we want to retrieve in the SELECT clause of the SQL statement (as we normally would with any other column). DB2 does an automatic conversion of the XML data type to a DB2 “native” data type, or we can do it explicitly by using one of the default cast functions. These are generated when the user-defined distinct type (that XML Extender uses) is created (at database enablement time).

This does not apply to XML documents stored in an XML column defined using the XMLFILE data type.

In the following section, we use different flavors of the same table, sales_tab.

`sales_tab_varchar`	Orders are stored in an XMLVARCHAR column
`sales_tab_clob`	Orders are store in an XMLCLOB column
`sales_tab_file`	Orders are store in an XMLFILE column

Into host variables

As mentioned above, when using XMLVARCHAR, XMLCLOB and to store your XML documents, they can be easily retrieved by specifying the column in the select list of your SQL statement. (You can also use XMLDBCLOB if your database supports DBCS data.) For example, when storing XML documents in an (XML-enabled) column called `order`, declared as an XMLVARCHAR, we can select documents from it as follows:

```
select order from sales_tab_varchar
```

However, if we need to perform additional operations on the result, for example applying the length function, we must cast the XML data type to a source data type, for example:

```
select length(db2xml.varchar(order)) from sales_tab_varchar
```

When the XML UDTs are defined, DB2 also generates default casting functions for those data types, as shown in Table 5-1.

Table 5-1 Default cast functions

Cast function to specify in the SELECT statement	Result data type	Description
VARCHAR(xmlvarchar)	VARCHAR	XML document stored as XMLVARCHAR is casted to a source VARCHAR
CLOB(xmlclob)	CLOB	XML document residing in XMLCLOB is converted to CLOB
DBCLOB(xmldbclob)	DBCLOB	XML document residing in XMLDBCLOB is converted to DBCLOB
VARCHAR(xmlfile)	VARCHAR	XML filename is converted to a VARCHAR.

Do not forget to qualify your cast function in case DB2XML is not part of your function resolution PATH, as shown in the example above.

When using XMLFILE, however, retrieving the content of the XML document cannot be achieved by just specifying the column name of the XML column. For example, when the order column is defined as XMLFILE, using:

```
select order from sales_tab_file
```

We only retrieve the external file name where the XML document is stored, and not the content itself, for example:

```
C:\XMLdocs\order0001.xml
```

To retrieve the actual content you must use the **`CONTENT()`** UDF. The content function exists in multiple flavors; it is a so-called overloaded function.

Unfortunately we can only directly retrieve the content of an XML document, stored in an XMLFILE column, into an XMLCLOB (CLOB locator).

For example, to retrieve our document above, we can use:

```
select db2xml.content(order)from sales_tab_file
```

If we prefer a varchar, we can cast it to a varchar as follows:

```
select varchar(db2xml.clob(db2xml.content(order))) from sales_tab_file
```

You have to use the db2xml.clob cast function first (to cast between XMLCLOB and a source CLOB) before you can cast the CLOB to a VARCHAR.

Into an external file

You can also retrieve the information of the XML column (XMLVARCHAR or XMLCLOB) and store the result directly into a external file. This is also done by using the content() function.

For example, to retrieve an XML document stored in an XMLCLOB and store it in an external file, you can use:

```
select db2xml.content(order,'c:\exportxml\extractorder.xml')
  from sales_tab_clob
```

The result of the statement is:

```
c:\exportxml\extractorder.xml
```

The result of the content() function is the filename that the XML document was stored in. The XML document that was retrieved from the XMLCLOB column, is stored in the file system under the name extractorder.xml in the c:\exportxml directory.

Important: Make sure that when you retrieve multiple rows (multiple XML documents) your application program assigns different external file names for each row that you retrieve. Otherwise the content is overwritten each time.

To avoid this, you may want to include the key of the table that stores the XML document as part of the external file name. In our sales_tab_file, invoice_num is the key column.

```
select
 db2xml.content(order,'c:\exportxml\extractorder'||'invoice_num'||'.xml')
  from sales_tab_clob
```

When using the DB2 command line processor you may have to include the entire statement in “ (double quotes) in order for it to run properly.

The Content() function works identical when the XML document is stored as an XMLVARCHAR. Unfortunately there is no equivalent function for XMLDBCLOB columns.

To summarize, there are three flavors of the Content() function, depending on the type of parameters:

- The Content() UDF to export an XML document from an XMLVARCHAR format to an external file
- The Content() UDF to export an XML document from an XMLCLOB format to an external file
- The Content() UDF to export an XML document from an XMLFILE format to a CLOB locator

Tip: When using XML columns, the XML document can be retrieved in the exact same form it was originally using before it was stored, including white space. This is normally not the case when using XML collections.

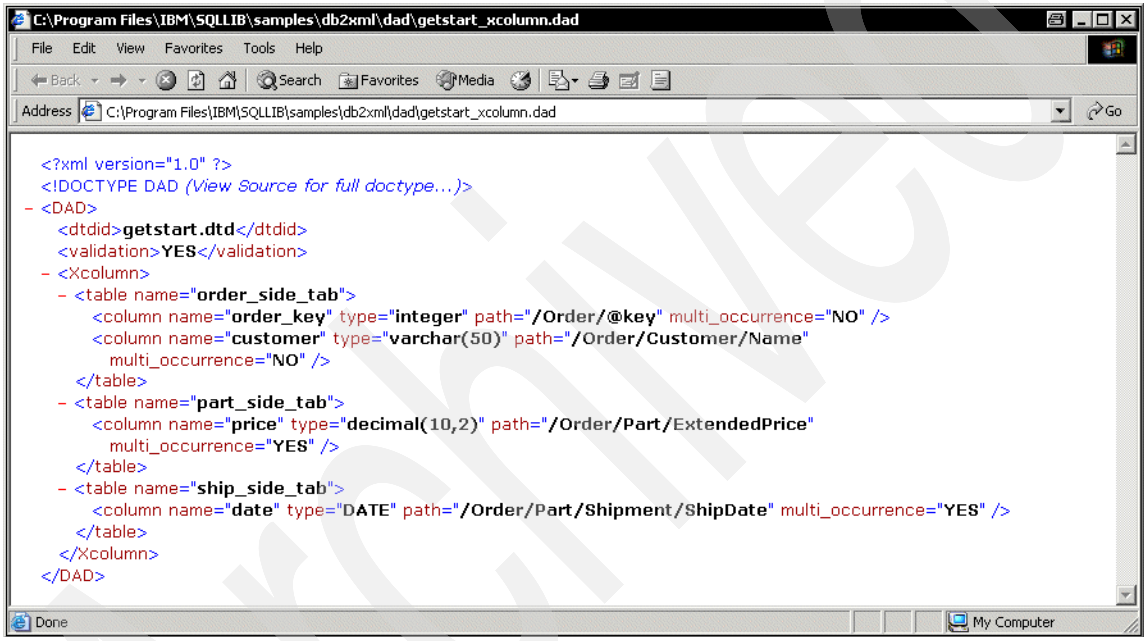
5.2.2 Retrieving elements and attributes from XML documents

Depending on where the information is stored, there are different ways to retrieve elements and attributes of XML documents stored in an XML column.

From side tables

Remember that when you enable the XML column, XML Extender can create side table(s), based on information provided in the DAD file. These side tables contain element and attribute values, extracted from the XML document when it is inserted (stored) into the XML column. If the information you want to retrieve is available in the side table, you can select the information from the side table, instead of using the XML column itself.

Let us look at an example. In our sales_tab_xmlc table, we store XML documents as an XML column in the order column as an XMLCLOB data type. We also created a number of side tables; one of them is the ORDER_SIDE _TAB. This side table contains the invoice number (that is to be used to correlate all information belonging to the same order), and two “fields” extracted from the XML documents, namely order_key and customer. The DAD is shown in Figure 5-1.


C:\Program Files\IBM\SQLLIB\samples\db2xml\dad\getstart_xcolumn.dad
File
Edit
View
Favorites
Tools
Help
Back
Search
Favorites
Media
Address
C:\Program Files\IBM\SQLLIB\samples\db2xml\dad\getstart_xcolumn.dad
Go
Done
My Computer

```
  <?xml version="1.0" ?>
  <!DOCTYPE DAD (View Source for full doctype...)>
- <DAD>
    <dtdid>getstart.dtd</dtdid>
    <validation>YES</validation>
  - <Xcolumn>
    - <table name="order_side_tab">
        <column name="order_key" type="integer" path="/Order/@key" multi_occurrence="NO" />
        <column name="customer" type="varchar(50)" path="/Order/Customer/Name"
          multi_occurrence="NO" />
      </table>
    - <table name="part_side_tab">
        <column name="price" type="decimal(10,2)" path="/Order/Part/ExtendedPrice"
          multi_occurrence="YES" />
      </table>
    - <table name="ship_side_tab">
        <column name="date" type="DATE" path="/Order/Part/Shipment/ShipDate" multi_occurrence="YES" />
      </table>
    </Xcolumn>
  </DAD>
```

Figure 5-1 XML column DAD file

So if we want to retrieve the customer that placed our sales orders, we can use the column in the side table instead of having to use the actual XML document. For example, when using the DB2 CLP (Example 5-1 on page 119):

Example 5-1 Using the side table

```
db2 select customer from order_side_tab

CUSTOMER
--------------------------------------------------
American Motors
European Engines

  2 record(s) selected.
```

Note again that we did not specify a `where` condition in the example, as we are still focusing on retrieving the data itself, not filtering out rows based on selection criteria. We will deal with that later on in this chapter.

From the XML document itself

When the data you want to retrieve is not available in side tables, you can retrieve it from the XML document stored in the XML column itself. To do so, you can use so-called *extraction UDFs*. Extraction UDFs are based on the location path expressions (XPath notation) to locate the desired element or attribute.

The extracting UDFs are divided into these groups:

- Scalar extracting UDFs: These allow you to find an element or attribute within an XML document. This element or attribute must have *only one occurrence* in the whole XML document. These UDFs return a scalar SQL data type.
- Table extracting UDFs: These give the possibility to find a *multiple occurring elements or attributes* within an XML document and return a DB2 table having multiple rows of the considered SQL data type.

Each extracting UDF expects two input parameters:

- The XML document to be searched (XMLFile, XMLVarchar, or XMLCLOB)
- The location path expressed in XPath notation to identify the element or attribute you are looking for

The extracting UDFs convert the value of an element or attribute in the XML document to one of the following SQL data types:

- CHAR
- VARCHAR
- CLOB
- INTEGER
- SMALLINT
- DOUBLE

- REAL
- DATE
- TIME
- TIMESTAMP

Table 5-2 summarizes the existing extracting functions. Refer to *DB2 XML Extender Administration and Programming V8*, SC27-1234 for a more detailed description of each of these functions.

Table 5-2 Extracting UDFs summary

Scalar UDF	Table UDF	Return type	Returned column name (table function)
extractInteger	extractIntegers	INTEGER	returnedInteger
extractSmallint	extractSmallints	SMALLINT	returnedSmallint
extractDouble	extractDoubles	DOUBLE	returnedDouble
extractReal	extractReals	REAL	returnedReal
extractChar	extractChars	CHAR	returnedChar
extractVarchar	extractVarchars	VARCHAR	returnedVarchar
extractCLOB	exrtactCLOBs	CLOB	returnedCLOB
extractDate	extractDates	DATE	returnedDate
extractTime	extractTimes	TIME	returnedTime
extractTimestamp	extractTimestamps	TIMESTAMP	returnedTimestamp

Let us look at a few examples:

Example 1

We want to construct a list of e-mail addresses of our customers. Unfortunately we do not “extract” this information and put it into side tables, when we store the XML document in the XML column. However, using the extract functions we can still obtain that information. Because the e-mail address can only occur once in an order document (according to the DTD shown in Example 5-2 on page 121), we can use the scalar version of the extract UDFs.

Example 5-2 getstart.dtd

```
<?xml encoding="US-ASCII"?>

<!ELEMENT Order (Customer, Part+)>
<!ATTLIST Order key CDATA #REQUIRED>
<!ELEMENT Customer (Name, Email)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Email (#PCDATA)>
<!ELEMENT Part (key,Quantity,ExtendedPrice,Tax, Shipment+)>
<!ELEMENT key (#PCDATA)>
<!ELEMENT Quantity (#PCDATA)>
<!ELEMENT ExtendedPrice (#PCDATA)>
<!ELEMENT Tax (#PCDATA)>
<!ATTLIST Part color CDATA #REQUIRED>
<!ELEMENT Shipment (ShipDate, ShipMode)>
<!ELEMENT ShipDate (#PCDATA)>
<!ELEMENT ShipMode (#PCDATA)>
```

Because we want to extract the e-mail address as a VARCHAR, we use the exctractVarchar UDF, as shown in Example 5-3.

Example 5-3 Using extractvarchar

```
db2 "select substr(db2xml.extractVarchar(order,'/Order/Customer/Email')
                    ,1,40) from sales_tab"

1
----------------------------------------
parts@am.com
parts2@eu.com

  2 record(s) selected.
```

Note that our sample table contains more rows than the sample table that ships with XML Extender. This is to be able to illustrate some additional items in later sections.

Example 2

Find out what parts have been ordered. Again, the part key is not stored in a side table, and we have to retrieve the information from the actual XML document stored in the XML column. Because we can have multiple parts in a single XML order document (see the DTD in Example 5-2), we use the table UDF flavour (ending on “s”), extractintegers. As part numbers are integer numbers, we use the integer data type (we may just as well use a character data type in this case, since we have no plans to do arithmetic on the extracted information). Example 5-4 shows our first attempt.

Example 5-4 Using extractintegers

```
db2 select * from table(db2xml.extractintegers(
                        (select order from sales_tab ),'/Order/Part/key')
                       ) a

RETURNEDINTEGER
---------------
SQL0811N  The result of a scalar fullselect, SELECT INTO statement, or
VALUES INTO statement is more than one row.  SQLSTATE=21000
```

Why did we receive an SQL error? We are using the table UDF to handle multi-occurrences. That is true, however, the first argument of the (table) UDF needs to be a single XML document. In our case, we have multiple rows in the sales_tab and the query fails. We can re-code the query as follows (Example 5-5):

Example 5-5 Using extractintegers -2

```
db2 "select *
      from table(db2xml.extractintegers(
                  (select order from sales_tab where invoice_num ='123456')
                  ,'/Order/Part/key'
                                      )
                ) a"

RETURNEDINTEGER
---------------
             68
            128

  2 record(s) selected.
```

Although this works, it only allows us to extract the part keys from a single XML document (row in the table), `where invoice_num = ‘123456’`, and not all part keys of all stored documents.

Since we want to look at all stored XML documents, we use the following SQL statement (Table 5-6):

Example 5-6 Using extractintegers -3

```
db2 select x.returnedinteger
      from sales_tab, table(db2xml.extractintegers(order,
                                               '/Order/Part/key')
                         ) as x

RETURNEDINTEGER
---------------
             68
            128
             99
            128

  4 record(s) selected.
```

In the query, we pass the order column of the sales_tab (containing the XML documents) for every row to the table UDF, to extract (one or more) part keys of that XML document. Using this technique we use the extractintegers table UDF to extract multiple occurrences of the part key out of an XML document, as well as request each row (XML document) in the sales_tab to be searched. Note that in the SELECT clause we only select the column (returnedinteger) that is returned by the table UDF (extractintegers).

Tip: Using the extract UDFs, we can access all the information stored in the XML document, even when no side table is constructed on the element or attribute. Note however that using the extract UDFs requires XML Extender to parse each individual XML document and retrieve the requested elements and attributes. This is ok for occasional searches and/or when the table does not contain a large number of XML documents, or only small XML documents. Therefore, for performance reasons, it is highly recommended to construct side tables on all frequently accessed elements and attributes.

5.2.3 Retrieving fragments from data stored in an XML column

You can use the **`ExtractCLOB()`** or **`ExtractCLOBs()`** UDFs to extract (a) fragment(s) of an XML documents, including its element and attribute markup, and content of elements and attributes, including sub-elements. The ExtractCLOB(s) function is different from the other extract functions.
The other extract functions (for example ExtractVarchar and ExtractVarchars)

only returns the content of elements and attributes. They are used to *return simple values*.
The ExtractCLOB and ExtractCLOBs functions are used to *extract document fragments*.

Therefore, if you want to extract more than just a single element content or the value of an attribute, you can use ExtractCLOB() or ExtractCLOBs() to do so. The difference between ExtractCLOB() and ExtractCLOBs() function is similar to comparing ExtractVARCHAR() and ExtractVARCHARs(). ExtractCLOB() is a scalar UDF and returns a single fragment, whereas ExtractCLOBs() returns a table, with potentially multiple fragments. When the element (and its subelements or attributes) that you extract can occur multiple times, you must use the table flavor of the UDF.

Example 5-7 shows how to extract the customer fragment from our order XML column. Note that it extracts the `<customer>` tag, as well as its dependent tags, `<Name>` and `<Email>`. Note also that when the result of the SQL statement (after applying the WHERE clause) returns multiple XML documents, the `<customer>` tags of all returned documents are extracted (as shown in the bottom part of the example).

Example 5-7 Using ExtractCLOB()

```
db2 select substr(db2xml.extractclob(order,'/Order/Customer'),1,100)as RESULT
     from sales_tab
     where invoice_num = '123456'

RESULT
-------------------------------------------------------------------------------
<Customer>
    <Name>American Motors</Name>
    <Email>parts@am.com</Email>
  </Customer>

  1 record(s) selected.


-- When more than one document is returned, all fragments from all xml column
-- rows are extracted.


db2 select substr(db2xml.extractclob(order,'/Order/Customer'),1,100) as RESULT
      from sales_tab

RESULT
-------------------------------------------------------------------------------
<Customer>
```

```
    <Name>American Motors</Name>
    <Email>parts@am.com</Email>
  </Customer>
<Customer>
    <Name>European Engines</Name>
    <Email>parts2@eu.com</Email>
  </Customer>
<Customer>
    <Name>ITSO Insurance, Inc.</Name>
    <Email>my-email@asia.gov</Email>
  </Customer>

  3 record(s) selected.
```

Example 5-8 shows how to use the ExtractCLOBs UDF. In the example, we extract the parts fragment. As there can be multiple parts within a single XML order document we must use the ExtractCLOBs function instead of the ExtractCLOB function.

Example 5-8 Using ExtractCLOBs

```
db2 select substr(a.returnedclob,1,400) as RESULT
      from sales_tab s,
           table(db2xml.extractclobs(s.order,'/Order/Part') )a
      where s.invoice_num = '123456'

RESULT
-------------------------------------------------------------------------------

<Part color="black ">
    <key>68</key>
    <Quantity>36</Quantity>
    <ExtendedPrice>34850.16</ExtendedPrice>
    <Tax>6.000000e-2</Tax>
    <Shipment>
      <ShipDate>1998-08-19</ShipDate>
      <ShipMode>BOAT  </ShipMode>
    </Shipment>
    <Shipment>
      <ShipDate>1998-08-19</ShipDate>
      <ShipMode>AIR   </ShipMode>
    </Shipment>
  </Part>
<Part color="red   ">
    <key>128</key>
    <Quantity>28</Quantity>
    <ExtendedPrice>38000.00</ExtendedPrice>
    <Tax>7.000000e-2</Tax>
    <Shipment>
      <ShipDate>1998-12-30</ShipDate>
      <ShipMode>TRUCK </ShipMode>
    </Shipment>
  </Part>


  2 record(s) selected.
```

Note that with this query can also be used when multiple (order) XML documents are processed (when multiple rows are returned after applying the conditions in the WHERE clause).

Using this UDF can be very useful when decomposing complex XML documents, especially when the standard DAD shredding capabilities are not sufficient.

5.2.4 Filtering the XML documents you want to retrieve

In this section we look at how to construct the WHERE clause of our SQL statements to filter which XML documents (or parts thereof) we want to retrieve.

Using side tables

When we set up the XML column, we can also specify for which elements and attributes we want XML Extender to create side tables through a DAD file (see Figure 5-1 on page 118). These side tables are the best way to guarantee good performance when retrieving information stored in XML columns. The side tables are populated when the documents are inserted.

You can use these side tables to perform filtering in the WHERE clause. For example, the price information is stored in a side table (part_side_tab). To retrieve the name of the salesperson that sold parts > $2500, you can use the following query (Example 5-9).

Example 5-9 Filtering using side tables

```
db2 "select sales_person
       from sales_tab
       where invoice_num in (select invoice_num
                               from part_side_tab
                               where price > 2500.00)"

SALES_PERSON
```

```
--------------------
Sriram Srinivasan
Willy The Whale

  2 record(s) selected.

OR

db2 "select sales_person
      from sales_tab T,part_side_tab S
      where T.invoice_num = S.invoice_num
        and price > 2500.00"

SALES_PERSON
--------------------
Sriram Srinivasan
Sriram Srinivasan
Willy The Whale

  3 record(s) selected.
```

We use the invoice_num column to “merge” the information from the side table (part_side_tab) that contains the price information, and the table that contains the XML documents (sales_tab) and the sales_person. Invoice_num is the primary key in the sales_tab and was specified as the root_id at XML column enable time (-r option). Therefore the invoice_num column also exists in all the side_tables.
Note that it is better to use a subselect (top query in Example 5-9 on page 126) than a join (bottom query in Example 5-9 on page 126) for this type of queries. Using a subselect filters out duplicates in the IN-list. This means that by using a join, you may end up with duplicates (as shown in the result of the bottom query of Example 5-9 on page 126). These duplicates may exist because of multiple occurrences of certain attributes or elements within the same document (multi_occurrence =”yes” elements or attributes). In our example above we use the price to do filtering. The price column in the side table is populated through the multi_occurrence element “ExtendedPrice”, which explain the fact that “Sriram Srinivasan” shows up twice in the result.

Using a joined view

When you enable the XML column, you can also have a default view created (using the -v option on the dxxadm enable_column command). This default view joins the application table and all the side tables that you specify in the DAD using a unique ID (specified on the -r option of the dxxadm **`enable_column`** command or a DXXROOT_ID column created by XML Extender). The advantage of the default view is that it provides a single virtual view of the application table and its side tables. The disadvantage is that the more side tables you have, the slower your queries against the default view will run, since each new side table adds an extra table to the join. For our sales table, using the DAD of Figure 5-1 on page 118, the default view definition looks like Example 5-10.

Example 5-10 Default view definition

```
CREATE VIEW sales_order_view
    AS SELECT SALES_TAB.INVOICE_NUM,SALES_TAB.ORDER,SALES_TAB.SALES_PERSON
             ,order_side_tab.order_key,order_side_tab.customer
             ,part_side_tab.price
             ,ship_side_tab.date
        FROM order_side_tab,part_side_tab,ship_side_tab,SALES_TAB
        WHERE SALES_TAB.invoice_num = order_side_tab.invoice_num
          AND SALES_TAB.invoice_num = part_side_tab.invoice_num
          AND SALES_TAB.invoice_num = ship_side_tab.invoice_num
```

Again, still looking for salespersons (sales_person) who have sold line item orders with a price over $2500, you can issue a simple query against the default view (Example 5-11).

Example 5-11 Retrieving data using the default view

```
db2 "select sales_person
       from sales_order_view
       where  price > 2500.00"

SALES_PERSON
--------------------
Sriram Srinivasan
Sriram Srinivasan
Sriram Srinivasan
Sriram Srinivasan
Sriram Srinivasan
Sriram Srinivasan
Willy The Whale
Willy The Whale
Willy The Whale

  9 record(s) selected.
```

Note that the result is even more prone to duplicates being returned, since the default view contains all combinations of the base and side tables. Note also that the filtering (price > 2500) is performed on a column that is part of a side table, included in the default view.

You can also create views yourself, for example joining the table containing the XML documents (sales_tab) with a single side table (order_side_tab) as shown in Example 5-12.

Example 5-12 Joining XMLcolumn table with an individual side table

```
CREATE VIEW sales_order_only_view
    AS SELECT SALES_TAB.INVOICE_NUM,SALES_TAB.ORDER,SALES_TAB.SALES_PERSON
              ,order_side_tab.order_key,order_side_tab.customer
        FROM order_side_tab,SALES_TAB
        WHERE SALES_TAB.invoice_num = order_side_tab.invoice_num
```

Using this view makes it easier to retrieve information related to the sales_tab and the order related information stored in its side table (order_side_tab). For example, you can retrieve the complete XML document related to orders from a certain customer (from the side table),

```
select order
  from sales_order_only_view
  where customer = 'American Motors'
```

When you try this using the sales_order_view (the default view created at XML column enable time),

```
select order
  from sales_order_view
  where customer = 'American Motors'
```

you will receive duplicates. Because you cannot use a DISTINCT on a CLOB column, using the sales_order_only_view is a good solution. Note that this technique may not work in cases where you have a side_table on a multi_occurrence=”yes” element or attribute.

Using Extractxxx() UDFs

You can also use XML Extender’s **`Extractxxx()`** UDFs to filter on elements and attributes on which you did not create side tables. As with retrieving attributes and elements in the SELECT clause of an SQL statement, using the extractxxx() UDFs in the WHERE clause requires XML Extender to retrieve the entire XML document and parse it. This can be a very expensive operation, especially for large XML documents, and/or tables that contain many XML documents, as DB2 XML Extender needs to parse all of them to be able to apply the filtering.

Example 5-13 gives a simple example of how to find the salespersons who did business with IBM.

Example 5-13 Using extract UDFs in the WHERE clause

```
C:\Program Files\IBM\SQLLIB\BIN>db2 "select sales_person from sales_tab
where  db2xml.extractvarchar(order,'/Order/Customer/Name') like 'IBM%'"

SALES_PERSON
--------------------

  0 record(s) selected.
```

Because the customer name is also part of a side table, it is probably more meaningful to select this information directly from the side table, as shown previously.

The next example (Example 5-14) lists all customers that have a government e-mail address (.gov). As mentioned above, the customer name itself is in a side table (order_side_table) and does not need to be extracted from the XML document. The e-mail address on the other hand, is not part of any side table and needs to be extracted. Again, we can write this as a subselect or a join. However, since invoice_number is the primary key of the sales table, there cannot be any duplicates. As a matter of fact, DB2 is likely to convert the subselect into a join during query rewrite.

Example 5-14 Using extract UDFs in the WHERE clause -2

```
db2 "select customer
       from order_side_tab
       where invoice_num in
             (select invoice_num
                from sales_tab
                where db2xml.extractvarchar(order,'/Order/Customer/Email')
                             like '%.gov')"

CUSTOMER
--------------------------------------------------
Asian Manicure

  1 record(s) selected.

db2 "select customer
       from order_side_tab a,sales_tab b
      where a.invoice_num = b.invoice_num
```

```
        and db2xml.extractvarchar(order,'/Order/Customer/Email')
                   like '%.gov'"

CUSTOMER
--------------------------------------------------
Asian Manicure

  1 record(s) selected.
```

To make things a bit more complicated, Example 5-15 tries to find the salesperson responsible for sending shipments by air (which are very expensive).

Example 5-15 Using extract UDFs in the WHERE clause -3

```
db2 "select sales_person
       from sales_tab
           ,table(db2xml.extractvarchars(order,
                                         '/Order/Part/Shipment/ShipMode')
                 ) x
       where x.returnedvarchar = 'AIR'"

SALES_PERSON
--------------------
Sriram Srinivasan

  1 record(s) selected.
```

Note that we need to use the table version of the UDF as we can have multiple shipments per order.

Predicates in the location path

Extractxxx() UDFs support location paths that have predicates with attributes, not elements. This means that if you need to do filtering on an attribute, you can do it within the UDF itself, and not as part of the WHERE clause of the SQL statement.

In Example 5-16, we retrieve the invoice number and the price of all black orders.

Example 5-16 Filtering inside the extract UDFs

```
db2 "select invoice_num, x.returnedreal
       from sales_tab
           ,table(db2xml.extractreals(
                       order
                      ,'/Order/Part[@color="""black """]/ExtendedPrice'
                                     )
                 )x"

INVOICE_NUM RETURNEDREAL
----------- ------------------------
123456                 +3.48502E+004
555555                 +1.23160E+002

  2 record(s) selected.
```

As part of the extract UDF, we code the filtering on color while retrieving the price.

```
'/Order/Part[@color="""black """]/ExtendedPrice'
```

Note that in a “native” DB2 CLP window, we have to treble the “ (double quotes). This is not required in the DB2 Command Center.

Note also that you must specify a single blank after the filtering predicate, because the data in the original XML document that was inserted contains that blank (`<Part color="black ">`).

```
[@color="""black """]
```

So, you must specify “black “ and not “black”.

> **Tip:** Be careful when using Internet Explorer to view XML documents. It seems to consider trailing blanks as insignificant and displays the <part> element above as `<Part color="black">`

As mentioned before, unfortunately you can only use this filtering inside the UDF on attributes, not on elements. Therefore Example 5-17 returns an error.

Example 5-17 Filtering inside the extract UDFs -2

```
db2 "select sales_person
       from sales_tab
           ,table(db2xml.extractvarchars
                        ( order
                         ,'/Order/Part/Shipment/ShipMode = """TRUCK """')
                 ) x "

SALES_PERSON
--------------------
```

```
SQL0443N  Routine "*VARCHARS" (specific name "") has returned an error
SQLSTATE with diagnostic text "DXXD002E  A syntax error occurred near
position "29" in the".  SQLSTATE=38X09
```

Since filtering inside the Extractxxx() UDF is likely to be cheaper than having to pass the results back to the SQL layer, and do the filtering in the WHERE clause, when you expect to do ad hoc queries and you are in charge of the layout of the XML documents, it may be worth considering to design potential search fields as attributes instead of elements. If multiple predicates are required, that means multiple invocations of the Extracxxx() function, and will require multiple parsings.

5.3 Retrieving XML columns using an XML wrapper

The XML wrapper is part of the family of DB2 Information Integrator's non-relational wrappers. It allows you to define a "relational look" over an XML document. For more information on using the XML wrapper, see Chapter 11, "XML wrapper" on page 305.

The XML wrapper can also provide read-only access to XML documents that are stored in an XML column. To illustrate its use with an XML column, we use a number of XML documents stored in an XMLFILE XML column. They were inserted into the XMLFILE column (order) using statements similar to those shown in Example 5-18.

Example 5-18 Inserting into XMLFILE XML column

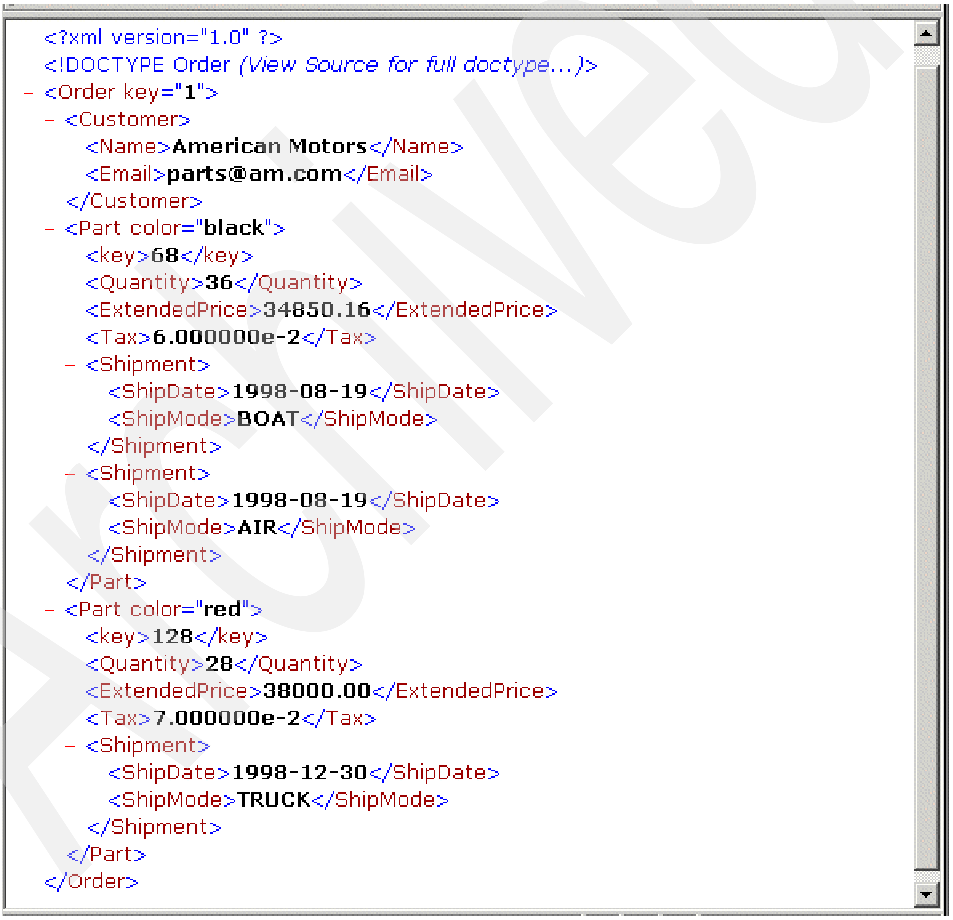
```
db2 "insert into sales_tab(invoice_num, sales_person, order)
      values('123456'
             ,'Sriram Srinivasan'
             ,db2xml.xmlfilefromvarchar(
                 db2xml.varchar(
                    db2xml.XMLVarcharFromFile('C:\SQLLIB\samples\db2xml\xml
\getstart.xml'
                                            )
                              )
                ,'C:\XMLDOCS\getstart.xml'
                                      )
            )"
```

Because in our example, the input is made up of existing XML documents in the file system, inserting them into an XMLFILE column may seem a bit artificial.

Notice that we read the XML document from the file system (`C:\SQLLLIB\samples\db2xml\xml`) using the **`XMLVarcharFromFile`** UDF, convert the result to a VARCHAR, and insert the VARCHAR into the XMLFILE order column, resulting in the file to be put into the C:\XMLDOCS directory.

However, in real life it may very well be that the XML document comes in via MQ Series or any other type of application, and has to be stored into an XMLFILE XML column. In addition, by inserting into the XMLFILE column, the side tables are automatically populated and can be used to access the data as well.

The source of the XML document that we inserted in Example 5-18 on page 133 is shown in Figure 5-2.



```
  <?xml version="1.0" ?>
  <!DOCTYPE Order (View Source for full doctype...)>
- <Order key="1">
  - <Customer>
      <Name>American Motors</Name>
      <Email>parts@am.com</Email>
    </Customer>
  - <Part color="black">
      <key>68</key>
      <Quantity>36</Quantity>
      <ExtendedPrice>34850.16</ExtendedPrice>
      <Tax>6.000000e-2</Tax>
    - <Shipment>
        <ShipDate>1998-08-19</ShipDate>
        <ShipMode>BOAT</ShipMode>
      </Shipment>
    - <Shipment>
        <ShipDate>1998-08-19</ShipDate>
        <ShipMode>AIR</ShipMode>
      </Shipment>
    </Part>
  - <Part color="red">
      <key>128</key>
      <Quantity>28</Quantity>
      <ExtendedPrice>38000.00</ExtendedPrice>
      <Tax>7.000000e-2</Tax>
    - <Shipment>
        <ShipDate>1998-12-30</ShipDate>
        <ShipMode>TRUCK</ShipMode>
      </Shipment>
    </Part>
  </Order>
```

Figure 5-2 getstart.xml document

The hierarchical structure of the XML document is shown in Figure 5-3.

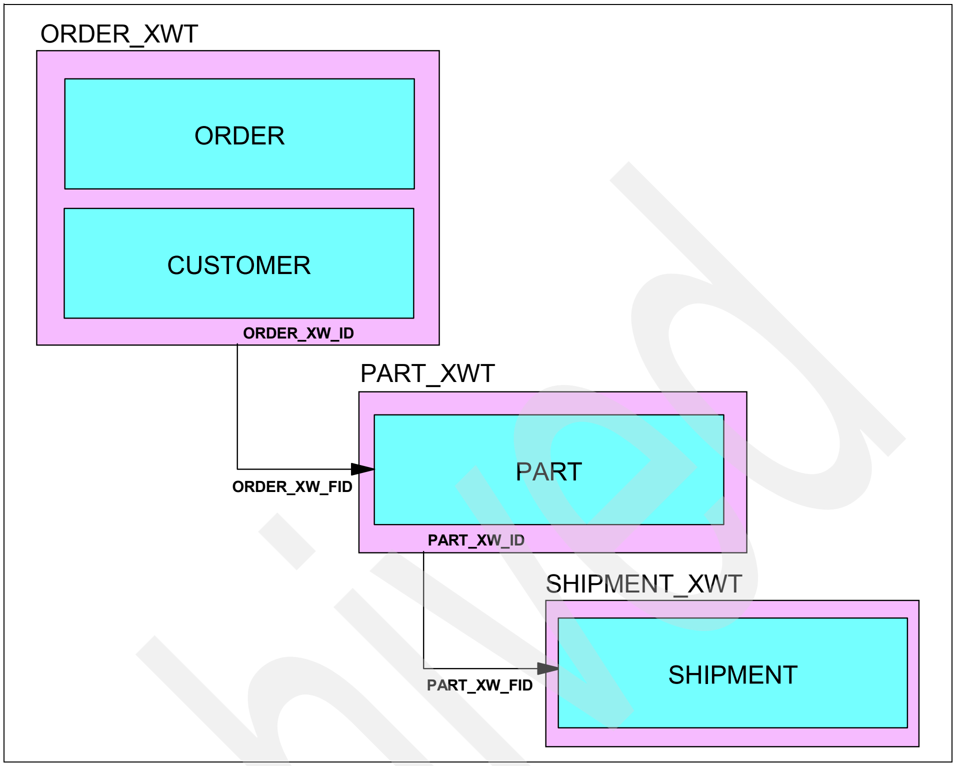


Figure 5-3 XML document structure

Note that this structure was derived from the XML document shown in Figure 5-2 on page 134. If you have an XML Schema or DTD definition, you should use that instead, as a sample document may not represent the full structure.

We use the following DDL to set up the wrapper, server, nicknames and views (Example 5-19).

Example 5-19 Setting up the XML wrapper definitions

```
CREATE WRAPPER "XMLWRAP" LIBRARY 'db2lsxml.dll';

CREATE SERVER XMLSERV WRAPPER "XMLWRAP";

CREATE NICKNAME ORDER_XWT ( CUSTOMER_NAME VARCHAR (48) OPTIONS(XPATH'./Customer/Name/text()')
                           ,CUSTOMER_EMAIL VARCHAR (48) OPTIONS(XPATH'./Customer/Email/text()')
                           ,ORDER_XWT_ID VARCHAR (16) OPTIONS(PRIMARY_KEY 'YES')
                           ,KEY INTEGER OPTIONS(XPATH './@key'))
       FOR SERVER "XMLSERV"  OPTIONS(XPATH '//Order' , DIRECTORY_PATH 'C:\XMLDOCS');
```

```
CREATE NICKNAME PART_XWT ( KEY INTEGER OPTIONS(XPATH './key/text()')
                          ,QUANTITY INTEGER OPTIONS(XPATH './Quantity/text()')
                          ,EXTENDEDPRICE DECIMAL (10,2) OPTIONS(XPATH'./ExtendedPrice/text()')
                          ,TAX REAL OPTIONS(XPATH './Tax/text()')
                          ,PART_XWT_ID VARCHAR (16) OPTIONS(PRIMARY_KEY 'YES')
                          ,COLOR VARCHAR (48) OPTIONS(XPATH './@color')
                          ,ORDER_XWT_FID VARCHAR (16) OPTIONS(FOREIGN_KEY 'ORDER_XWT'))
       FOR SERVER "XMLSERV"  OPTIONS(XPATH './Part');

CREATE NICKNAME SHIPMENT_XWT ( SHIPDATE DATE OPTIONS(XPATH './ShipDate/text()')
                              ,SHIPMODE VARCHAR (48) OPTIONS(XPATH'./ShipMode/text()')
                              ,PART_XWT_FID VARCHAR (16) OPTIONS(FOREIGN_KEY 'PART_XWT'))
       FOR SERVER "XMLSERV"  OPTIONS(XPATH './Shipment');

CREATE VIEW ORDER_XWV AS
       SELECT Order.Customer_Name, Order.Customer_Email, Order.Order_XWT_ID, Order.key
         FROM   Order_XWT Order;

CREATE VIEW PART_XWV AS
       SELECT Part.key, Part.Quantity, Part.ExtendedPrice, Part.Tax, Part.Part_XWT_ID
             ,Part.color, Order.Order_XWT_ID
         FROM   Part_XWT Part, Order_XWT Order
         WHERE  Order.Order_XWT_ID = Part.Order_XWT_FID;

CREATE VIEW SHIPMENT_XWV AS
       SELECT Shipment.ShipDate, Shipment.ShipMode, Part.Part_XWT_ID
         FROM   Shipment_XWT Shipment, Part_XWT Part, Order_XWT Order
         WHERE  Part.Part_XWT_ID = Shipment.Part_XWT_FID
           AND  Order.Order_XWT_ID = Part.Order_XWT_FID;
```

For more details on the meaning of all these definitions, see Chapter 11, “XML wrapper” on page 305. For now, it is sufficient to know that these definitions allow us to access *all* XML documents (files with an .xml extension) that reside in the C:\XMLDOCS directory via SQL statements. The different views allow us to address the different hierarchical levels (order, customer, part, and shipment) inside the XML documents. Because we validated the documents (via the DAD specification at enable XML column time), we can be sure all documents have the same “XML structure.”

Note: We use the DIRECTORY_PATH option when creating the root-level nickname. This way, we access the XML documents directly in the file system, and not via the XML column. We can access XML documents through the XML column by using the DOCUMENT COLUMN option. However, using the DOCUMENT COLUMN option, you can only access a single XML document at a time, whereas using by using the DIRECTORY_PATH option, you can access all XML documents in the directory.

Now let us look at a few questions that we can answer using XML wrapper functionality.

Example 1

List the shipping modes we used for shipments since 01/01/2000. The query and result is shown in Example 5-20.

Example 5-20 Query against XML wrapper nickname

```
db2 "select distinct (shipmode)
       from shipment_xwv
       where shipdate > '01/01/2000'"

SHIPMODE
------------------------------------------------
BIKE
ELEPHANT
FOOT
MULE

  4 record(s) selected.
```

Note that we used a view in the previous example. When querying parts of an XML document within the same "hierarchy" level, it is easier to use the views, since they give you direct access to this information.

Example 2

Find out the parts, customer, and order number for orders that were shipped by bike (Example 5-21).

Example 5-21 Query against XML wrapper nickname -2

```
db2 "select o.customer_name, o.key as order_key, p.key as part_key
          , p.quantity from part_xwt p , order_xwt o, shipment_xwt s
```

```
      where  o.order_xwt_id = p.order_xwt_fid
        and p.part_xwt_id = s.part_xwt_fid
        and s.shipmode ='BIKE'"

CUSTOMER_NAME                        ORDER_KEY   PART_KEY    QUANTITY
------------------------------------ ----------- ----------- -----------
European Engines                               2         128          58

  1 record(s) selected.
```

In the example above, we use the actual nicknames instead of the views. It may complicate coding somewhat but it offers more flexibility.

5.4 Updating XML documents stored in an XML column

There are two ways to update an XML document that is residing in an XML column:

- Using the SQL UPDATE statement
- Using the Update() UDF

5.4.1 Using the SQL UPDATE statement

You can use the UPDATE SQL statement to *replace* an XML document by another one. For example to replace the existing document stored in the XML column “order”, with a new XML document located on the file system, you can use the following SQL statement:

```
update sales_tab_clob set order =
                      db2xml.XMLCLOBFromFile('c:\SG246994\orderbis.xml')
       where invoice_num = ‘123456’;
```

When you replace an XML document using the SQL UPDATE statement, the side tables are immediately updated to stay synchronized with the content of the XML document.

5.4.2 Using the Update() UDF

You can use the **Update()** UDF that comes with XML Extender to replace the XML document stored in an XML column, by only changing an element or attribute value without needing a complete updated XML document in the file system. The Update() function uses the location path to locate the attribute or element whose value must be changed.

For example, the following SQL statement changes the value of the Name element of the XML document stored in the XML column that has an invoice number of 999999:

```
update sales_tab_xclob set order =
                       db2xml.Update(order,’/Order/Customer/Name’ , ‘IBM’)
       where invoice_num = ‘999999’;
```

Note that the Update() UDF also automatically updates the side tables if the element or attribute that you are updating is part of a side table.

If the location path that you specify occurs more than once in a document, the Update() UDF replaces all of the existing values with the value provided in the Update() UDF (‘IBM’ in the example above). So the value of every element or attribute that matches the location path is updated.
This is important when using the Update() UDF with XML documents with multiple occurrences. The following statement:

```
update sales_tab set order =
                 db2xml.Update(order,'/Order/Part/Tax' , '9.000000e-2')
       where invoice_num = '555555'
```

updates the tax value for all parts in the order with invoice number 555555. If there are multiple `<Part>` elements in the order with a `<Tax>` element, all of them will be replaced by ‘9.000000e-2’.

As with the Extractxxx() UDFs, the Update() UDF allows filtering on attributes, not on elements. For more details, see “Predicates in the location path” on page 131. If you can perform attribute filtering in the location path, only those that qualify the attribute filter are updated. The following example updates the `<Tax>` element in XML documents that correspond with invoice number 555555, but only for those parts that have a black color.

```
db2 update sales_tab set order =
                     db2xml.Update( order
                                   ,'/Order/Part[@color="""black """]/Tax'
                                   ,'6.000000e-2')
           where invoice_num ='555555'
```

However, if you cannot use an attribute filter to limit which attributes or elements you want updated in an XML with multiple occurrences of an element, you should not use the Update() UDF.

Another thing you should be aware of when using the Update() UDF is that it can change the “look” of your XML document. This has to do with the fact that the document is parsed, processed and put together again. Based on the output of the XML parser, some parts of the original document are preserved, while others are lost or changed. Usually the changes are not major, but it is definitely worthwhile checking out the details in section of the XML Extender Administration and Programming manual that describes the Update() UDF.

5.5 Deleting XML documents stored in an XML column

You can use the SQL DELETE statement to delete the row containing an XML document from an XML column. You can specify a WHERE clause to delete specific documents.

Example 5-22 deletes all documents that have a value for <ExtendedPrice> greater than 2500.00. As the <ExtendedPrice> element is part of a side table (part_side_table), we can use the side table to quickly retrieve the corresponding invoice numbers and use them to delete the corresponding rows containing the XML column (order) rows from the table.

Example 5-22 Deleting XML documents from an XML column

```
DELETE from sales_tab_clob
 WHERE invoice_num in (SELECT invoice_num
                         FROM part_side_tab
                         WHERE price >2500.00
                     )
```

The corresponding rows in the side tables are automatically deleted when the XML document in the XML column is deleted (through the delete triggers). This is also the case when you archive your XML documents elsewhere, and delete them from their original location in the XML column.

5.6 Best practices

In this section, we give a few guidelines on how to retrieve information from XML documents stored in an XML column.

5.6.1 Use side tables as much as possible

Using side tables is the fastest way to retrieve information that is stored in an XML column. This is true for information that you pass back to the application (columns in the SELECT clause), as well as when filtering out which information to pass back (WHERE clause predicates). If the information exists in a side table, you should always use it.

Because it is very difficult to add or change side tables after the XML column is populated with data, it is imperative to implement a good design of the side tables from day one. Here are a few tips:

- Make sure that all elements and attributes that are frequently retrieved or used to filter documents exist in a side table. Because maintaining the side tables is done at insert, update and delete time, you do not want to create a side table column for every element and attribute. It would increase the insert, update, and delete time considerably.
- Whenever possible, put multiple, related fields in the same side table. This minimizes the number of joins in queries. (Each side table is joined back to the table with the XML column in the default view.)
- Be sure to create an index on the columns in the side table. XML Extender does not automatically create an index on the side table column when the side table is created.

We *strongly recommend* that you do *not* update the side tables directly, because doing so can easily cause data inconsistency problems. You must update the original XML documents stored in the XML columns. Those will trigger automatic updates of the side tables.

5.6.2 Where to filter

When filtering documents, try to perform filtering using the following techniques, preferably in the order presented here:

1. Filter using side table columns in the WHERE clause. It prevents DB2 from having to parse the XML document, and as you normally have an index on the columns in the side table, DB2 can use an access path that uses the index for fast retrieval of the value and evaluation of the WHERE clause predicate.
2. If the element or attribute that you want to filter on, is not available in a side table, you can use the Extractxxx() UDF in a WHERE clause.
3. In addition, when it is possible to do additional filtering inside the UDF, by specifying a simple predicate. It can be worth doing so, as it can avoid having to parse the document multiple times. However, be aware that location path filtering only applies to simple predicates, and is only allowed on attributes as well. Another example is shown in Example 5-23.

Example 5-23 Exctractxxx() UDF filtering

```
select sales_person
 from sales_tab
   ,table(db2xml.extractvarchars(order, '/Order/Part/Shipment/ShipMode')
         ) x
   ,table(db2xml.extractvarchars(order, '/Order/Part/@color')
         ) y
 where y.returnedvarchar = 'black'
   and x.returnedvarchar = 'FOOT'

-- ---------------------------------------------------------------------

select sales_person
  from sales_tab
      ,table(db2xml.extractvarchars(
                 order
                 ,'/Order/Part[@color="black "/Shipment/ShipMode'
                                  )
            ) x
  where x.returnedvarchar = 'FOOT'
```

Both queries return the same result. However, in the top query we parse the order documents twice, whereas we only parse the documents once in the bottom query, which should help performance considerably, especially when dealing with many and/or long documents.

4. Use filtering in the location path in the SELECT clause only if the filtering cannot be done in the WHERE clause. Again, if possible, it can be worthwhile to do the filtering in the SELECT clause to avoid having to parse the document multiple times.

5.6.3 Using location path expressions

Queries using a location path do not scale well because the XML document must be parsed (and potentially parsed multiple times). Therefore, use location path queries only in the following situations:

- Infrequently occurring ad hoc queries
- Ad hoc queries that extract information from single documents that have been filtered out using predicates on side table columns in the WHERE clause
- Querying all documents in an XML column only if there are very few columns.

6

Shredding XML into relational tables

In this chapter, we discuss how to decompose XML documents into DB2 relational tables. The following topics are covered:

- An overview of the shredding concepts using DB2 XML Extender and the XML wrapper
- A detailed discussion on shredding with DB2 XML Extender, including:
- Planning and design
- Configuration and execution
- Alternatives to XML Extender to decompose XML documents, such as writing your own code or using the XML wrapper for shredding
- A set of best practices
- A step-by-step example

6.1 Shredding overview

Before going into the shredding details, we start out with an overview of what is meant by shredding XML documents, and when you want to use this technique.

6.1.1 What is shredding?

Shredding, or decomposing, is the process of breaking down an XML document and storing the contents of the XML elements or attributes (the data contained in your XML document) in new or existing database tables. The data is stored untagged in the relational tables. Note that shredding is not restricted to just relational tables. However, in this publication, we assume that we are shredding into a relational database management system, DB2 in particular.

In order to transfer data between XML documents and a relational database, it is necessary to map the schema of the XML document to the relational database schema. To be able to do this, the structure of the document must exactly match the structure expected by the mapping. Otherwise, prior to transferring data to the database, the XML document has to be transformed (using XSLT transformation for instance) to transform the original document into one that matches the structure expected by the mapping.

In this chapter, we give a detailed description of how to decompose XML documents into DB2 relational tables, using both the XML collection features of XML Extender, as well as the XML wrapper provided by DB2 Information Integrator (DB2 II).

6.1.2 When should you use shredding?

In addition to allowing storage of an entire XML document in a column of a relational table using an XML column (see 4.4, “Storing intact XML documents with XML Extender” on page 101for details), DB2 XML Extender also provides a method for storing XML documents in relational tables, by shredding, or decomposing them. This is called an *XML collection*.

An XML collection is a set of relational tables that contain data mapped to an XML document. The XML collection method allows you to decompose XML documents into DB2 tables.

You can also use XML collections when you need to generate XML documents from a particular set of (normal) relational column data. If the source information is composed of XML documents, you need to decompose (shred) the documents first, and store them into regular relational tables. Then, you can generate “new” XML documents based on that now relational information. For more information about composing XML documents from relational data, see Chapter 8, “Publishing data as XML” .

Shredding into relational tables has an advantage when you need to update individual attributes and/or elements on a regular basis. When the XML documents are stored in XML columns, updating parts of the document involves reading the entire document, parsing it, performing the update, re-assembling the XML document, and writing the entire document back to the database. This can be a time-consuming task when frequent updates have to be made, especially on large documents.
If the XML document is shredded, you can use normal SQL UPDATE statements to update individual elements and attributes that are now stored as columns of a relational table. This will improve performance considerably.

You use a *Document Access Definition* file (DAD file) to map XML data to DB2 tables using the XML collection access and storage methods. The DAD file is an important part of administering DB2 XML Extender. It specifies how the XML document structure relates to your DB2 data that resides in relational tables. The DAD file is discussed in more detail in 6.2, “Shredding using DB2 XML Extender” on page 146.

We recommend using XML collections in the following situations:

- You have XML documents that map well to an existing relational model, and the XML documents contain information that needs to be stored with existing data in relational tables.
- You want to create different views of your relational data using different mapping schemes.
- You have XML documents that come from other data sources. You are interested in the data but not the tags, and want to store pure data in your database. You want the flexibility to decide whether to store the data in existing tables or in new tables.
- A small subset of your XML documents needs to be updated often, and performance of those updates is critical. Since the data is stored in regular DB2 columns after shredding, you can use normal SQL UPDATE statements (therefore not using XML Extender functionality, since shredding itself does not support updates) to directly update those columns that require changes.
- If you do not need to store the XML document intact, but still want to store all data contained in the XML document, you can shred the XML document into relational tables, and retrieve only those columns that are currently used by applications using normal SQL SELECT operations.
- You have data in existing relational tables, you want to compose XML documents (that adhere to a certain DTD), and you do not have SQL/XML at your disposition because you are not yet using DB2 V8.

As an alternative to using XML collections, you can use the *XML wrapper* for seamless access to XML documents, providing the user with a relational 'view' of the XML document. XML wrapper in a way simulates the decomposition of the XML document into relational tables. However, keep in mind that the XML wrapper offers only SELECT functionality, and does not allow you to make any changes (INSERT, UPDATE or DELETE) to your XML document data.

6.2 Shredding using DB2 XML Extender

We now discuss shredding using DB2 XML Extender in more detail.

6.2.1 Shredding — planning and design

As explained in Chapter 2, "XML services in DB2 and DB2 Information Integrator" , at the time of enablement, an XML collection must be configured with a DAD file. The purpose of the DAD file is to map relational tables and columns to the XML data.

Two types of mapping between relational and XML are supported by DB2 XML Extender's DAD file:

- *RDB node mapping*. This is an object-relational mapping which can be used to map information in a way that it specifies the relationship between tables and columns in the relational model, and element and attribute values in the XML documents. RDB node mapping allows both SELECT statements (composition - publishing), and INSERT statements (shredding) to use the same DAD document. An RDB node DAD document can be used both for decomposition (going from an XML document to a set of relational tables), or publishing (going from a set of relational tables to an XML document).
- *SQL mapping*. This is a template-based language in which the user specifies an SQL SELECT statement and states where the results of the query should be placed within the template (that defines the layout of the XML document). SQL mapping can only be used to transfer data from relational tables to an XML document, and cannot be used for decomposition. See 8.2.2, "Publishing XML documents using a DAD file" on page 223 for more details on using SQL mapping for publishing.

Since we are interested in storing (decomposing) XML documents in this chapter, we use the bi-directional RDB node mapping to shred XML data. (As suggested by the name 'bi-directional', RDB node mapping can also be used for publishing XML data from relational. For more information, see 8.2.2, "Publishing XML documents using a DAD file" on page 223.

How to create a decomposition DAD file

The main difference between a DAD file used for composition (publishing) and DAD file used for decomposition (shredding) is that the column type for each attribute or text node that you intend to map must be specified. Column types are defined by adding the “type” attribute to the column element, as shown in Example 6-1.

Example 6-1   Column type specification for RDB node during shredding

```
<element_node name="firstname">
   <text_node>
      <RDB_node>
         <table name="EMPLOYEE"/>
         <column name="firstnme" type="VARCHAR(12)"/>
      </RDB_node>
   </text_node>
</element_node>
```

A sample DAD file

Example 6-4 on page 150 shows a DAD document using RDB node mapping, specifying a mapping between the XML document shown in Example 6-2, and the EMPLOYEE and EMP_ACT tables of DB2 SAMPLE database.

Example 6-2   An XML document with employee information

```
<?xml version="1.0" encoding="UTF-8"?>
<employees>
   <employee id="EMP010" dept="X01" sex="M">
      <firstname>Olivier</firstname>
      <midnameinit>G</midnameinit>
      <lastname>Guyennet</lastname>
      <educlevel>15</educlevel>
      <hiredate>2001-04-01</hiredate>
      <job>TS</job>
      <projactivity>
         <project>XML001</project>
         <activity>100</activity>
         <time>1</time>
         <startdate>2003-04-28</startdate>
         <enddate>2003-06-13</enddate>
      </projactivity>
   </employee>
</employees>
```

Example 6-3 shows the DDL for the definition of the EMPLOYEE and EMP_ACT table.

Example 6-3 EMPLOYEE and EMP_ACT table definitions

```
   CREATE TABLE "EMPLOYEE"  (
           "EMPNO" CHAR(6) NOT NULL ,
           "FIRSTNME" VARCHAR(12) NOT NULL ,
           "MIDINIT" CHAR(1) NOT NULL ,
           "LASTNAME" VARCHAR(15) NOT NULL ,
           "WORKDEPT" CHAR(3) ,
           "PHONENO" CHAR(4) ,
           "HIREDATE" DATE ,
           "JOB" CHAR(8) ,
           "EDLEVEL" SMALLINT NOT NULL ,
           "SEX" CHAR(1) ,
           "BIRTHDATE" DATE ,
           "SALARY" DECIMAL(9,2) ,
           "BONUS" DECIMAL(9,2) ,
           "COMM" DECIMAL(9,2) )
          IN "USERSPACE1" ;


   CREATE TABLE "EMP_ACT"  (
           "EMPNO" CHAR(6) NOT NULL ,
           "PROJNO" CHAR(6) NOT NULL ,
           "ACTNO" SMALLINT NOT NULL ,
           "EMPTIME" DECIMAL(5,2) ,
           "EMSTDATE" DATE ,
           "EMENDATE" DATE )
          IN "USERSPACE1" ;
```

The mapping between the XML document shown above, and the EMPLOYEE and the EMP_ACT tables in the SAMPLE database of DB2, that is building a DAD file using RDB node mapping, can be done as follows:

1. A DAD file is also an XML document. It needs a definition for a header, and a root element. The `<DAD>` root element contains all the other elements.

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <!DOCTYPE DAD SYSTEM "C:\PDB2\SQLLIB\SAMPLES\DB2XML\DTD\dad.dtd">
   <DAD>
   ```

2. If you intend to validate your XML document against a DTD in the DTD repository table, set the `<validation>` element to “YES”.

   ```
   <validation>YES</validation>
   ```

In our example, we are not validating, so we insert the following code:

```
<validation>NO</validation>
```

3. Next, you use the <Xcollection> and </Xcollection> tags to indicate that the storage (and access) method is an XML collection.

```
<Xcollection>...</Xcollection>
```

4. When using the RDB_node mapping, you need to use the RDB_node element for defining the element_node, attribute_node and text_node of the XML document.

Now, we discuss the RDB_node element, and its use in element, attribute and text nodes in more detail. Note that the spelling for RDB_node is case sensitive.

Definition of the RDB_node for the top element

The top `element_node` in the DAD file represents the root element of the XML document. When specifying an RDB_node for the top element:

- Specify all tables that are associated with the XML document. For the mapping in our example, we must specify the two tables involved in the RDB_node of the `<employees>` element in Example 6-2 on page 147, which is the top element node.
- Also specify the join condition(s) between the tables in the top element node in a `<condition>` tag.

```
<element_node name="employees">
   <RDB_node>
      <table name="EMPLOYEE"/>
      <table name="EMP_ACT"/>
      <condition>
         EMPLOYEE.empno=EMP_ACT.empno
      </condition>
   </RDB_node>
```

Definition of the RDB_node for an attribute_node

You also need to define an RDB_node for each attribute_node, to specify the table and column that map the XML data in the attribute_node. In our example, the table and column name for the attribute “id” of the employee element are table EMPLOYEE and column EMPNO. This is reflected in the DAD file as follows:

```
<element_node name="employee">
   <attribute_node name="id">
      <RDB_node>
         <table name="EMPLOYEE"/>
         <column name="empno" type="CHAR(6)"/>
      </RDB_node>
   </attribute_node>
```

Definition of the RDB_node for a text_node

You need to define an RDB_node for each text_node to specify the mapping between the table and column, and the data in the XML document. In our example, the text_node for the element `<firstname>` is mapped to column `firstnme` of table `EMPLOYEE`.

```
<element_node name="firstname">
   <text_node>
      <RDB_node>
         <table name="EMPLOYEE"/>
         <column name="firstnme" type="VARCHAR(12)"/>
      </RDB_node>
   </text_node>
</element_node>
```

Let us now look at the complete DAD document used for mapping of the XML document shown in Example 6-2 on page 147 to the EMP and EMP_ACT tables.

Example 6-4 The DAD file for employee XML document

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DAD SYSTEM "C:\DB2\SAMPLES\DB2XML\DTD\dad.dtd">
<DAD>
   <validation>NO</validation>
   <Xcollection>
      <prolog>?xml version="1.0"?</prolog>
      <doctype/>
      <root_node>
         <element_node name="employees">
            <RDB_node>
               <table name="EMPLOYEE"/>
               <table name="EMP_ACT"/>
               <condition>
                  EMPLOYEE.empno=EMP_ACT.empno
               </condition>
            </RDB_node>
            <element_node name="employee">
               <attribute_node name="id">
                  <RDB_node>
                     <table name="EMPLOYEE"/>
                     <column name="empno" type="CHAR(6)"/>
                  </RDB_node>
               </attribute_node>
               <attribute_node name="dept">
                  <RDB_node>
                     <table name="EMPLOYEE"/>
                     <column name="workdept" type="CHAR(3)"/>
                  </RDB_node>
```

```
</attribute_node>
<attribute_node name="sex">
   <RDB_node>
      <table name="EMPLOYEE"/>
      <column name="sex" type="CHAR(1)"/>
   </RDB_node>
</attribute_node>
<element_node name="firstname">
   <text_node>
      <RDB_node>
         <table name="EMPLOYEE"/>
         <column name="firstnme" type="VARCHAR(12)"/>
      </RDB_node>
   </text_node>
</element_node>
<element_node name="midnameinit">
   <text_node>
      <RDB_node>
         <table name="EMPLOYEE"/>
         <column name="midinit" type="CHAR(1)"/>
      </RDB_node>
   </text_node>
</element_node>
<element_node name="lastname">
   <text_node>
      <RDB_node>
         <table name="EMPLOYEE"/>
         <column name="lastname" type="VARCHAR(15)"/>
      </RDB_node>
   </text_node>
</element_node>
<element_node name="educlevel">
   <text_node>
      <RDB_node>
         <table name="EMPLOYEE"/>
         <column name="edlevel" type="SMALLINT"/>
      </RDB_node>
   </text_node>
</element_node>
<element_node name="hiredate">
   <text_node>
      <RDB_node>
         <table name="EMPLOYEE"/>
         <column name="hiredate" type="DATE"/>
      </RDB_node>
   </text_node>
</element_node>
<element_node name="job">
   <text_node>
```

```
         <RDB_node>
            <table name="EMPLOYEE"/>
            <column name="job" type="CHAR(8)"/>
         </RDB_node>
      </text_node>
   </element_node>
   <element_node name="projactivity" multi_occurrence="YES">
      <element_node name="project">
         <text_node>
            <RDB_node>
               <table name="EMP_ACT" />
               <column name="projno" type="CHAR(6)" />
            </RDB_node>
         </text_node>
      </element_node>
      <element_node name="activity">
         <text_node>
            <RDB_node>
               <table name="EMP_ACT" />
               <column name="actno" type="SMALLINT" />
            </RDB_node>
         </text_node>
      </element_node>
      <element_node name="time">
         <text_node>
            <RDB_node>
               <table name="EMP_ACT" />
               <column name="emptime" type="DECIMAL(5,2)" />
            </RDB_node>
         </text_node>
      </element_node>
      <element_node name="startdate">
         <text_node>
            <RDB_node>
               <table name="EMP_ACT" />
               <column name="emstdate" type="DATE" />
            </RDB_node>
         </text_node>
      </element_node>
      <element_node name="enddate">
         <text_node>
            <RDB_node>
               <table name="EMP_ACT" />
               <column name="emendate" type="DATE" />
            </RDB_node>
         </text_node>
      </element_node>
   </element_node>
</element_node>
```

```
            </element_node>
         </root_node>
      </Xcollection>
   </DAD>
```

6.2.2 Planning and design: hints and tips

In this section, we describe a set of recommendations and things that are worth knowing when setting up your shredding environment.

XML document design

If you have control over the structure of the XML document, we provide some design guidelines for XML documents, in case they need to be shredded by DB2 XML Extender.

Mixed content

An element can have four types of content:

- Empty content. An element with empty content is an element that carries no information such as the element below:

```
   <data /> or <data></data>
```

- Simple content. An element has simple content when it carries only text (also known as text content). A simple content element can look like this:

```
      <data>Hello Universe</data>
```

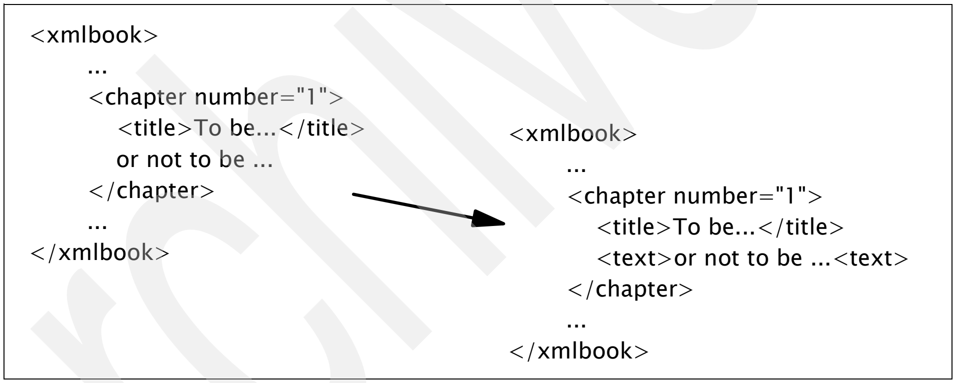
- Element content. An element is said to have element content when it contains other elements. It is also said to be a parent element.

```
   <data>
      <child1>Hello</child1>
      <child2>World</child2>
   </data>
```

- Mixed content. An element with mixed content is an element that contains both elements and text.

```
<data>My flowers are Beautiful.
   <child1>Hello</child1>
   <child2>World</child2>
</data>
```

DB2 XML Extender does not support shredding of elements with mixed content. Suppose you have an XML document that you want to decompose and that XML document has elements with mixed contents. In order to be able to shred it into tables, you would have to go through a transformation of your XML document in order to change the mixed content elements to either simple content elements, or elements content elements. This is illustrated in Figure 6-1 below, where the element `<chapter>` has an attribute `number`, and contains both an element `<title>` and text “or not to be ...“ . After transformation, the text part is moved under a (new) element `<text>`.



```
<xmlbook>
    ...
    <chapter number="1">
       <title>To be...</title>
       or not to be ...
    </chapter>
    ...
</xmlbook>
```

```
<xmlbook>
    ...
    <chapter number="1">
       <title>To be...</title>
       <text>or not to be ...<text>
    </chapter>
    ...
</xmlbook>
```

Figure 6-1 From mixed content element to element content element

Before transformation, the mapping would not have been supported because of the element <chapter> and its mixed content. After transformation, we have simple-content elements, element-content elements (elements that contain other elements) but no more mixed-content element, which allows us to create a mapping and decompose the document into relational tables.

Note, however, that transformation is not the solution for all mixed-content elements. As text can appear anywhere when dealing with mixed content, it is not possible to write a DAD for the case that multiple <text> elements can appear anywhere among children of an element node.

You can find more details on how to transform your XML document in 9.1, “Transformation” on page 248.

Decomposition with non-unique attribute and element names

In XML documents, different attributes and elements can have identical names. They appear with the same name but in different contexts. In Example 6-5, the *ID attribute* is used multiple times within the same XML document, but in a different context, with a different meaning. It occurs for the Order, Customer and Salesperson tag. The same is true for the *Name element*. It occurs twice; once under the Customer element and once under the Salesperson element.

This type of non-uniqueness in attribute or element names was not supported with XML Extender in the past. However with FP3 of UDB V8, or FP11 for DB2 UDB V7, or later, this functionality is supported.

You can now decompose documents that contain non-unique attributes and/or non-unique element names that map to different columns (of the same or different tables) without receiving the DXXQ045E error. Example 6-5 shows an XML document with non-unique attributes and non-unique element names.

Example 6-5 XML document with non-unique names

```
<Order ID="0001-6789">
       <!-- Note: attribute name ID is non-unique -->
       <Customer ID = "1111">
                    <Name>John Smith</Name>
       </Customer>
       <!-- Note: element name Name is non_unique -->
       <Salesperson ID = "1234">
              <Name>Jane Doe</Name>
       </Salesperson>
       <OrderDetail>
              <ItemNo>xxxx-xxxx</ItemNo>
              <Quantity>2</Quantity>
              <UnitPrice>12.50</UnitPrice>
       </OrderDetail>
       <OrderDetail>
              <ItemNo>yyyy-yyyy</ItemNo>
              <Quantity>4</Quantity>
              <UnitPrice>24.99</UnitPrice>
       </OrderDetail>
</Order>
```

The accompanying DAD, which maps the duplicate elements/attributes to different columns, looks like Example 6-6 on page 156:

Example 6-6 DAD mapping for non-unique names

```
<element_node name="Order">
  <RDB_node>
    <table name="order_tab" key="order_id"/>
    <table name="detail_tab"/>
    <condition>
      order_tab.order_id = detail_tab.order_id
    </condition>
  </RDB_node>

  <!-- attribute ID duplicated below, but mapped to a different column -->
  <attribute_node name="ID">
    <RDB_node>
      <table name="order_tab" />
      <column name="order_id" type="char(9)"/>
    </RDB_node>
  </attribute_node>

  <element_node name="Customer">
    <!-- attribute ID duplicate, but mapped to a different column -->
    <attribute_node name="ID">
      <RDB_node>
        <table name="order_tab" />
        <column name="cust_id" type="integer"/>
      </RDB_node>
    </attribute_node>

    <!-- element name duplicate, but mapped to a different column -->
    <element_node name="Name">
      <text_node>
        <RDB_node>
          <table name="order_tab" />
          <column name="cust_name" type="char(20)" />
        </RDB_node>
      </text_node>
    </element_node>
  </element_node>

  <element_node name="Salesperson">
    <!-- attribute ID duplicate, but mapped to a different column -->
    <attribute_node name="ID">
      <RDB_node>
      <RDB_node>
        <table name="order_tab" />
        <column name="salesp_id" type="integer"/>
      </RDB_node>
    </attribute_node>
    <!-- element name duplicate, but mapped to a different column -->
    <element_node name="Name">
      <text_node>
        <RDB_node>
          <table name="order_tab" />
          <column name="salesp_name" type="char(20)" />
        </RDB_node>
      </text_node>
    </element_node>
  </element_node>

  <element_node name="OrderDetail" multi_occurrence="YES">
    <element_node name="ItemNo">
      <text_node>
        <RDB_node>
          <table name="detail_tab" />
          <column name="itemno" type="char(9)"/>
        </RDB_node>
      </text_node>
    </element_node>
    <element_node name="Quantity">
      <text_node>
        <RDB_node>
          <table name="detail_tab" />
          <column name="quantity" type="integer"/>
        </RDB_node>
      </text_node>
    </element_node>
    <element_node name="UnitPrice">
      <text_node>
        <RDB_node>detail_tab" />
          <table name="detail_tab" />
          <column name="unit_price" type="decimal(7,2)"/>
        </RDB_node>
      </text_node>
    </element_node>
  </element_node>
</element_node>
```

The contents of the tables look like Example 6-7 on page 158 after the document above is decomposed:

Example 6-7 Table content after decomposition

```
ORDER _TAB:
-----------
ORDER_ID        CUST_ID       CUST_NAME       SALESP_ID       SALESP_NAME
0001-6789       1111          John Smith      1234            Jane Doe


DETAIL_TAB:
-----------
ORDER_ID           ITEMNO         QUANTITY          UNIT_PRICE
0001-6789          xxxx-xxxx      2                 12.50
0001-6789          yyyy-yyyy      4                 24.99
```

> **Note:** Multiple element/attribute mappings to the same column of the same table are not allowed regardless of whether the element/attribute names are different or the same.

Recursion

If the recursion has a fixed length (that is, you know how many levels deep the recursion is), it is possible to write a DAD file for the mapping of your XML documents. For instance, consider an XML document that you want to decompose into a single table. For example, an XML document that has manager/employees information, where an employee can be manager of one or more employees. You have several ways to write an XML document to represent such a hierarchy, for example by using a structure with an element `<employees>` and empty-content subelements `<employee>` with attributes.

```
<employees>
   <employee empid=”0010” name="Bart STEEGMANS"/>
   <employee empid=”0060” name=”Irving STERN” mgrid=”0010”/>
   <employee empid=”0220” name=”Jennifer LUTZ” mgrid=”0060”/>
   <employee empid=”0170” name=”Masatoshi YOSHIMURA” mgrid=”0060”/>
</employees>
```

It is also possible that the XML document uses a recursive way to represent this information. In this case, the element `<employee>` has attributes and element-content, the `<manages>` element which itself has the very same `<employee>` element with attributes.

```
<?xml version="1.0" encoding="UTF-8"?>
<xml>
   <employee name="Bart STEEGMANS" id="0010">
      <manages>
         <employee name="Irving STERN" id="0060">
            <manages>
```

```
               <employee name="Jennifer LUTZ" id="0220"/>
               <employee name=”Masatoshi YOSHIMURA” id=”0170”/>
            </manages>
         </employee>
      </manages>
   </employee>
</xml>
```

Example 6-8 shows a DAD file to handle this type of XML document for shredding purposes.

Example 6-8 An XML document and DAD file with recursion

```
<root_node>
   <element_node name="xml">
   <RDB_node>
      <table name="employee" key="id"/>
      <table name="manager_1line" key="id"/>  <!-- create alias manager_1line for employee -->
      <table name="manager_2line" key="id"/>  <!-- create alias manager_2line for employee -->
      <condition>
         manager_1line.id=employee.managerid AND
         manager_2line.id=manager_1line.managerid
      </condition>
   </RDB_node>
   <element_node name="employee" multi_occurrence="NO">
      <attribute_node name="name">
         <RDB_node>
            <table name="manager_2line"/>
            <column name="name" type="varchar(50)"/>
         </RDB_node>
      </attribute_node>
      <attribute_node name="id">
         <RDB_node>
            <table name="manager_2line"/>
            <column name="id" type="char(10)"/>
         </RDB_node>
      </attribute_node>

      <element_node name="manages" multi_occurrence="NO">
         <element_node name="employee">
            <attribute_node name="name">
               <RDB_node>
                  <table name="manager_1line"/>
                  <column name="name" type="varchar(50)"/>
               </RDB_node>
            </attribute_node>
            <attribute_node name="id">
               <RDB_node>
```

```
                  <table name="manager_1line"/>
                  <column name="id" type="char(10)"/>
               </RDB_node>
            </attribute_node>
            <element_node name="manages" multi_occurrence="NO">
               <element_node name="employee">
                  <attribute_node name="name">
                     <RDB_node>
                        <table name="employee"/>
                        <column name="name" type="varchar(50)"/>
                     </RDB_node>
                  </attribute_node>
                  <attribute_node name="id">
                     <RDB_node>
                        <table name="employee"/>
                        <column name="id" type="char(10)"/>
                     </RDB_node>
                  </attribute_node>
               </element_node> <!-- employee -->
            </element_node>  <!-- manages  -->
         </element_node> <!-- employee -->
      </element_node> <!-- manages -->
   </element_node> <!-- employee -->
   </element_node> <!-- xml -->
</root_node>
```

Important: If fixed length recursion mapping is to be handled, remember that the shredding of "different" elements with the same name into different columns is supported by the XML Extender with FP3 of DB2 UDB V8 and FP11 for DB2 UDB V7.

Note: Unlimited recursion is not supported by DB2 XML Extender for decomposition of XML documents as it cannot be represented by a DAD file.

Also note that a DAD that is written to shred a document with recursion, when used to recompose the document, may produce a document with a slightly different structure.

Mapping to DB2 relational tables

In the following section, we discuss some considerations when shredding XML documents into relational tables using DB2 XML Extender.

UPDATE considerations of shredded data (only INSERT)

Suppose you receive an XML document with sales and customer information to be shredded in to a CUSTOMER and SALES table. If there is no existing record for this customer (new customer), you perform an INSERT on both CUSTOMER and SALES tables, while shredding the XML document. However, if you already have a record for that customer, but you want to update it with a new address for example, you would expect the shredding process to replace the address information (street, city, postcode, ...) in the CUSTOMER table.

The DB2 XML Extender shredding component currently does not support update operations. Shredding always results in inserting a new record in your CUSTOMER table, which will result in an error if you have a primary key specified, for example on customer ID.

To avoid such problem, you can:

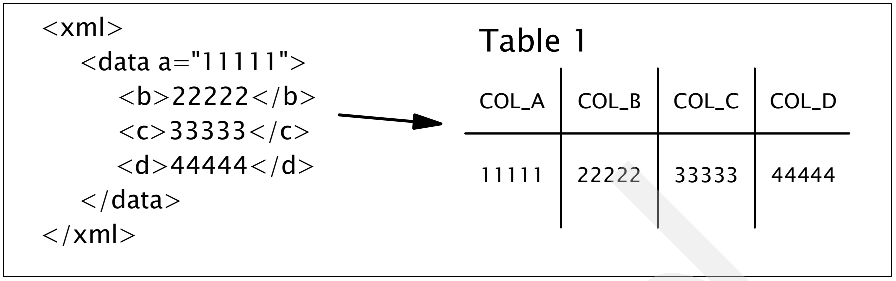
- Transform your XML document prior to performing the actual shredding, so that you split customer information and sales information into two XML documents. You can then shred your sales information, and check in the XML document whether the customer information exists/needs to be updated/is new.
- Use a staging table in DB2 where you shred your XML documents. After populating the staging table, you can update the CUSTOMER table, and INSERT sales information into SALES table, using triggers, or some batch process.
- Write your own application for processing customer information from the XML document, and shred the XML document in your SALES table by mapping only the sales information that you require (see "Decomposing parts of an XML document" on page 169 later in this chapter).

Decomposition of one element or attribute value

- Into different columns of one table

  In an XML document that is to be decomposed, attribute values or element content can populate at most only one column of a table. To have the same data inserted into two columns, the document must have two elements or attributes that contain the same data.

  Figure 6-2 on page 162 shows the decomposition of a very simple XML document into a relational table. One column of data can be generated for the attribute `<a>`, and elements `<b>`, `<c>` and `<d>`.



```
<xml>
   <data a="11111">
      <b>22222</b>
      <c>33333</c>
      <d>44444</d>
   </data>
</xml>
```

Table 1

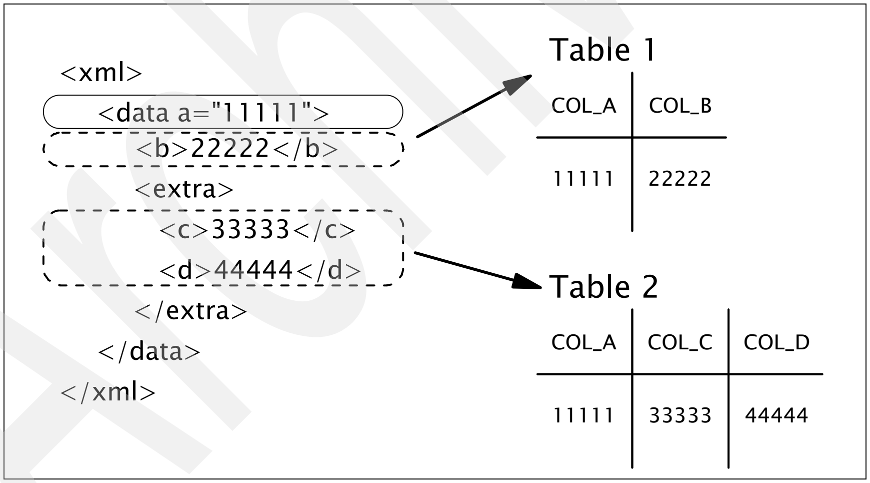
COL_A	COL_B	COL_C	COL_D
11111	22222	33333	44444

Figure 6-2 A simple example of decomposition into a relational table

- Into multiple tables

  The only case where an attribute value or element content can be used to populate more than 1 column of *different tables* is if the XML data is destined for a column that is part of a primary key-foreign key relationship. This is expressed via join conditions in the DAD file.

  To illustrate this, we modify the XML document of the previous example. We want to shred the XML data into two tables, `Table1` and `Table2` like in Figure 6-3.



```
<xml>
   <data a="11111">
      <b>22222</b>
      <extra>
         <c>33333</c>
         <d>44444</d>
      </extra>
   </data>
</xml>
```

Table 1

COL_A	COL_B
11111	22222

Table 2

COL_A	COL_C	COL_D
11111	33333	44444

Figure 6-3 A multi-tables decomposition

To perform such decomposition requires only one `<RDB_node>` mapping for the XML element or attribute involved in the join condition defined in the `<RDB_node>` of the top element in the DAD file. In our case, the attribute `a` is used in the join condition between `Table1` and `Table2`.

The join condition we use for the decomposition is:

```
<condition> Table1.COL_A = Table2.COL_A </condition>
```

The column that is unmapped is automatically populated during decomposition because of the join condition specification. Note that the root element of the subtree that contains all the mappings to the child table (table that contains the unmapped column— `Table 2`) must be an element that is mapped to the parent table (`Table 1`).

The complete DAD file that can be used for the mapping shown in Figure 6-3 on page 162, is provided in Example 6-9 below.

Example 6-9 The DAD file for our multi-tables decomposition example

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DAD SYSTEM "C:\PDB2\SQLLIB\SAMPLES\DB2XML\DTD\dad.dtd">
<DAD>
   <validation>NO</validation>
   <Xcollection>
      <prolog>?xml version="1.0"?</prolog>
      <doctype/>
      <root_node>
         <element_node name="xml">
            <RDB_node>
               <table name="Table1"/>
               <table name="Table2"/>
               <condition>
                  Table1.COL_A=Table2.COL_A                             1
               </condition>
            </RDB_node>
            <element_node name="data">
               <attribute_node name="a">
                  <RDB_node>
                     <table name="Table1"/>
                     <column name="COL_A" type="INTEGER"/>              2
                  </RDB_node>
               </attribute_node>
               <element_node name="b">
                  <text_node>
                     <RDB_node>
                        <table name="Table1"/>
                        <column name="COL_B" type="INTEGER"/>
                     </RDB_node>
                  </text_node>
               </element_node>
               <element_node name="extra" multi_occurrence=”YES”>       3
                  <element_node name="c">
```

```
                     <text_node>
                        <RDB_node>
                           <table name="Table2"/>
                           <column name="COL_C" type="INTEGER"/>
                        </RDB_node>
                     </text_node>
                  </element_node>
                  <element_node name="d">
                     <text_node>
                        <RDB_node>
                           <table name="Table2"/>
                           <column name="COL_D" type="INTEGER"/>
                        </RDB_node>
                     </text_node>
                  </element_node>
               </element_node>
            </element_node>
         </element_node>
      </root_node>
   </Xcollection>
</DAD>
```

The attribute a is specified only for Table1 on line 2, but Table2 will be automatically populated because of the join condition we defined on line 1, for COL_A.
In the next paragraph, we discuss the multi_occurrence attribute that is specified in line annotated 3 of the DAD file).

Multi occurrence with a wrapper element

A wrapper element in a DAD file is an element that:

- Has no attributes or text element
- Has one or more child elements (with or without attributes) that map to the same table.

In the following case, the multi_occurrence setting is mandatory:

Suppose we have the XML document shown in Example 6-10 on page 165, where all elements are mapping the same table Table1, and element <c> and <d> and grouped under the element <data> (<c> and <d> are sibling).

Example 6-10 An XML document with 2 pairs of sibling elements

```
<xml>
   <a>11111</a>
   <b>22222</b>
   <data>
      <c>33333</c>
      <d>44444</d>
   </data>
</xml>
```

We use the RDB_node DAD file shown in Example 6-11 for the mapping of our four elements to the columns in `Table1`.

Example 6-11 RDB_NODE DAD file without multi_occurence=yes

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DAD SYSTEM "C:\PDB2\SQLLIB\SAMPLES\DB2XML\DTD\dad.dtd">
<DAD>
   <validation>NO</validation>
   <Xcollection>
      <prolog>?xml version="1.0"?</prolog>
      <doctype/>
      <root_node>
         <element_node name="xml">
            <RDB_node>
               <table name="Table1"/>
            </RDB_node>
            <element_node name="a">
               <text_node>
                  <RDB_node>
                     <table name="Table1"/>
                     <column name="COL_A" type="INTEGER"/>
                  </RDB_node>
               </text_node>
            </element_node>
            <element_node name="b">
               <text_node>
                  <RDB_node>
                     <table name="Table1"/>
                     <column name="COL_B" type="INTEGER"/>
                  </RDB_node>
               </text_node>
            </element_node>
            <element_node name="data" >
               <element_node name="c">
```

```
                  <text_node>
                     <RDB_node>
                        <table name="Table1"/>
                        <column name="COL_C" type="INTEGER"/>
                     </RDB_node>
                  </text_node>
               </element_node>
               <element_node name="d">
                  <text_node>
                     <RDB_node>
                        <table name="Table1"/>
                        <column name="COL_D" type="INTEGER"/>
                     </RDB_node>
                  </text_node>
               </element_node>
            </element_node>
         </element_node>
      </root_node>
   </Xcollection>
</DAD>
```

When shredding using the DAD file above, the XML document populates `Table1` with several incomplete rows, as illustrated in Figure 6-4.

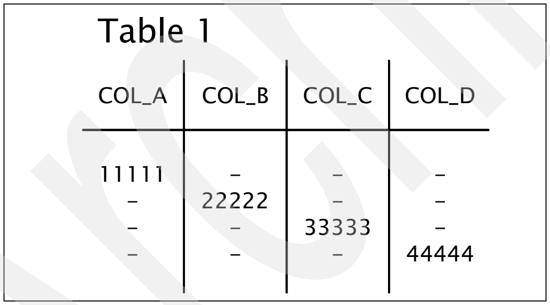


Table 1

COL_A	COL_B	COL_C	COL_D
11111	-	-	-
-	22222	-	-
-	-	33333	-
-	-	-	44444

Figure 6-4 Shredded without multi_ocurrence setting

When we insert a multi_occurrence attribute in the * line in Example 6-11 on page 165,

```
<element_node name="xml" multi_occurrence="YES">
```

this setting will ensure that element nodes that map to the same table populate the same row. The table populated with the updated DAD file now looks like Figure 6-5 on page 167.

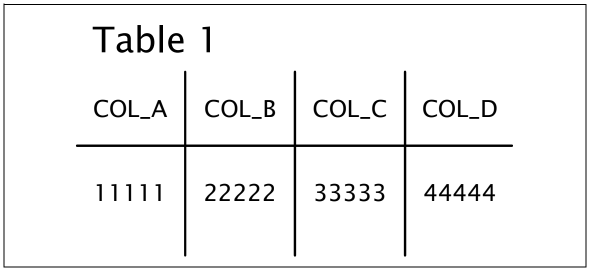


Table 1

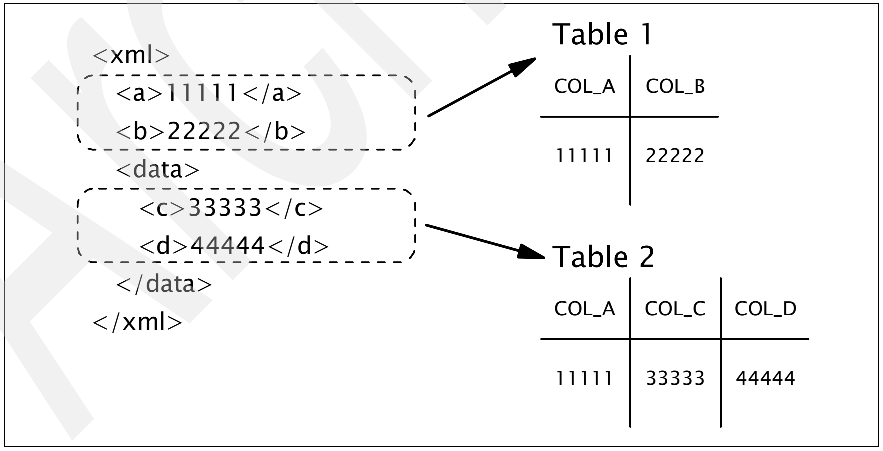
COL_A	COL_B	COL_C	COL_D
11111	22222	33333	44444

Figure 6-5 Shredded with multi_occurrence="YES"

We illustrate the usage of multi_occurrence with a another example. We want to decompose the same XML document as described in Example 6-10 on page 165, but this time, we map elements <c> and <d> to a second table Table2. `COL_A` is used as foreign key by Table2. As a basis, we use the DAD file from the example above, and modify it as follows:

- Specify `multi_occurrence=“YES“` in element_node `<xml>` and `<data>`. This will ensure that both tables are populated with one single row.
- Add Table2 to the root `<RDB_node>`.
- Add a join condition `Table1.COL_A=Table2.COL_A` to the root `<RDB_node>`.
- Map elements `<c>` and `<d>` to Table2. (Because `COL_A` of `Table2` is part of the join condition, it does not need to be mapped. This column will be populated automatically during decomposition.)

The populated tables Table1 and Table2 are illustrated in Figure 6-6.



```
<xml>
  <a>11111</a>
  <b>22222</b>
  <data>
    <c>33333</c>
    <d>44444</d>
  </data>
</xml>
```

Table 1

COL_A	COL_B
11111	22222

Table 2

COL_A	COL_C	COL_D
11111	33333	44444

Figure 6-6 Decomposition into multiple tables

The complete DAD file to perform such a decomposition is shown in Example 6-12 on page 168.

Example 6-12 DAD for decomposition into multiple tables

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DAD SYSTEM "C:\DB2\SQLLIB\SAMPLES\DB2XML\DTD\dad.dtd">
<DAD>
   <validation>NO</validation>
   <Xcollection>
      <prolog>?xml version="1.0"?</prolog>
      <doctype/>
      <root_node>
         <element_node name="xml" multi_occurrence="YES">
            <RDB_node>
               <table name="Table1"/>
               <table name="Table2"/>
               <condition>Table1.COL_A=Table2.COL_A</condition>
            </RDB_node>
            <element_node name="a">
               <text_node>
                  <RDB_node>
                     <table name="Table1"/>
                     <column name="COL_A" type="INTEGER"/>
                  </RDB_node>
               </text_node>
            </element_node>
            <element_node name="b">
               <text_node>
                  <RDB_node>
                     <table name="Table1"/>
                     <column name="COL_B" type="INTEGER"/>
                  </RDB_node>
               </text_node>
            </element_node>
            <element_node name="data"  multi_occurrence="YES">
               <element_node name="c">
                  <text_node>
                     <RDB_node>
                        <table name="Table2"/>
                        <column name="COL_C" type="INTEGER"/>
                     </RDB_node>
                  </text_node>
               </element_node>
               <element_node name="d">
                  <text_node>
                     <RDB_node>
                        <table name="Table2"/>
                        <column name="COL_D" type="INTEGER"/>
                     </RDB_node>
                  </text_node>
               </element_node>
            </element_node>
         </element_node>
      </root_node>
   </Xcollection>
</DAD>
```

Decomposing parts of an XML document

DB2 XML Extender supports the existence of unmapped elements in a DAD file. This means that if you have a set of elements or attributes in your XML document, but only half of them are defined in the mapping of your tables to be populated, XML Extender does not generate any error. It just ignores the unmapped elements and attributes.

The opposite is also possible. You do not have to specify all the columns of the tables that you shred into, in the DAD file. However, this implies that the column in the populated relational table, is defined without the NOT NULL option. If the column is defined with the NOT NULL option, the element must be specified in the XML document, as well as in the DAD file.

For example, we have the following XML document shown in Example 6-13:

Example 6-13 Sample XML document for NOT NULL column shredding

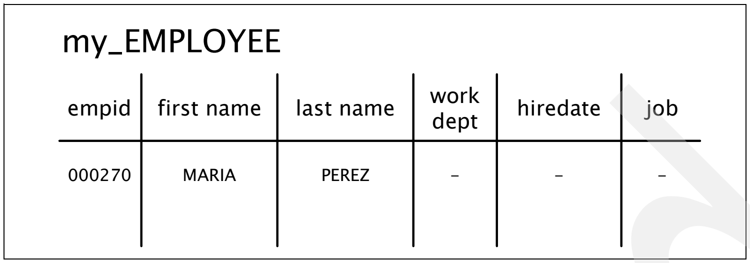
```
   <xml>
      <employee id="000270">
         <firstname>MARIA</firstname>
         <lastname>PEREZ</lastname>
         <workdept>D21</workdept>
         <hiredate>1980-09-30</hiredate>
         <job>CLERK</job>
      </employee>
   </xml>
```

We want to map only the attribute `id` and the elements `<firstname>` and `<lastname>`, ignoring the rest of the document. The consequent DAD file is shown in Example 6-14 on page 170.

Example 6-14 DAD for partial mapping

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DAD SYSTEM "C:\DB2\SQLLIB\SAMPLES\DB2XML\DTD\dad.dtd">
<DAD>
   <validation>NO</validation>
   <Xcollection>
      <prolog>?xml version="1.0"?</prolog>
      <doctype/>
      <root_node>
         <element_node name="xml">
            <RDB_node>
               <table name="my_EMPLOYEE"/>
            </RDB_node>
            <element_node name="employee">
               <attribute_node name="id">
                  <RDB_node>
                     <table name="my_EMPLOYEE"/>
                     <column name="empid" type="CHAR(6)"/>
                  </RDB_node>
               </attribute_node>
               <element_node name="firstname">
                  <text_node>
                     <RDB_node>
                        <table name="my_EMPLOYEE"/>
                        <column name="firstname" type="CHAR(15)"/>
                     </RDB_node>
                  </text_node>
               </element_node>
               <element_node name="lastname">
                  <text_node>
                     <RDB_node>
                        <table name="Table2"/>
                        <column name="lastname" type="CHAR(20)"/>
                     </RDB_node>
                  </text_node>
               </element_node>
            </element_node>
         </element_node>
      </root_node>
   </Xcollection>
</DAD>
```

Decomposing the XML document using the DAD above, results in populating the table my_EMPLOYEE as illustrated in Figure 6-7 on page 171. Only the elements defined in the DAD file are decomposed, and no error is generated by the XML Extender.



my_EMPLOYEE

empid	first name	last name	work dept	hiredate	job
000270	MARIA	PEREZ	–	–	–

Figure 6-7 Result of partial DAD mapping

About OrderBy

The *orderby* attribute can be specified in the root element RDB_node, as an attribute of a table (table definition). It allows to define the name of the columns that determine the sequence order of multi-occurring text or attribute values when generating (*composing, publishing*) XML documents.

Whether or not the *orderby* attribute is set in the DAD file has *no impact on the shredding activity*, as XML Extender is ignoring it at decomposition time.

Features to consider when shredding

The following sections describes another set of shredding considerations.

Validation against XML Schema

XML Schema is an XML-based alternative to a DTD, to define the specifications of the content of XML documents. The XML Schema uses the XML format to define the element and attribute names within an XML document, and defines the type of content the elements and attributes are allowed to contain.

DTDs are easier to code and validate than XML Schemas. However, there are several advantages in using an XML Schema, which are listed below:

- XML Schemas are valid XML documents that can be processed by tools such as the XSD Editor in WebSphere Studio Application Developer (WSAD).
- XML Schemas are more powerful than DTDs. Everything that can be defined by DTD can also be defined by XML Schemas, but not vice versa.
- XML Schemas support a set of data types, similar to the ones used in most common programming languages, and provide the ability to create additional types. You can constrain the document content to the appropriate type. For example, you can replicate the properties of fields found in DB2.

- XML Schemas support regular expressions to set constraints on character data, which is not possible if you use a DTD.
- XML Schemas provide better support for XML namespaces, which enable you to validate documents that use multiple namespaces, and to reuse constructs from schemas already defined in different namespaces.
- XML Schemas provide better support for modularity and reuse with include and import elements.
- XML Schemas support inheritance for element, attribute and data type definitions.

Example 6-15 shows a simple XML document containing employee information such as employee-id, name, phone numbers, working department and position.

Example 6-15 Simple XML document

```
<xmldata>
   <employee empid=”8310”>
      <name>
         <firstname>Olivier</firstname>
         <lastname>Guyennet</lastname>
      </name>
      <phone type=”Office”>555-12345</phone>
      <phone type=”Portable”>555-56789</phone>
      <job dept=”ITSO”>Clerk</job>
   </employee>
</xmldata>
```

The DTD document in Example 6-16 allows validation of the XML document above.

Example 6-16 A simple DTD

```
<!ELEMENT xmldata (employee+)>
   <!ELEMENT employee (name,phone+,(job|position)?)>1
      <!ATTLIST employee empid CDATA #REQUIRED>
      <!ELEMENT name (firstname,lastname)>
         <!ELEMENT firstname CDATA #REQUIRED>
         <!ELEMENT lastname CDATA #REQUIRED>
      <!ELEMENT phone CDATA #REQUIRED>
         <!ATTLIST phone type CDATA #IMPLIES>
      <!ELEMENT job CDATA #REQUIRED>
         <!ATTLIST job dept CDATA #REQUIRED>
      <!ELEMENT position CDATA #REQUIRED>
         <!ATTLIST position experience CDATA #REQUIRED>
```

In the DTD above, some key information is defined in line 1 :

```
<!ELEMENT employee (name,phone+,(job|position)?)>
```

The key information is:

- The '+' symbol after phone, indicating that the child element `<phone>` must appear one or more time inside the `<employee>` element.
- (job | position) declares that the `<employee>` element must contain either a `<job>` or a `<position>` child element.
- The question mark (?) after the alternative statement `(job|position)` specifies that one of the two eventual child elements can occur zero or one time inside the `<employee>` element.

The XML Schema for the same XML document is shown in Example 6-17, which is easier to understand than the equivalent DTD.

Example 6-17 Simple XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<!-- definition of simple elements -->
<xs:element name="firstname" type="xs:string"/>
<xs:element name="lastname" type="xs:string"/>

<!-- definition of attributes -->
<xs:attribute name="empid" type="xs:string"/>
<xs:attribute name="type" type="xs:string"/>
<xs:attribute name="dept" type="xs:string"/>

<!-- definition of complex elements -->
<xs:element name="name">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="firstname"/>
         <xs:element ref="lastname"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="phone">
   <xs:complexType>
      <xs:SimpleContent>
         <xs:extension base="xs:string">
            <xs:attribute ref="type"/>
         </xs:extension>
      </xs:SimpleContent>
   </xs:complexType>
</xs:element>
<xs:element name="job">
   <xs:complexType>
      <xs:SimpleContent>
         <xs:extension base="xs:string">
            <xs:attribute ref="dept"/>
         </xs:extension>
      </xs:SimpleContent>
   </xs:complexType>
</xs:element>

<xs:element name="employee">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="name"/>
         <xs:element ref="phone" minoccurs="1" maxOccurs="unbounded"/>
         <xs:element ref="job" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute ref="empid" use="required"/>
   </xs:complexType>
</xs:element>

<xs:element name="xmldata">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="employee" maxOccurs="1"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>

</xs:schema>
```

DB2 XML Extender can use the **SVALIDATE()** UDF to validate XML documents against an XML Schema. See 9.2.1, “Validating XML documents using UDFs” on page 259 for more details on DB2 XML Extender’s validation capabilities.

Commit control

When running XML Extender UDFs, be aware of the following commitment control guidelines:

- XML Extender assumes that the application handles COMMIT and ROLLBACK, and so, the stored procedures never perform these actions.
- It is recommended that application design include all XML Extender UDF, XML Extender stored procedures, and any INSERT, UPDATE, or DELETE

which use XML Extender triggers, within one commitment control definition. This commit definition should be completed by your application with the appropriate COMMIT or ROLLBACK.
- Because XML Extender UDFs require isolation level CS (Cursor Stability), it is required that the application explicitly performs a COMMIT before performing another SQL action on the same row that has already been updated by an XML Extender UDF or another SQL query.

6.2.3 Shredding: configuration and execution

The following sections address considerations about the configuration and execution of shredding.

Enabling an XML collection

When you enable an XML collection, DB2 XML Extender performs the following tasks:

- It creates an XML collection usage entry in the XML_USAGE table.
- It creates the tables to store the XML collection, specified in the DAD file, in case the tables do not already exist in the database. However, to have better control over where the tables are created (for example in which table space), and to avoid potential problems with complex RI structures for example, it is best to create the tables manually, before you enable the collection.
- When you enable an XML collection, the DAD file is parsed to map the tables and columns related to the XML document and this information is stored in the XML_USAGE table.

To enable an XML collection from the DB2 command window, issue the following command:

```
dxxadm enable_collection dbName collection DAD_file -t tablespace
```

In this command:

- *dbName* is the name of the database.
- *collection* is the name of the XML collection. This value is used as a parameter for the stored procedures that can operate on an XML collection.
- *DAD_file* is the name of the file that contains the Document Access Definition (DAD).
- *Tablespace* specifies an existing table space that is to be used to create the new DB2 tables that are created for decomposition. If not specified, the default table space will be used.

Note that the database itself has to be enabled for XML before you can enable a collection.

Executing XML Extender decomposition stored procedures

The decomposition stored procedures provided by XML Extender are *dxxInsertXML()* and *dxxShredXML()*. They are used to break down or shred incoming XML documents, and to store data in relational tables.

- The dxxShredXML() stored procedure takes a DAD file as input. It does not require an enabled XML collection.
- The dxxInsertXML() stored procedure takes an enabled XML collection name as input. It therefore obviously requires you to enable the collection first.

dxxShredXML()

This stored procedure can be used by applications that do occasional updates or applications that do not want the overhead of administering the XML data. The stored procedure dxxShredXML() does not required an enabled collection. It uses a DAD file as input parameter instead.

The stored procedure dxxShredXML() takes two input parameters:

- A DAD file and
- The XML document that is to be decomposed

It returns two output parameters:

- A return code and
- A return message

The dxxShredXML() stored procedure decomposes the XML document, and inserts the data into a set of relational tables according to the `<Xcollection>` specification in the input DAD file.
The tables that are specified in the `<Xcollection>` of the DAD file must exist at the time the document is shredded, and the columns are assumed to meet the data types specified in the DAD mapping. If this is not true, an error message is returned.

The definition of the dxxShredXML stored procedure is as follows:

```
dxxShredXML( CLOB(100K)      DAD,         /*input */
             CLOB(1M)        xmlobj,      /*input */
             long            returnCode,  /*output */
             varchar(1024)   returnMsg)   /*output */
```

The stored procedure dxxShredXML() does not require a primary-foreign key relationship among joining tables.

To make the use of the stored procedure somewhat easier, DB2 comes with a sample client program `dxxshrd`. This program will invoke the dxxShredXML() stored procedure. You can execute the program, providing a database name, the name of the DAD file, and the XML document that you want to shred, as follows:

```
dxxshrd mydb C:\Progra~1\IBM\SQLLIB\samples\db2xml\dad\neworder2.dad
        C:\Progra~1\IBM\SQLLIB\samples\db2xml\xml\neworder2.xml
```

There is also a sample Java program shipped with DB2 XML Extender which can be used to decompose to XML document. The source is located under the samples\db2xml\jdbc\ directory where the XML Extender is installed, as shred.java.

dxxInsertXML()

The stored procedure dxxInsertXML() works the same as dxxShredXML(), except that dxxInsertXML() takes an enabled XML collection as its first input parameter. The stored procedure dxxInsertXML() inserts data from an XML document into an enabled XML collection, which is associated with a DAD file. That DAD file contains specifications of the XML collection tables and the mapping between the XML document and the relational tables. The XML collection tables are checked according to the specifications in the <Xcollection>. The stored procedure `dxxInsertXML()` then decomposes the XML document according to the mapping, and inserts the data into the tables of the named XML collection.

The declaration for the dxxInsertXML stored procedure follows:

```
dxxInsertXML(char(UDB_SIZE )    collectionName, /*input */
             CLOB(1M)           xmlobj,         /*input */
             long               returnCode,     /*output */
             varchar(1024)      returnMsg)      /*output */
```

Again, to make the use of the stored procedure somewhat easier, DB2 comes with a sample client program `dxxisrt`. This program invokes the dxxInsertXML() stored procedure. You can execute the program, providing a database name, the name of the XML collection, and the XML document that you want to shred, as follows:

```
dxxisrt mydb abc C:\Progra~1\IBM\SQLLIB\samples\db2xml\xml\neworder1.xml
```

For more details about the dxxShredXML() and dxxInsertXML() stored procedures, please refer to *DB2 XML Extender Administration and Programming V8*, SC27-1234, also available online at:

http://www.ibm.com/software/data/db2/library

6.3 Alternatives to shredding with DB2 XML Extender

In this section, we look at a few other techniques to decompose XML documents into a relational model.

6.3.1 Using the XML wrapper

One of the non-relational wrappers provided by DB2 Information Integrator V8.1 is the XML wrapper. It enables you to give a representation of XML data as if the information contained in the XML document was stored in relational tables, without the constraint of actually having to import or load the data.

Refer to Chapter 11, “XML wrapper” for more information and details on the XML wrapper creation and usage.

Consider a simple XML document stored in a file system, or (as is) in a column of a relation table. The location is not the point here, but rather, the consideration that the data in the XML document can be mapped and made to look like a relational table (or usually a set of relational tables).

Using the Cross Loader utility provided by DB2 allows you to LOAD data from one relational table or nickname to another table. In that way, you can keep an intact copy of your XML documents, while also providing a simulated document shred into relational tables through the XML wrapper.
You can map a single XML document to a nickname (a virtual table of (part of) your XML document), an entire file directory, a URI or a column of a (DB2) table.

However, you must keep in mind that every time you access the XML document through a nickname using a SELECT statement (insert, update, delete are not supported for XML files), the entire XML document has to be parsed. The bigger your XML documents are, the heavier parsing becomes, both in terms of time and memory requirements.

Important: Keep in mind that if your XML wrapper nickname references an XML document stored in a column of a DB2 table, the table containing that column must be located in the same database.

6.3.2 Writing your own code

Instead of using DB2 XML Extender to shred XML documents, you can of course also write your own programs to do so. In this section, we briefly describe two interfaces to manipulate XML documents, SAX and DOM.

Shredding simple documents with SAX

At the heart of every XML application is an XML processor that parses an XML document, so that the document elements or attributes can be retrieved and transformed into a presentation understood by the target client. The other responsibility of the parser is to check the syntax and structure of the XML document.

SAX is the Simple API for XML, originally a Java-only API. SAX was the first widely adopted API for XML in Java. SAX APIs are event-based APIs, which report parsing events (such as start and end elements) directly to the application through callbacks, and does not usually build an internal tree. These event-driven APIs are used for accessing XML documents and extracting information from them.

They cannot be used to manipulate the internal structures of XML documents. As the XML document is parsed, the application using SAX receives information about the various parsing events. The application implements handlers to deal with these different events, much like handling events in a graphical user interface. The logical structure of an application using the SAX API as a parser is shown in Figure 6-8.

SAX parses XML documents by event, so there is no need to cache the entire document in memory or secondary storage. You can parse documents that are much larger than your available system memory.

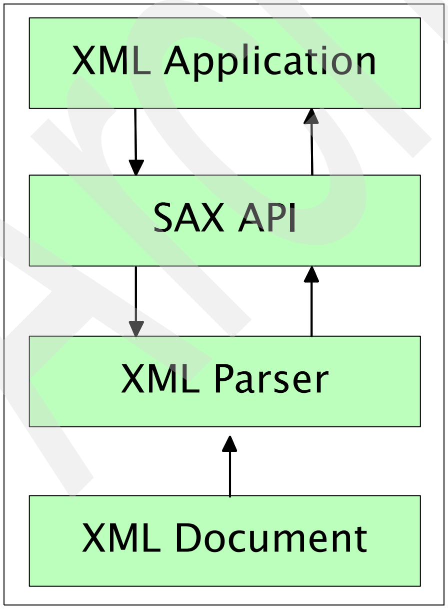


Figure 6-8 Logical structure of an application using SAX

Using the Document Object Model (DOM)

The Document Object Model (DOM) defines a set of interfaces to access tree-structured XML documents. DOM specifies how XML and HTML documents can be represented as objects. Unlike SAX, DOM allows creating and manipulating the contents of XML documents.

DOM provides a set of standard object interfaces that an XML parser can use to expose the contents of a document to a client application. These interfaces provide access to all the information from the original document, organized in a hierarchical tree structure. The base interface for navigating this tree structure is the Node interface that defines the necessary methods to navigate and manipulate the tree structure of XML documents. The methods include getting, deleting, and modifying the children of a node, as well as inserting new children to it.

The sample structure shown in Figure 6-9 can be traversed using the parent, child and sibling links available through the *node interface*.

Document represents the complete document. The interface defines methods for creating elements, attributes, comments, and so on. Attributes of a node are manipulated using the methods of the *element interface*.

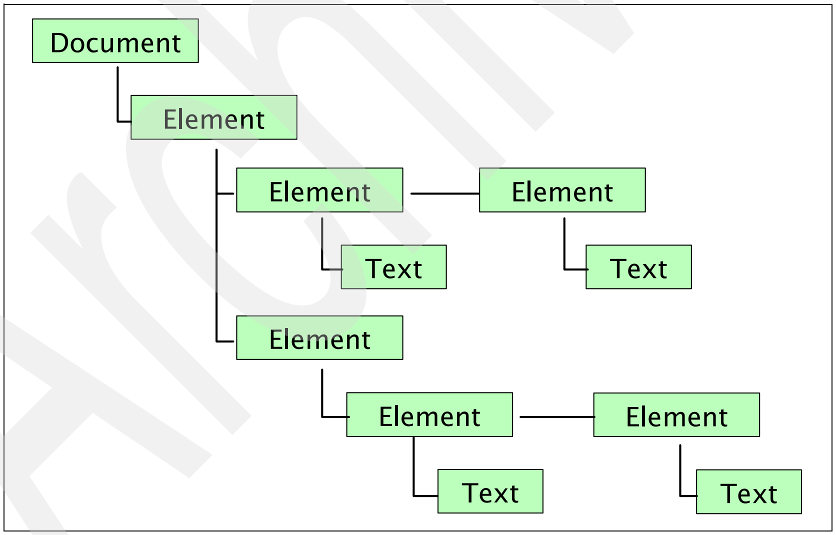


Figure 6-9 Example structure generated DOM tree

SAX versus DOM

Unlike SAX, a DOM parser requires the entire document to be copied into memory for processing. This is usually not a problem with small XML documents, but when it comes to larger files, memory requirement might become a problem. In the worst case, the SAX-based tool buffers all of the data in a document. On the other hand, at best, all of the DOM-based tools buffer a DOM tree plus one row of data. Because the DOM tree contains all of the data, DOM-based tools necessarily use more memory. How much more memory depends on the average size of the data per node. A DOM level 1 node contains 9 pointers to other nodes or, at 4 bytes per pointer, 36 bytes of pointers per node. It also contains other information such as the name and type of the node. If the data is large with respect to the node overhead (for example, 1000 bytes per node) then the size of the DOM tree is roughly the size of the data in that tree. If the data is small with respect to the node overhead (for example, 10 bytes per node, which is more common in data-centric applications) then the size of the DOM tree can be several times the size of the data.

With respect to speed, the SAX parser tends to show better performance than the DOM parser. The SAX API can provide faster and less costly processing of XML data when you do not need to access all of the data in an XML document. SAX-based tools visit each node only once. Furthermore, the SAX-based tools do not need to spend time building a DOM tree and also traverses the document more quickly, mainly because traversing the document is part of parsing and does not require any extra method calls.

When using the object-based mapping to transfer data from XML documents to the database, SAX-based tools are always faster and use less memory than DOM-based tools. However, you should keep in mind that SAX is better used with simple structured XML documents.

It is worth noting that XML documents used by the SAX-based tool can be "tuned," that is, they can be designed to minimize the amount of data that needs to be buffered. To do this, each set of siblings must be arranged in the following order: primary key elements, data elements, foreign key elements. When this is done, the SAX-based tool buffers only one row of data at a time and scales to arbitrarily large documents.

Example of shredding using SAX

The code to transfer data from an XML document to the database follows a common pattern, regardless of whether it uses SAX or DOM:

- Table element start: Prepare an INSERT statement
- Row element start: Clear INSERT statement parameters
- Column elements: Buffer PCDATA and set INSERT statement parameters
- Row element end: Execute INSERT statement
- Table element end: Close INSERT statement

The code does not make any assumptions about the names of the tags. In fact, it uses the name of the table-level tag to build the INSERT statement and the names of the column-level tags to identify parameters in the INSERT statement. Thus, these names could correspond exactly to the names in the database or could be mapped to names in the database using a configuration file.

SAX reads a document in a single pass, in depth-first, width-second order. This means that one or more rows of data will have to be buffered while processing documents. How many rows must be buffered depends on how the document is mapped to the database and the intelligence of the transfer tool.

Example 6-18 shows a code sample using SAX for a document containing a single table:

Example 6-18 Sample SAX program

```
int state = UNKNOWN;
   PreparedStatement stmt;
   StringBuffer data;

   public void startElement(String uri, String name, String qName,
                            Attributes attr) {
      if (state == UNKNOWN) {
         stmt = getInsertStmt(name);
         state = TABLE;
      }
      else if (state == TABLE) {
         state = ROW;
         stmt.clearParameters();
      } else if (state == ROW) {
         state = COLUMN;
         data = new StringBuffer();
      } else { // if (state == COLUMN)
         throw new SAXException("XML document nested too deep.");
      }
   }

   public void characters (char[] chars, int start, int length) {
      if (state == COLUMN)
         data.append(chars, start, length);
   }

   public void endElement(String uri, String name, String qName) {
      if (state == TABLE) {
         stmt.close();
         state = UNKNOWN;
      }
      else if (state == ROW) {
         stmt.executeUpdate();
         state = TABLE;
      } else if (state == COLUMN) {
         setParameter(stmt, name, data.toString());
         state = ROW;
      } else { // if (state == UNKNOWN)
```

```
            throw new SAXException("Invalid program state.");
        }
    }
```

6.4 Shredding with XML Extender: a step-by-step example

This section provides a very simple scenario that demonstrates the steps to go through when you want to shred an XML document into a set of relational tables using DB2 XML Extender.
We assume that your database has not yet been enabled for XML Extender. We use the same XML document used in 4.4, “Storing intact XML documents with XML Extender” on page 101. In this example, we decompose the XML data into two new relational tables; XPROJECT and XEMPLOYEE, as shown in Figure 6-10. To allow decomposition into multiple tables, we have to make modifications to the DAD file, such as adding a wrapper element `<participant>` for the `<employee>` element. This modification implies a modification of the DTD file and the XML document.

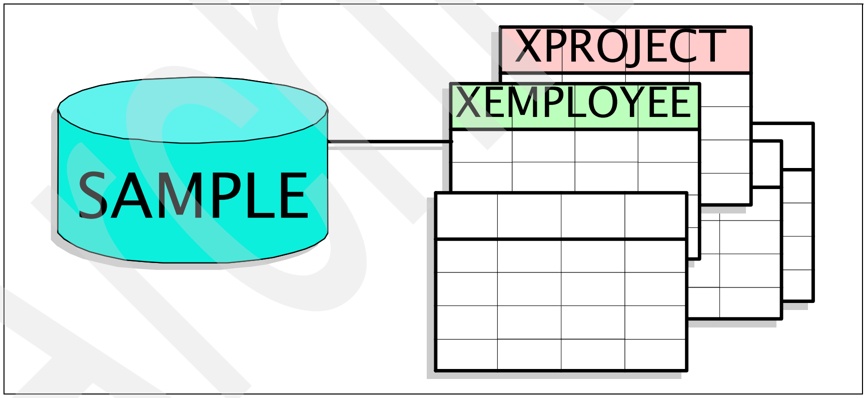


Figure 6-10 The SAMPLE database used in our example

The DDL for constructing our tables XEMPLOYEE and XPROJECT is shown in the Example 6-19 on page 185.

Example 6-19 The DDL for XPROJECT and XEMPLOYEE

```
------------------------------------------------
-- DDL Statements for table "XPROJECT"
------------------------------------------------

 CREATE TABLE "XPROJECT"  (
        "PROJID" SMALLINT NOT NULL ,
        "PROJDESC" CHAR(50) ,
        "PROJSTART" DATE )
       IN "USERSPACE1" ;

-- DDL Statements for primary key on Table "XPROJECT"

ALTER TABLE "XPROJECT"
   ADD PRIMARY KEY
      ("PROJID");


------------------------------------------------
-- DDL Statements for table "XEMPLOYEE"
------------------------------------------------

 CREATE TABLE "XEMPLOYEE"  (
        "EMPID" CHAR(4) NOT NULL ,
        "DEPT" CHAR(3) ,
        "FIRSTNME" CHAR(15) ,
        "LASTNAME" CHAR(15) ,
        "PROJID" SMALLINT )
       IN "USERSPACE1" ;

-- DDL Statements for primary key on Table "XEMPLOYEE"

ALTER TABLE "XEMPLOYEE"
   ADD PRIMARY KEY
      ("EMPID");
```

6.4.1 Step 1 - XML enabling of your database

Enable the `SAMPLE` database to XML using the `dxxadm` command from a DB2 command window:

```
dxxadm enable_db MYXMLDB
```

6.4.2 Step 2 - Creating the DAD file

When you enable an XML collection, the DAD file specified in the command is parsed to identify the tables and columns related to the XML document. This information is stored in the XML_USAGE table. So before enabling the XML collection, you have to design your DAD file.

The XML, DTD and DAD files that we used are shown in Example 6-20, Example 6-21 and Example 6-22 on page 187, respectively.

Example 6-20 XML document: Project.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE data SYSTEM "C:\CHAP6\project.dtd">
<data>
   <project>
      <projectID id="1234"/>
      <description>Writing a Redbook</description>
      <startdate>2003-04-28</startdate>
      <participant>
         <employee>
            <employeeID empid="2300"/>
            <department>D01</department>
            <firstname>Olivier</firstname>
            <lastname>Guyennet</lastname>
         </employee>
         <employee>
            <employeeID empid="5090"/>
            <department>B05</department>
            <firstname>Stephen</firstname>
            <lastname>Priestley</lastname>
         </employee>
      </participant>
   </project>
</data>
```

The DTD file Project.dtd is shown next.

Example 6-21 DTD file: Project.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT data (project)>
<!ELEMENT project (projectID, description, startdate, participant)>
<!ELEMENT projectID EMPTY>
<!ELEMENT description (#PCDATA)>
<!ELEMENT startdate (#PCDATA)>
```

```
<!ATTLIST projectID id CDATA #REQUIRED>

<!ELEMENT participant (employee*)>

<!ELEMENT employee (employeeID, department, firstname, lastname)>
<!ELEMENT employeeID EMPTY>
<!ELEMENT department (#PCDATA)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ATTLIST employeeID empid CDATA #REQUIRED>
```

Here, you find the DAD file that we used.

Example 6-22 DAD file: Project.dad

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DAD SYSTEM "c:\pdb2\sqllib\samples\db2xml\dtd\dad.dtd">
<DAD>
   <validation>NO</validation>
   <Xcollection>
      <prolog>?xml version="1.0"?</prolog>
      <doctype/>
      <root_node>
         <element_node name="data">
            <RDB_node>
               <table name="XPROJECT"/>
               <table name="XEMPLOYEE"/>
               <condition>XPROJECT.projid=XEMPLOYEE.projid</condition>
            </RDB_node>
            <element_node name="project" multi_occurrence="YES">
               <element_node name="projectID">
                  <attribute_node name="id">
                     <RDB_node>
                        <table name="XPROJECT"/>
                        <column name="projid" type="smallint"/>
                     </RDB_node>
                  </attribute_node>
               </element_node>
               <element_node name="description">
                  <text_node>
                     <RDB_node>
                        <table name="XPROJECT"/>
                        <column name="projdesc" type="char(50)"/>
                     </RDB_node>
                  </text_node>
               </element_node>
               <element_node name="startdate">
                  <text_node>
                     <RDB_node>
                        <table name="XPROJECT"/>
                        <column name="projstart" type="date"/>
                     </RDB_node>
                  </text_node>
               </element_node>

               <element_node name="participant">
                  <element_node name="employee" multi_occurrence="YES">
                     <element_node name="employeeID">
                        <attribute_node name="empid">
                           <RDB_node>
                              <table name="XEMPLOYEE"/>
                              <column name="empid" type="char(4)"/>
                           </RDB_node>
                        </attribute_node>
                     </element_node>
                     <element_node name="department">
                        <text_node>
                           <RDB_node>
                              <table name="XEMPLOYEE"/>
                              <column name="dept" type="char(3)"/>
                           </RDB_node>
                        </text_node>
                     </element_node>
                     <element_node name="firstname">
                        <text_node>
                           <RDB_node>
                              <table name="XEMPLOYEE"/>
                              <column name="firstnme" type="char(15)"/>
                           </RDB_node>
                        </text_node>
                     </element_node>
                     <element_node name="lastname">
                        <text_node>
                           <RDB_node>
                              <table name="XEMPLOYEE"/>
                              <column name="lastname" type="char(15)"/>
                           </RDB_node>
                        </text_node>
                     </element_node>
                  </element_node>
               </element_node>
```

```
            </element_node>
         </element_node>
      </root_node>
   </Xcollection>
</DAD>
```

6.4.3 Step 3 - Enablement of the XML collection

The command to enable our XML collection is:

```
dxxadm enable_collection MYXMLDB mycollection project.dad
```

6.4.4 Step 4 - Decomposing the XML data into relational tables

We use the shred utility `dxxshrd` provided by DB2 XML Extender. The command requires you to specify the database name, the DAD file name and an XML document name:

```
dxxshrd MYXMLDB project.dad project.xml
```

Since we are using an XML collection, we can also use the `dxxisrt` program:

```
dxxisrt MYXMLDB mycollection project.xml
```

6.4.5 Step 5 - Checking the result using the SELECT statement

The result of the decomposition we just performed on our XML document can be seen by running a SELECT statement against both the XPROJECT and XEMPLOYEE table. The result is shown in Example 6-23 on page 190.

Example 6-23 Result of the shredding operation

```
db2 SELECT * FROM XPROJECT

PROJID PROJDESC                                           PROJSTART
------ -------------------------------------------------- ----------
  1234 Writing a Redbook                                  2003-04-28

  1 record(s) selected.

db2 SELECT * FROM XEMPLOYEE

EMPID DEPT FIRSTNME        LASTNAME        PROJID
----- ---- --------------- --------------- ------
2300  D01  Olivier         Guyennet          1234
5090  B05  Stephen         Priestley         1234

  2 record(s) selected.
```

Bulk processing of XML documents

This chapter describes two ways to process XML documents in bulk:

- Using an XML cutter to process repetitive documents
- Using XSLT and load / import for bulk inserts

7.1 An XML cutter to process repetitive documents

Many data-centric XML documents are actually composed of many smaller documents. The document in Example 7-1 contains many sales orders documents.

> **Note:** These smaller documents are XML documents in a limited sense. In particular, they start with a root element and do not have a prolog (a *prolog* precedes the root element in an XML document. It may contain an XML declaration, a DOCTYPE declaration, comments, processing instructions, and whitespace).

Example 7-1 Sample XML document

```
<SalesOrders>
   <SalesOrder>
      ...
   </SalesOrder>
   <SalesOrder>
         ...
   </SalesOrder>
      ...
   <SalesOrder>
         ...
   </SalesOrder>
</SalesOrders>
```

Such documents are common in bulk-loading situations, such as when replicating the contents of a database. What is important about the structure of these documents is that a single element type defines the start of each sub-document. This makes it easy to cut each document into many smaller documents, each of which has this element type as its root. For example, the sub-documents in the preceding example each start with the SalesOrder element. Those elements outside the sub-documents are ignored.

There are a number of reasons for cutting repetitive documents into sub-documents. For example:

- You want to store each sub-document in an XML column.
- You want to use separate transactions to shred each sub-document into a set of tables. (When you shred a document with DB2 XML Extender, the entire shredding operation takes place in a single transaction. If an error occurs

while inserting any of the data, the entire transaction must be rolled back and none of the data will be inserted.)

- You want to transform each sub-document, but XSLT runs out of memory when trying to process the entire document.

In the following sections, we describe a sample tool for cutting XML documents. A formal cutting tool from IBM may be made available at a future date.

7.1.1 The SAXCutter sample tool

SAXCutter is a sample tool written in Java for cutting XML documents. SAXCutter is an XMLFilter. That is, it functions as both a SAX application and a SAX parser. It functions as a SAX application by implementing the ContentHandler interface, which receives SAX events from an XML parser. It functions as an XML parser by firing SAX events to a SAX application. For example, Figure 7-1 shows how an application could use SAXCutter to cut a repetitive XML document and insert sub-documents into an XML column.

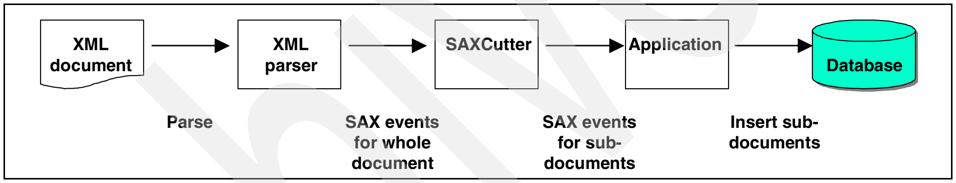


Figure 7-1 Cutting a repetitive document and insert sub-documents into the DB

If you have an existing SAX application, you can write a small controller to hook this up with the SAXCutter (Figure 7-2). In this case, the application will not realize that the ultimate source of the documents it is processing is a large, repetitive document. All it will see is a series of small documents.
The SAXCutter class is described in more detail in "The SAXCutter sample" on page 635.

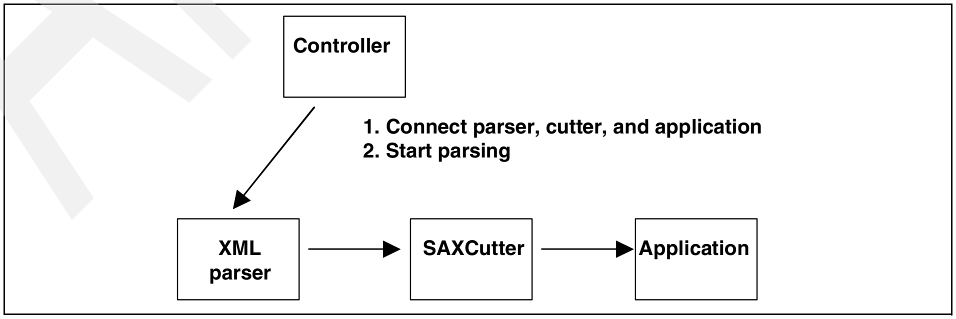


Figure 7-2 Using an existing application with the SAXCutter class

7.1.2 The CutterTester sample application

As an example of a SAX application that uses SAXCutter, the following code (Example 7-2) is from the CutterTester. This is a sample application that uses SAXCutter to cut a repetitive document into pieces and store each one in a separate file. This code accepts the name of an XML file, and the namespace URI, and the local name of the element type to cut on. It then instantiates an XMLReader (1), instantiates a SAXCutter and hooks this up to the XMLReader (2), sets up some global variables (3), and calls the parse method on the XMLReader (4).

Example 7-2 CutterTester sample code

```
public void run(String filename, String uri, String localName)
      throws Exception
   {
      XMLReader xmlReader;
      SAXCutter cutter;
      // 1. Get a new XMLReader and set namespace processing on.
      xmlReader = getXMLReader();
      xmlReader.setFeature("http://xml.org/sax/features/namespaces", true);
      xmlReader.setFeature("http://xml.org/sax/features/namespace-prefixes",
                           true);
      // 2. Get a new SAXCutter, set the ContentHandler to
      //    this CutterTester object, and set the cut element type.
      cutter = new SAXCutter(xmlReader);
      cutter.setContentHandler(this);
      cutter.setCutElementType(uri, localName);
      // 3. Set up the global variables.
      baseName = getBaseName(filename);
      extension = getExtension(filename);
      fileNumber = 1;
      // 4. Parse the input file.
      cutter.parse(new InputSource(new FileInputStream(filename)));
   }
```

The actual work is done in the ContentHandler methods. For example, the CutterTester implements the following startDocument and endDocument methods (see Example 7-3 on page 195). startDocument constructs a new filename and opens a FileWriter over it. Methods like startElement, endElement, and characters use this Writer to serialize the XML document to the file. endDocument simply closes the file.

Example 7-3 startDocument and endDocument methods of CutterTester

```
public void startDocument () throws SAXException
   {
      String     filename;
      // Construct the next filename and increment the file number,
      // then open a FileWriter over the file.
      filename = baseName + String.valueOf(fileNumber) + extension;
      fileNumber++;
      try
      {
         writer = new FileWriter(filename);
      }
      catch (IOException io)
      {
         throw new SAXException(io);
      }
   }
public void endDocument() throws SAXException
   {
      // Close the file.
      try
      {
         writer.close();
      }
      catch (IOException io)
      {
         throw new SAXException(io);
      }
   }
```

7.1.3 Writing your own application

Of course, you will probably want to use SAXCutter for something different. The easiest way to do this is to modify the CutterTester class. For example, you can easily modify CutterTester to insert each sub-document into an XML column. Suppose we have a table with two columns, SalesOrderNumber and SalesOrderDocument. To insert sub-documents into this table, we would need to:

- Allocate global variables to hold the sales order number, a JDBC Connection, and a JDBC PreparedStatement:

  ```
  private int               salesOrderNumber;
  private Connection        conn;
  private PreparedStatement  insert;
  ```

- Change the code that sets up global variables to allocate the connection and prepare an INSERT statement:

```
// Call a private function to get a connection, then allocate and
// prepare an insert statement.

    conn = getConnection();
    insert = conn.prepareStatement("INSERT INTO OrderDocs " +
                                   "VALUES (?, XMLVARCHAR(?))");
```

- Add code to startElement to get the sales order number. For example, if this is stored in the Number attribute of the SalesOrder element, this would be:

```
if (localName.equals("SalesOrder"))
   {
   String s = attrs.getValue("", "Number");
   salesOrderNumber = (Integer.valueOf(s)).intValue();
   }
```

- Change the code in startDocument to use a StringWriter instead of a FileWriter:

```
public void startDocument () throws SAXException
   {
   writer = new StringWriter();
   }
```

- Change the code in endDocument to insert the string into the database:

```
public void endDocument() throws SAXException
   {
   try
      {
      insert.setInt(1, salesOrderNumber);
      insert.setString(2, writer.toString());
      insert.executeUpdate();
      conn.commit();
      }
      catch (SQLException e)
         {
         throw new SAXException(e);
         }
   }
```

7.2 Using XSLT for bulk inserts

For large files, it is sometimes possible to increase the performance of shredding by using XSLT combined with the DB2 Load or Import utility. This is particularly true when the document contains large amounts of data to be inserted into only a few tables, as is the case with repetitive documents. More details on XSLT transformations can be found in 9.1.1, “XSL, stylesheets and transformation” on page 248.

7.2.1 Flattening XML documents

The process has two steps. First, you use XSLT to flatten the document. *Flattening* is the process of taking a deeply nested document and transforming it into one that can be used with the table-based mapping. That is, the data for each table is grouped together and all tables are at the same nesting level. For example, when we flatten the document in Example 7-4,

Example 7-4 Orders document before flattening

```
<Orders>
   <SalesOrder SONumber="123">
      <Customer CustNumber = "543">
         <CustName>ITSO Insurance, Inc.</CustName>
         <Street>123 Main St.</Street>
         <City>Chicago</City>
         <State>IL</State>
         <PostCode>60609</PostCode>
      </Customer>
      <OrderDate>20030708</OrderDate>
      <Item LineNumber="1">
         <Part PartNumber="TW47">
            <Description>Turkey wrench</Description>
            <Price>9.95</Price>
         </Part>
         <Quantity>10</Quantity>
      </Item>
      <Item LineNumber="2">
         <Part PartNumber="SEP12">
            <Description>Stuffing separator</Description>
            <Price>13.27</Price>
         </Part>
         <Quantity>5</Quantity>
      </Item>
   </SalesOrder>
   <SalesOrder SONumber="456">
      <Customer CustNumber = "563">
         <CustName>XYZ Industries</CustName>
         <Street>11 Pine St.</Street>
         <City>San Jose</City>
         <State>CA</State>
         <PostCode>95120</PostCode>
      </Customer>
      <OrderDate>20031029</OrderDate>
      <Item LineNumber="1">
         <Part PartNumber="CD32">
            <Description>T3 Bolt: Cast iron</Description>
            <Price>0.65</Price>
         </Part>
         <Quantity>100</Quantity>
      </Item>
      <Item LineNumber="2">
         <Part PartNumber="HSA230">
            <Description>Fan blade</Description>
            <Price>14.46</Price>
         </Part>
         <Quantity>8</Quantity>
      </Item>
   </SalesOrder>
</Orders>
```

we get the document in Example 7-5.

Example 7-5 Flattened orders document

```
<Tables>
   <Table Name="Orders">
      <Row>
         <SONumber>123</SONumber>
         <OrderDate>20030708</OrderDate>
         <CustomerNumber>543</CustomerNumber>
      </Row>
      <Row>
         <SONumber>456</SONumber>
         <OrderDate>20031029</OrderDate>
         <CustomerNumber>563</CustomerNumber>
      </Row>
   </Table>
   <Table Name="Customers">
      <Row>
         <CustomerNumber>543</CustomerNumber>
         <CustName>ITSO Insurance, Inc.</CustName>
         <Street>123 Main St.</Street>
         <City>Chicago</City>
         <State>IL</State>
         <PostCode>60609</PostCode>
      </Row>
      <Row>
         <CustomerNumber>563</CustomerNumber>
         <CustName>XYZ Industries</CustName>
         <Street>11 Pine St.</Street>
         <City>San Jose</City>
         <State>CA</State>
         <PostCode>95120</PostCode>
      </Row>
   </Table>
   <Table Name="Items">
      <Row>
         <SONumber>123</SONumber>
         <LineNumber>1</LineNumber>
         <PartNumber>TW47</PartNumber>
         <Quantity>10</Quantity>
      </Row>
      <Row>
         <SONumber>123</SONumber>
         <LineNumber>2</LineNumber>
         <PartNumber>SEP12</PartNumber>
         <Quantity>5</Quantity>
      </Row>
      <Row>
         <SONumber>456</SONumber>
         <LineNumber>1</LineNumber>
         <PartNumber>CD32</PartNumber>
         <Quantity>100</Quantity>
      </Row>
      <Row>
         <SONumber>456</SONumber>
         <LineNumber>2</LineNumber>
         <PartNumber>HSA320</PartNumber>
         <Quantity>8</Quantity>
      </Row>
   </Table>
   <Table Name="Parts">
      <Row>
         <PartNumber>TRW7</PartNumber>
         <Description>Turkey wrench</Description>
         <Price>9.95</Price>
      </Row>
      <Row>
         <PartNumber>SEP12</PartNumber>
         <Description>Stuffing separator</Description>
         <Price>13.27</Price>
      </Row>
      <Row>
         <PartNumber>CD32</PartNumber>
         <Description>T3 Bolt: Cast iron</Description>
         <Price>0.65</Price>
      </Row>
      <Row>
         <PartNumber>HSA320</PartNumber>
```

```
            <Description>Fan blade</Description>
            <Price>14.46</Price>
         </Row>
      </Table>
   </Tables>
```

One important thing to notice here is that we have copied primary key values to the rows that reference them. For example, we copied the customer number from the CustNumber attribute of the Customer element to the CustomerNumber element in each sales order row. And we have copied the sales order number from the SONumber attribute of the SalesOrder element to the SONumber element in each line item row. This ensures that the rows will be properly linked together in the database.

7.2.2 Converting from XML to the DB2 load format

In addition to flattening the XML document, we also need to convert the data from elements and attributes to the DB2 load format. In this format, column values are separated by commas and rows are separated by new line characters (x'0A'). In addition, specific formats are used for each data type. For example, strings are quoted with a double quote, dates use *yyyymmdd* format, and null values are indicated by no data between column delimiters —that is, two commas in a row.

Note: On Windows or OS/2®, this can also be a carriage return/line feed (x'0D0A'). On EBCDIC systems, this should be the EBCDIC LF character (x'25').

Tip: If the data in your XML document does not use the formats used by the DB2 load format (for example, your dates use the *yy-mm-dd* format instead of the *yyyymmdd* format), you must either convert the data to before transforming the document, or you must call formatting routines from your stylesheet. For a complete description of the DB2 load format, see Appendix C, Export/Import/Load Utility File Formats, of the Data Movement Utilities Guide and Reference.

To convert the data to the DB2 load format, we use the same stylesheet as the one that flattens our document. Thus, the output of our XSLT transformation is actually the following document (Example 7-6 on page 201).

Example 7-6 Output of XSLT transformation

```
<Tables>
   <Table Name="Orders">123,20030708,543
                        456,20031029,563
   </Table>
   <Table Name="Customers">
          543,"ITSO Insurance, Inc.","123 Main St.","Chicago","IL","60609"
          563,"XYZ Industries","11 Pine St.","San Jose","CA","95120"
   </Table>
   <Table Name="Items">123,1,"TW47",10
                       123,2,"SEP12",5
                       456,1,"CD32",100
                       456,2,"HSA320",8
   </Table>
   <Table Name="Parts">"TRW7","Turkey wrench",9.95
                       "SEP12","Stuffing separator",13.27
                       "CD32","T3 Bolt: Cast iron",0.65
                       "HSA320","Fan blade",14.46
   </Table>
</Tables>
```

This conforms to the following DTD:

```
<!ELEMENT Tables (Table+)>
<!ELEMENT Table (#PCDATA)>
<!ATTLIST Table
          Name CDATA #REQUIRED>
```

7.2.3 The XSLT stylesheet

The stylesheet we use to flatten and convert the document is shown in Example 7-7.

Example 7-7 Stylesheet to flatten and convert

```
<?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" encoding="UTF-8" indent="yes"

xalan:indent-amount="2"/>
<xsl:strip-space elements="*"/>
```

```
  <!-- ************************ -->
  <!-- Construct Table elements -->
  <!-- ************************ -->
  <xsl:template match="/">
   <Tables>
      <Table name="Orders">
         <xsl:for-each select="/Orders/SalesOrder">
            <xsl:apply-templates select="."/>
         </xsl:for-each>
      </Table>
      <Table name="Customers">
         <xsl:for-each select="/Orders/SalesOrder/Customer">
            <xsl:apply-templates select="."/>
         </xsl:for-each>
      </Table>
      <Table name="Items">
         <xsl:for-each select="/Orders/SalesOrder/Item">
            <xsl:apply-templates select="."/>
         </xsl:for-each>
      </Table>
      <Table name="Parts">
         <xsl:for-each select="/Orders/SalesOrder/Item/Part">
            <xsl:apply-templates select="."/>
         </xsl:for-each>
      </Table>

   </Tables>
  </xsl:template>

  <!-- ***************** -->
  <!-- Construct row data -->
  <!-- ***************** -->
  <xsl:template name="order" match="/Orders/SalesOrder">
     <xsl:value-of select="./@SONumber"/>,<xsl:value-of

select="./Customer/@CustNumber"/>,<xsl:value-of select="./OrderDate/text()"/>
  </xsl:template>
  <xsl:template name="customer" match="/Orders/SalesOrder/Customer">
      <xsl:value-of select="./@CustNumber"/>,"<xsl:value-of

select="./CustName/text()"/>","<xsl:value-of select="./Street/text()"/>",
"<xsl:value-of select="./City/text()"/>","<xsl:value-of

select="./State/text()"/>","<xsl:value-of select="./PostCode/text()"/>"
  </xsl:template>

  <xsl:template name="item" match="/Orders/SalesOrder/Item">
      <xsl:value-of select="../@SONumber"/>,<xsl:value-of
```

```
select="./@LineNumber"/>,"<xsl:value-of

select="./Part/@PartNumber"/>","<xsl:value-of select="./Quantity/text()"/>"
  </xsl:template>

  <xsl:template name="part" match="/Orders/SalesOrder/Item/Part">
      "<xsl:value-of select="./@PartNumber"/>","<xsl:value-of

select="./Description"/>",<xsl:value-of select="./Price/text()"/>
  </xsl:template>
</xsl:stylesheet>
```

The first part of the stylesheet flattens the document. It constructs a separate `<Table>` element for the data to be inserted into each table. For example, the following template constructs a `<Table>` element for the Parts table:

```
<Table name="Parts">
   <xsl:for-each select="/Orders/SalesOrder/Item/Part">
      <xsl:apply-templates select="."/>
   </xsl:for-each>
</Table>
```

The second part of the document constructs the rows for each table. For example, the following template constructs rows for the Parts table:

```
<xsl:template name="part" match="/Orders/SalesOrder/Item/Part">
    "<xsl:value-of select="./@PartNumber"/>","<xsl:value-of
    select="./Description"/>",<xsl:value-of select="./Price/text()"/>
</xsl:template>
```

Individual values are extracted with the `xsl:value-of element`. If the value is a string, such as the part number or description, its value is placed in double quotes. If the value is a number, such as the price, its value is inserted without any further formatting. Commas separate each value. Note that if a value is missing from the XML document —in database terms, this is a null value— then the result is two consecutive commas. This is the format that DB2 uses to indicate a null value.

7.2.4 Cutting the flattened document

Although we have flattened the document and converted the data to the DB2 load format, we are still not ready to load the data into DB2. The reason is that the transformed document still contains XML markup, as well as data to be inserted into multiple tables. The Load and Import utilities do not understand XML and require that a file contain the data for a single table. We therefore need to remove the XML markup from our document and cut it into separate documents, each of which contain the data for a single table.

Although we could have created these files directly from the original document, this is more expensive than using our intermediate format. The problem is that XSLT can only create one output document per transformation. Thus, we would need to perform one XSLT transformation per target table. Since parsing is expensive, this means that performance will degrade as the number of target tables increases. By using our intermediate format, we parse only two documents: we parse the first document during transformation to our intermediate format, and we parse the intermediate document while cutting it into per-table documents.

To cut intermediate documents into separate DB2 load files, we use a Java application named TableCutter. This uses the SAXCutter class discussed in 7.1, “An XML cutter to process repetitive documents” on page 192. For example, if we cut our transformed document with this application, the output is four DB2 load files shown in Example 7-8.

Example 7-8 Cut up document into separate load files

```
Orders.del
----------
123,20030708,543
456,20031029,563


Customers.del
-------------
543,"ITSO Insurance, Inc.","123 Main St.","Chicago","IL","60609"
563,"XYZ Industries","11 Pine St.","San Jose","CA","95120"


Items.del
---------
123,1,"TW47",10
123,2,"SEP12",5
456,1,"CD32",100
456,2,"HSA320",8
```

```
Parts.del
---------
"TRW7","Turkey wrench",9.95
"SEP12","Stuffing separator",13.27
"CD32","T3 Bolt: Cast iron",0.65
"HSA320","Fan blade",14.46
```

The command line syntax of the TableCutter application is:

```
java TableCutter <filename> <output-directory>
```

where <filename> is the name of the intermediate file, and <output-directory> is the directory in which to place the load files. TableCutter can also be called programmatically through its cut method. For example:

```
TableCutter cutter = new TableCutter();
cutter.cut(filename, outputDirectory);
```

There are two important parts to the TableCutter application. The cut method instantiates an XMLReader (1), instantiates a SAXCutter and hooks this up to the XMLReader (2), sets up some global variables (3), and calls the parse method on the XMLReader (4). See for details Example 7-9.

Example 7-9 TableCutter - cut method

```
public void cut(String filename, String outputDirectory)
      throws Exception
   {
      XMLReader xmlReader;
      SAXCutter cutter;
      // 1. Get a new XMLReader and set namespace processing on.
      xmlReader = getXMLReader();
      xmlReader.setFeature("http://xml.org/sax/features/namespaces", true);
      xmlReader.setFeature("http://xml.org/sax/features/namespace-prefixes",
                                 true);
      // 2. Get a new SAXCutter, set the ContentHandler to
      //    this TableCutter object, and set the cut element type to Table.
      cutter = new SAXCutter(xmlReader);
      cutter.setContentHandler(this);
      cutter.setCutElementType("", "Table");
      // 3. Set the global variables.
      this.outputDirectory = outputDirectory;
      writer = null;
      // 4. Parse the input file.
      cutter.parse(new InputSource(new FileInputStream(filename)));
   }
```

The startElement method looks for Table elements. When it finds one, it opens a new file (Example 7-10).

Example 7-10 startElement method

```
public void startElement (String uri, String localName, String qName,
                             Attributes attrs)
     throws SAXException
   {
      String tableName, filename;
      // If we are not on the Table element, just return.
      if ((uri.length() != 0) || (!localName.equals("Table"))) return;
      // Create a new output file.
      try
      {
         tableName = attrs.getValue("Name");
         filename = outputDirectory + tableName + ".del";
         writer = new FileWriter(filename);
      }
      catch (IOException io)
      {
         throw new SAXException(io);
      }
   }
```

The characters method (Example 7-11) writes the load data out to the file:

Example 7-11 characters method

```
public void characters (char ch[], int start, int length)
     throws SAXException
   {
      try
      {
         if (writer != null) writer.write(ch, start, length);
      }
      catch (IOException io)
      {
         throw new SAXException(io);
      }
   }
```

The endElement method closes the file when it finds a Table element (Example 7-12 on page 207).

Example 7-12 endElement method

```
public void endElement (String uri, String localName, String qName)
         throws SAXException
   {
      // If we are not on the Table element, just return.
      if ((uri.length() != 0) || (!localName.equals("Table"))) return;
      // Close the file.
      try
      {
         writer.close();
      }
      catch (IOException io)
      {
         throw new SAXException(io);
      }
      // Set the writer to null so that we only write to the file when
      // we are inside the Table element.
      writer = null;
   }
```

For the complete code of the TableCutter class, see "The TableCutter tool" on page 649.

7.2.5 Importing or loading the data

To load the data into DB2, we can use the Import utility or the Load utility. The Import utility inserts data from an input file into a table or updateable view. It can either replace existing data or append the new data to the existing data. The Load utility also inserts data from an input file into a table (but not a view) and can either replace existing data or append the new data to the existing data.

The main difference between the utilities is that Load is faster than Import. This is because:

- Load writes directly to the database table, while Import uses INSERT statements.
- Load builds indexes after inserting all the data, while Import builds indexes as it inserts rows.
- Import logs all data, while Load performs minimal logging.

Of course, better performance comes at a price. Load does not fire triggers, does not perform referential or table constraints checking (other than validating the uniqueness of the indexes), and does not support updateable views. For a complete list of differences between the Import and Load utilities, see “Appendix B. Differences Between the Import and the Load Utility” in the Data Movement Utilities Guide and Reference.

Import and Load can both be invoked from the command line, the Control Center, or through an API (sqluimpr or sqluload). For example, to import data from the Items.del file into the ITEMS table, we use the following command:

```
IMPORT FROM "C:\load\Items.del" OF DEL METHOD P (1, 2, 3, 4)
INSERT INTO ITEMS (SONUMBER, LINENUMBER, PARTNUMBER, QUANTITY)
```

FROM gives the name of the file from which to load data. OF DEL means that we are using comma-delimited files (DEL=delimited). METHOD P lists the positions of the fields in our file to import; in our case, we are loading all four fields. INSERT INTO gives the name of the table and the names of the columns into which the data is to be loaded. The order of the column names corresponds to the fields listed in the METHOD P clause.

To load data into the Items table, we use the following command:

```
LOAD FROM "C:\load\Items.del" OF DEL METHOD P (1, 2, 3, 4)
INSERT INTO ITEMS (SONUMBER, LINENUMBER, PARTNUMBER, QUANTITY)
```

For more information about the Import and Load utilities, see the Data Movement Utilities Guide and Reference.

8

Publishing data as XML

In this chapter, we take a look at the different ways of publishing XML from relational data.

First, we walk through the SQL/XML support which is new in DB2 UDB V8. Then, we take a look at DB2 XML Extender which is now shipped as part of the DB2 V8 product code. We dissect the DAD file which is required by the XML Extender to publish documents, and we look at the stored procedures provided by DB2 XML Extender to externalize the documents.
We also take some time to look at the pros and cons of writing your own code to publish documents, best practices, and limitations of SQL/XML and DB2 XML Extender.

This chapter consist of the following topics:

- Publishing data using SQL/XML
- Publishing data with XML Extender
- Writing your own code to publish data as XML

8.1 Publishing data using SQL/XML

Relational data is the backbone of many businesses. With XML as a universal data exchange format, the capability of constructing XML data from existing relational data, while preserving the power of SQL, tremendously simplifies business-to-business (B2B) application development.

To help integrate both worlds, an informal group of companies, including IBM, Microsoft, Oracle, and Sybase began to define XML extensions for SQL in early 2000 in a working group called *SQLX* (`http://www.sqlx.org`). The group focuses on SQL capabilities and consciously avoids vendor extensions, while encouraging state-of-the-art and projected future developments. SQLX forwards proposals to the INCITS (International Committee for Information Technology Standards) H2 Database Committee for approval.

Currently the following functions are available in DB2 UDB for Linux, UNIX and Windows, V8:

- XMLELEMENT and XMLATTRIBUTES to construct XML elements with attributes
- XMLFOREST to construct a sequence of XML elements
- XMLCONCAT to concatenate XML elements
- XMLAGG to aggregate XML elements

These functions are also available with DB2 for z/OS V8. In addition, DB2 for z/OS supports the XMLNAMESPACES function, which was recently added to the SQL/XML standard. XMLNAMESPACES will also be supported in V8.2 of DB2 UDB for Linux, UNIX and Windows.

Let us now look at each of these SQL/XML functions in more detail. To demonstrate their use, and make it easy for the reader to reproduce them, we use the SAMPLE database supplied with DB2.

8.1.1 Constructing elements and attributes

To construct XML elements and attributes, we use the XMLELEMENT, XMLATTRIBUTES, and XMLFOREST SQL/XML functions.

XMLELEMENT and XMLATTRIBUTES

`XMLELEMENT()` is used to construct a new XML element (with attributes and content). The syntax diagram is shown in Figure 8-1 on page 211. XML element content is constructed from a variable list of value expressions.

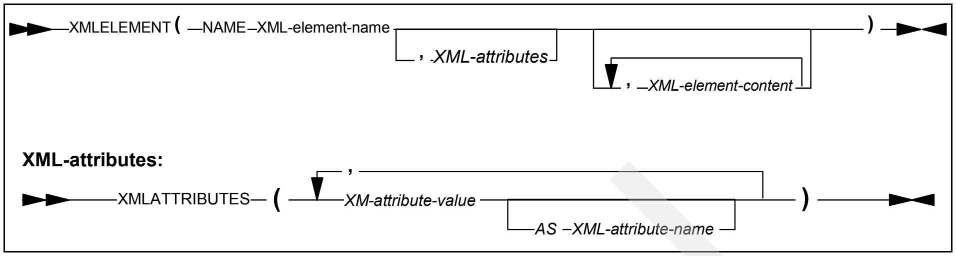


Figure 8-1 XMLElement and XMLAttributes function

Attribute names and values are specified via the **XMLATTRIBUTES()** function through column names or aliases for value expressions.

The result of the value expressions is mapped from SQL to XML according to the mapping rules specified in SQL/XML. SQL/XML defines rules to map SQL identifiers and XML identifiers, SQL data types and XML Schema types, and SQL data and XML data at a value, table, schema, and catalog level.

The following examples all use the **XML2CLOB()** cast function. This function is used to externalize the internal XML data type as a CLOB, as explained in 8.1.3, “Behind the scenes: the XML data type” on page 219. All SQL/XML queries need to be cast their result set data to a CLOB.

In Example 8-1, we produce an XML fragment that includes some employee details and their total salary, made up of their normal salary, bonuses and commission.

Example 8-1 Using XMLELEMENT and XMLATTRIBUTES

```
SELECT
  XML2CLOB(
   XMLELEMENT (NAME "EmployeeSalary",
      XMLATTRIBUTES (e.empno AS "id"),
         XMLELEMENT (NAME "Firstname", e.firstnme),
         XMLELEMENT (NAME "Lastname", e.lastname),
         XMLELEMENT (NAME "TotalSalary",(e.salary+e.bonus+e.comm))
   )
  )
FROM employee e WHERE SEX  = 'F'
```

The query returns a result set, where each row contains an XML fragment like Example 8-2 on page 212.

Example 8-2   Result of XMLELEMENT and XMLATTRIBUTES

```
<EmployeeSalary id="000010">
   <Firstname>CHRISTINE</Firstname>
   <Lastname>HAAS</Lastname>
   <TotalSalary>000057970.00</TotalSalary>
</EmployeeSalary>XMLFOREST
```

XMLFOREST

XMLFOREST() simplifies the coding of SQL/XML queries, as it constructs sequences of XML elements from SQL value expressions in the order of the expressions. XMLFOREST is a short-hand for a sequence of XMLELEMENT invocations. XMLFOREST takes a variable list of SQL value expressions as input, and produces for each expression an XMLELEMENT with, the column name or alias of the expression as the tag name, and the value of the expression as the element content. An example is shown in Example 8-3.

Example 8-3   Using XMLFOREST

```
select
  XML2CLOB (
   XMLELEMENT (NAME "EmployeeSalary",
      XMLATTRIBUTES(e.empno AS "id"),
      XMLFOREST(e.firstnme AS "Firstname",
                e.lastname AS "lastname",
                e.salary+e.bonus+e.comm as "salary")
   )
  )
FROM employee e WHERE SEX = 'F’
```

This XMLFOREST query produces the following result (Example 8-4).

Example 8-4   Result of XMLFOREST

```
<EmployeeSalary id="000010">
   <Firstname>CHRISTINE</Firstname>
   <lastname>HAAS</lastname>
   <salary>000057970.00</salary>
</EmployeeSalary>
```

XMLFOREST and XMLELEMENT handle null values differently. XMLFOREST ignores a null value, and does not include it in the result, whereas XMLELEMENT returns an empty element.

Concatenation using XMLCONCAT

XMLCONCAT() takes a variable number of XML value expressions and constructs a single XML value as a sequence of XML values. This function is used to construct an XML element from pieces of independently constructed XML. An example is shown in Example 8-5.

Example 8-5 Using XMLCONCAT

```
select XML2CLOB(
      XMLCONCAT(
         XMLELEMENT(NAME "Employee",
            XMLATTRIBUTES(e.firstnme ||' '|| e.lastname as "Name"),
               XMLELEMENT(NAME "Salary",e.salary)
         ),
         XMLELEMENT(NAME "Employee",
            XMLATTRIBUTES(e.firstnme ||' '|| e.lastname as "Name"),
               XMLELEMENT(NAME "Bonus",e.bonus)
         ),
         XMLELEMENT(NAME "Employee",
            XMLATTRIBUTES(e.firstnme ||' '|| e.lastname as "Name"),
               XMLELEMENT(NAME "Commission",e.comm)
         )
      )
   )
from employee e where sex = 'F'
```

This query returns the following result (Example 8-6 on page 214).

Example 8-6 Result of XMLCONCAT

```
<Employee Name="CHRISTINE HAAS">
   <Salary>0052750.00</Salary>
</Employee>
<Employee Name="CHRISTINE HAAS">
   <Bonus>0001000.00</Bonus>
</Employee>
<Employee Name="CHRISTINE HAAS">
   <Commission>0004220.00</Commission>
</Employee>
```

Aggregating data across rows using XMLAGG

`XMLAGG()` is an aggregate function, which constructs an XML value from a collection of XML value expressions. XMLAGG resolves the 1:n relationships in XML. The expression to be aggregated, and the expressions in the ORDER BY clause, do not need to be functionally dependent on the grouping columns.

The ORDER BY in Example 8-7 is used to list the names in alphabetical order and we use the GROUP BY clause to allow wrapping of the employees by department.

Example 8-7 Using XMLAGG

```
select XML2CLOB (
      XMLELEMENT(
         NAME "Department",
            XMLATTRIBUTES(e.workdept as "Name"),
            XMLAGG(
               XMLELEMENT(NAME "Employee", e.firstnme||' '||e.lastname)
             order by e.lastname)
         )
      )
from employee e
group by workdept
```

This query returns the following result (Example 8-6):

Example 8-8 Result of XMLAGG

```
<Department Name="A00">
   <Employee>CHRISTINE HAAS</Employee>
   <Employee>VINCENZO LUCCHESSI</Employee>
   <Employee>SEAN O'CONNELL</Employee>
</Department>

<Department Name="B01">
   <Employee>MICHAEL THOMPSON</Employee>
</Department>

<Department Name="C01">
   <Employee>SALLY KWAN</Employee>
   <Employee>HEATHER NICHOLLS</Employee>
   <Employee>DOLORES QUINTANA</Employee>
</Department>
```

Note that the different employees under the same department are extracted from multiple rows in the DB2 table.

XMLNAMESPACES

DB2 UDB V8.2 for Linux, UNIX, and Windows (as well as DB2 for z/OS V8) introduce a new SQL/XML function, XMLNAMESPACES(). It provides XML namespace declarations within the SQL/XML publishing functions XMLELEMENT and XMLFOREST. An XML namespace is a collection of names that is identified by a uniform reference identifier (URI). Namespaces are used in XML documents as element types and attribute names.

To give the reader some impression of this, we provide a simple example. Example 8-9 generates a sequence of elements produced from arguments of an XMLFOREST function. The example declares a default namespace to be associated with the first element, and a namespace whose prefix is “d”, associated with the second element.

Example 8-9 XMLNAMESPACES function

```
SELECT empno, XMLSERIALIZE(CONTENT
        XMLFOREST (XMLNAMESPACES(DEFAULT ’http://hr.org’,
                                         ’http://fed.gov’AS "d"
                                ),
                   lastname,
                   job AS "d:job"
                  ) AS CLOB
                          ) AS "Result"
```

```
       FROM employee
       WHERE edlevel = 12
```

The result of the query is shown in Example 8-10. This query generates an XML value with a textual representation (the output is formatted here for convenience; the output XML fragment has no extraneous white space characters, and the output generally appears as one line).

Example 8-10   Result of XMLNAMESPACES function

```
EMPNO    Result
-----    ------------------------------------------------------------------
000290   <LASTNAME xmlns="http://hr.org" xmlns:d="http://fed.gov">PARKER
         </LASTNAME>
         <d:job xmlns="http://hr.org" xmlns:d="http://fed.gov">OPERATOR
         </d:job>

000310   <LASTNAME xmlns="http://hr.org" xmlns:d="http://fed.gov">SETRIGHT
         </LASTNAME>
         <d:job xmlns="http://hr.org" xmlns:d="http://fed.gov">OPERATOR
         </d:job>
```

8.1.2 A more complex example

In this example (Example 8-11) we combine all of the functions we have described before, and use them to produce an HTML table displaying, Department Number, Department Name, Employee, Name and Phone. What makes this query interesting it the ability to use XMLCONCAT and XMLAGG to create a hierarchy.

When we wrote this redbook, you could only run this query against DB2 for z/OS V8. The ability to use a subquery in an XMLAGG function was not supported on the DB2 for LUW V8 FixPak 4 at the time of writing of this publication.

Example 8-11   Producing an HTML document using SQL/XML

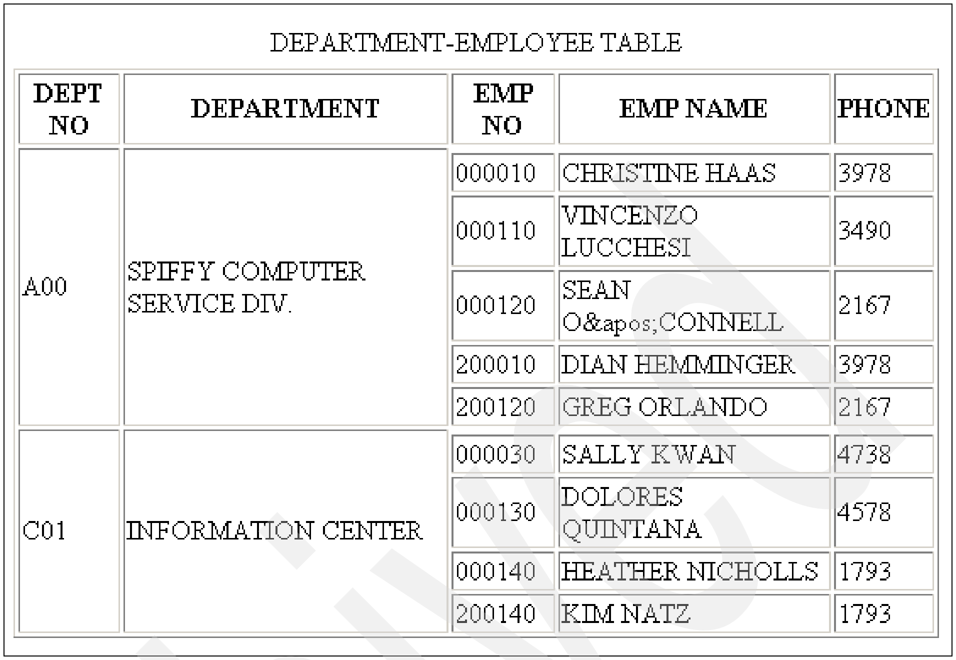
```
SELECT VARCHAR(
   XML2CLOB(XMLELEMENT(NAME "TABLE",
          XMLATTRIBUTES('1' AS "BORDER"),
          XMLELEMENT(NAME CAPTION,'DEPARTMENT-EMPLOYEE TABLE'),
          XMLELEMENT(NAME TR,XMLFOREST('DEPT NO' AS TH,
                                       'DEPARTMENT'AS TH,
                                       'EMP NO' AS TH,
                                    'EMP NAME' AS TH,
                                    'PHONE' AS TH)
                  ),
         XMLAGG(
            XMLCONCAT(
               XMLELEMENT(NAME TR, XMLELEMENT(NAME TD,
                  XMLATTRIBUTES(X.CNT+1 AS "ROWSPAN"),D.DEPTNO),
                  XMLELEMENT(NAME TD,
                     XMLATTRIBUTES(X.CNT+1 AS "ROWSPAN"),D.DEPTNAME)
                  ),
                  (SELECT XMLAGG(XMLELEMENT(NAME TR,
                     XMLFOREST(EMPNO AS TD,FIRSTNME||' '||LASTNAME AS TD,
                              PHONENO AS TD
                             )
                                         )
                               )
                     FROM DSN8810.EMP E
                     WHERE E.WORKDEPT = D.DEPTNO
                  )
               )
            )
         )
      )
   )
FROM DSN8810.DEPT D,(SELECT WORKDEPT,COUNT(*)
                     FROM DSN8810.EMP GROUP BY WORKDEPT) X(DEPTNO,CNT)
   WHERE D.DEPTNO = X.DEPTNO AND D.DEPTNO IN ('A00','C01')
```

The above example produces the following HTML table (Example 8-12).

Example 8-12 Resulting HTML table

```
<TABLE BORDER="1"><CAPTION>DEPARTMENT-EMPLOYEE TABLE</CAPTION>
<TR><TH>DEPT NO</TH><TH>DEPARTMENT</TH><TH>EMP NO</TH><TH>EMP
NAME</TH><TH>PHONE</TH></TR>
<TR><TD ROWSPAN="6">A00</TD><TD ROWSPAN="6">SPIFFY COMPUTER SERVICE
DIV.</TD></TR>
<TR><TD>000010</TD><TD>CHRISTINE HAAS</TD><TD>3978</TD></TR>
<TR><TD>000110</TD><TD>VINCENZO LUCCHESI</TD><TD>3490</TD></TR>
<TR><TD>000120</TD><TD>SEAN O&apos;CONNELL</TD><TD>2167</TD></TR>
<TR><TD>200010</TD><TD>DIAN HEMMINGER</TD><TD>3978</TD></TR>
<TR><TD>200120</TD><TD>GREG ORLANDO</TD><TD>2167</TD></TR>
<TR><TD ROWSPAN="5">C01</TD><TD ROWSPAN="5">INFORMATION CENTER</TD></TR>
<TR><TD>000030</TD><TD>SALLY KWAN</TD><TD>4738</TD></TR>
<TR><TD>000130</TD><TD>DOLORES QUINTANA</TD><TD>4578</TD></TR>
<TR><TD>000140</TD><TD>HEATHER NICHOLLS</TD><TD>1793</TD></TR>
<TR><TD>200140</TD><TD>KIM NATZ</TD><TD>1793</TD></TR></TABLE>
```

When you store this result in a file and open it with a Web browser, the result looks like Figure 8-2 on page 219.



DEPARTMENT-EMPLOYEE TABLE

DEPT NO	DEPARTMENT	EMP NO	EMP NAME	PHONE
A00	SPIFFY COMPUTER SERVICE DIV.	000010	CHRISTINE HAAS	3978
		000110	VINCENZO LUCCHESI	3490
		000120	SEAN O&apos;CONNELL	2167
		200010	DIAN HEMMINGER	3978
		200120	GREG ORLANDO	2167
C01	INFORMATION CENTER	000030	SALLY KWAN	4738
		000130	DOLORES QUINTANA	4578
		000140	HEATHER NICHOLLS	1793
		200140	KIM NATZ	1793

Figure 8-2 Result of the more complex SQL/XML statement

In the emp name column the name Sean O’Connell has been displayed as SEAN &apos;CONNELL. This is XML standard representation of an apostrophe. This is the result of applying the DB2 to XML mapping rules.

8.1.3 Behind the scenes: the XML data type

SQL/XML is part of the ISO SQL specification (Information technology - Database languages - SQL - Part 14: XML-Related Specifications (SQL/XML) ISO/IEC 9075-14:2003). It became an “official” standard in December 2003. SQL/XML support can be found in DB2 UDB for Linux, Unix and Windows, and in DB2 for z/OS V8.

One of the goals of SQL/XML is to provide a rich set of XML operations to permit XML data to be accessed/generated from within SQL. The SQL/XML functions discussed earlier are designed precisely to help achieve this goal of constructing XML values and persisting them within the XML data type.

Because the standardization work for the XML data type is not yet finalized (although a first “attempt” is part of the aforementioned standard, but is likely to change in the near future), DB2 has not yet implemented a native XML data type at the time of writing of this publication.

However, DB2 is able to support the SQL/XML functions without a true XML data type. To overcome this problem, DB2 returns the XML values as an internal data type, and the function XML2CLOB enables client applications to transform them from the internal data type to a CLOB, and to process the XML result as a CLOB data type.

Valid values for the XML data type include:

- An element
- An XMLforest of elements
- The textual content of an element
- An empty XML value

Until DB2 UDB for Linux, UNIX, and Windows V8.2, XML2CLOB was the only supported operation to convert (serialize) an XML data type value to a string data type value. Serialization is the inverse operation of parsing; it is the process of converting a parsed XML value into a textual XML value.

A new standard SQL/XML function, XMLSERIALIZE (with the CONTENT option) will be introduced in DB2 UDB for Linux, UNIX and Windows V8.2 (still under development at the time of writing of this publication). It will allow you to convert an XML data type value into a result string data type that is appropriate for the length of the XML output. XMLSERIALIZE converts an XML expression into an SQL string value which, in turn, can be bound out to host character variables. With XMLSERIALIZE (Figure 8-3), you can specify a result type like CHAR or VARCHAR, which might be more appropriate and result in better performance than CLOB.

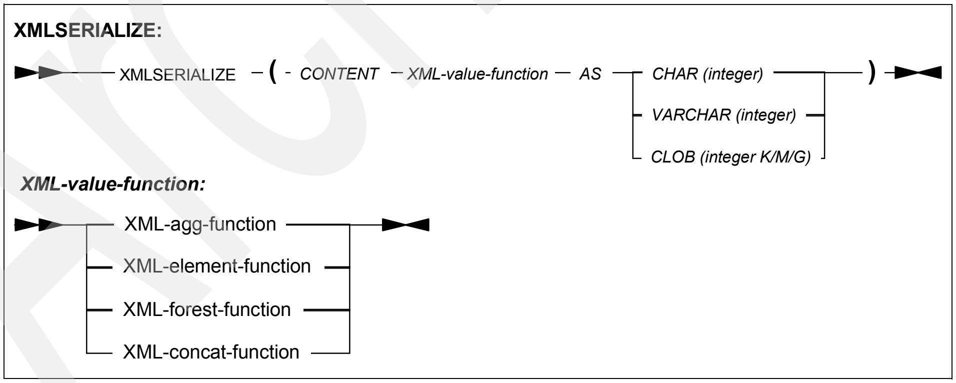


Figure 8-3 XMLSERIALIZE syntax diagram

The CONTENT option specifies that the value of XML-value-function can consist of more than one top-level element. Example 8-13 on page 221 shows the use of the XMLSERIALIZE function instead of XML2CLOB.

Example 8-13 Using XMLSERIALIZE

```
SELECT XMLSERIALIZE(CONTENT XMLELEMENT(NAME "Emp_Exempt",
                             XMLATTRIBUTES(e.firstnme,
                                           e.lastname AS "Lastname",
                                           e.midinit)) AS CLOB) AS "Result"
  FROM employee e
  WHERE e.lastname='GEYER'
```

Example 8-14 shows the result of the previous query.

Example 8-14 Result of XMLSERIALIZE

```
<Emp_Exempt
    FIRSTNME="JOHN"
    Lastname="GEYER"
    MIDINIT="B">
</Emp_Exempt>
```

8.2 Publishing data with XML Extender

As an alternative to publishing (creating) XML using SQL/XML, you can use DB2 XML Extender to create XML documents from data that is stored in relational tables.

In this section we use the samples supplied with DB2 XML Extender. You can find these in the *[install directory]*\samples\db2xml, where *[install directory]* is your DB2 install directory.

To use these examples we create and enable the SALES_DB database as follows:

1. Open a DB2 Command Window
2. Run the following commands to create the SALES_DB database, where **[install directory]** refers to the directory where DB2 is installed; the default for Windows is c:\program files\ibm\sqllib.

   cd \[install directory]\sample\db2xml\cmd\, and press **Enter**
   getstart_db, and press **Enter**
3. Now we need to prepare the database for the XML Extender stored procedures and DB2 CLI by running:

```
getstart_prep
```

The tables (Table 8-1 through Table 8-3) that are created, are structured as follows.

Table 8-1 ORDER _TAB

Column name	Data type
ORDER_KEY	INTEGER
CUSTOMER	VARCHAR(16)
CUSTOMER_NAME	VARCHAR(16)
CUSTOMER_EMAIL	VARCHAR(16)

Table 8-2 PART_TAB

Column name	Data type
PART_KEY	INTEGER
COLOR	CHAR(6)
QUANTITY	INTEGER
PRICE	DECIMAL(10,2)
TAX	REAL
ORDER_KEY	INTEGER

Table 8-3 SHIP_TAB

Column name	Data type
DATE	DATE
MODE	CHAR(6)
COMMENT	VARCHAR(128)
PART_KEY	INTEGER

8.2.1 DAD files

DB2 XML Extender uses Document Access Definition (DAD) files. The DAD is an XML formatted document. The DAD is used for a number of different things within XML Extender. It allows you to associate XML document structure with a DB2 database. The structure of a DAD file is different when using an XML column or an XML collection. When publishing XML data with DB2 XML Extender, the DAD file is used to map the relational data to an XML document (structure).

8.2.2 Publishing XML documents using a DAD file

When publishing XML data via XML Extender ‘s DAD files, you can use two different notations in the DAD file to describe the mapping between relational and XML data.

- *SQL Composition*: This notation uses an SQL SELECT statement, followed by instructions on how the resulting rows should be tagged as XML
- *RDB (relational database) Node*: This notation includes a list of the tables whose contents is to be tagged as XML, together with the primary foreign key relationships between the tables. The list of tables is followed by instructions on how the contents (or more typically a subset of the contents) should be tagged as XML.

Hereafter we discuss the construction of XML data using DAD files. We describe seven steps to consider when generating a DAD for publishing XML data.

1. Scoping the document content
2. Shaping the document structure
3. Mapping the relational content to the document
4. Controlling the number of documents generated
5. Outputting document header information
6. Validating the generated documents
7. Transforming the generated documents (for example, to HTML)

The steps we describe can be used to produce the following XML document from relational data (Example 8-15):

Example 8-15 Generated XML fragment

```
<order id=”4711”>
   <signdate>2002-03-18</signdate>
   <amount>24000</amount>
</order>
<order id=”4712”>
   <signdate>2002-03-19</signdate>
   <amount>44000</amount>
</order>
```

General constructs in DAD files used for XML publishing

A DAD itself is also an XML document. To be valid it has to adhere to the DTD for DADs. Here are some general things that you must code in every DAD that is used to generate XML documents from relational data (XML publishing).

- Create the DAD header

  ```
  <?xml version="1.0"?>
  <!DOCTYPE DAD SYSTEM "c:\progra~1\ibm\sqllib\samples\db2xml\dtd\dad.dtd">
  ```

- Insert the `<DAD>` and `</DAD>` tag. This element will contain all the other elements that make up the DAD.
- Insert the `<validation>` and `</validation>` tag to indicate whether DB2 XML Extender validates the XML document using the DTD in the DTD repository table (DTD_REF). Either specify:
  - `<validation>NO</validation>` ,if you do not want to validate
  - `<validation>YES</validation>`, if you intended to validate

  Validation is usually recommended, especially when you are exchanging information, and the other party expects the XML document to adhere to a certain structure (DTD). In our example, we decided to keep it simple and not use validation. Otherwise we would have had to create a DTD as well.
- Insert the `<Xcollection>` and `</Xcollection>` tags to specify the access and storage method as XML collection. XML Extender publishing functions using a DAD all use Xcollection (indicating that the data is stored in a collection of relational tables). All the other tags of the DAD go between the `<Xcollection>` and `</Xcollection>` tag.

1. Scoping the content of the generated documents

As mentioned above, there are two ways to ‘code’ DAD files for XML publishing:

- SQL composition
- RDB node mapping

Using an SQL composition DAD

The first part of the SQL composition DAD contains an SQL SELECT statement that retrieves all the rows and columns that are required in the XML document that we want to generate. The SQL statement is a regular SQL statement and can include join operations, subselects and SQL functions, as shown in Example 8-16 on page 225. The SQL statement is specified between the `<SQL_stmt>` and `</SQL_stmt>` tag.

Example 8-16 SQL statement in a DAD file

```
<SQL_stmt>
SELECT o.order_key,customer_name,customer_email,p.part_key,color,
      quantity,price,tax,ship_id,date,mode
 FROM  order_tab o,part_tab p,
      table(select substr(char(timestamp(generate_unique())),16) as ship_id
                   ,date,mode,part_key
            from ship_tab) s
 WHERE o.order_key = 1 and
      p.price > 20000 and
      p.order_key =o.order_key and
      s.part_key =p.part_key
 ORDER BY order_key,part_key,ship_id
</SQL_stmt>
```

Use the following guidelines when creating the SQL statement:

- Columns are specified in top-down order, by the hierarchy of the XML document structure
- The columns from the DB2 tables that will make up the an entity in the XML document are grouped together, and each group has an ‘object ID’ (candidate key) column; order_key, part_key, ship_id. The ‘object ID’ column is the first column in each group. For example, o.order_key precedes the columns related to the order element and p.part_key precedes columns for the part element. The SHIP_TAB table does not have a single key column, and therefore, the generate_unique DB2 built-in function is used to generate the ship_id column.
- The ‘object ID’ columns are then listed in top-down order in an ORDER BY statement.

Using a RDB_node mapping DAD

When using RDB_node mapping, the top element_node in the DAD file represents the root element of the XML document. Specify an RDB_node (using the `<RDB_node>` and `</RDB_node>` tags) for the top element_node as follows:

> **Note:** As you can see, RDB_node mapping uses the `<RDB_node>` tag instead of the `<SQL_stmt>` tag that is used for SQL statement mapping. In contrast to SQL statement DAD files, where the `<SQL_stmt>` tag is only used when specifying the SQL statement that makes up the XML document, RDB_node DAD files use the `<RDB_node>` tag also when defining elements and attributes. This is explained in more detail in the upcoming sections.

- Specify all tables that are associated with the XML document. The top element RDB_node contains a list of tables whose rows form the content of the generated document. The key relationships between the tables are listed here; the scoping portion of the DAD. Example 8-17 specifies three tables in the RDB_node of the element_node `<Order>`, which is the top element_node.

Example 8-17 RDB_node tables and conditions

```
<element_node name="Order">
<RDB_node>
   <table name="order_tab"/>
   <table name="part_tab"/>
   <table name="ship_tab"/>
      <condition>order_tab.order_key=part_tab.order_key AND
                 part_tab.part_key=ship_tab.part_key </condition>
</RDB_node>
```

- Use the orderBy attribute to recompose XML documents containing elements or attributes with multiple occurrence back to their original structure. This attribute allows you to specify the name of a column that will be used to preserve the order of the document. The orderBy attribute is part of the table element in the DAD file, and it is an optional attribute.

As you can see we only specified the join condition in the previous example. A condition can be added to restrict the documents generated. For example, to see only those that have an ‘open’ status, we can insert the following check into the DAD:

```
<condition>status=’open’</condition>
```

> **Note:** While DAD files using SQL statement mapping are only used for XML publishing, DAD files using RDB_node mapping can also be used for shredding. They are called bi-directional. In this chapter we only discuss the XML publishing capabilities of a DAD file using RDB_node mapping.

2. Shaping the structure of the generated documents

For both the SQL composition DAD and RDB_node DAD, the shape of the output document is governed by the structural tag layout in the second part of the DAD. Multiple hierarchies can be generated in a single document, and the way elements repeat can be controlled. Example 8-18 shows this for an SQL composition DAD.

Example 8-18 Shaping the structure of the XML document

```
<root_node>
   <element_node name=”order”>
      <attribute_node name=”id”>
         <column name=”id”/>
      </attribute_node>
      <element_node name=”signdate”>
         <text_node>
            <column name=”cdate”/>
         </text_node>
      </element_node>
      <element_node name=”amount”>
         <text_node>
            <column name=”total”/>
         </text_node>
      </element_node>
   </element_node>
</root_node>
```

Shape the XML document structure using the following types of nodes:

element_node

Specifies an element in the XML document. element_nodes can have child element_nodes.

attribute_node

Specifies the attribute of an element in the XML document.

Follow these guidelines for each type of node.

- For each element in the XML document, define an `<element_node>` tag with the name attribute (name=) set to the element’s name, as shown in Example 8-19.

Example 8-19 Adding element_node tags

```
<root_node>
<element_node name="Order">
      <element_node name="Customer">
         <element_node name="Name">
         </element_node>
         <element_node name="Email">
         </element_node>
      </element_node>
...
</element_node>
</root_node>
```

- For each attribute in the XML document define an `<attribute_node>` tag with the name attribute (name=) set to the attribute's name. The attribute_node elements are nested in their element_node (see Example 8-20).

Example 8-20 Adding attribute_node tags

```
<root_node>
<element_node name="Order">
   <attribute_node name="key">
   </attribute_node>
      <element_node name="Customer">
...
```

3. Mapping the relational content to XML

For both the SQL composition DAD and RDB _node DAD, the mapping is governed by instructions that appear alongside the structural tags in the second part of the DAD by specifying text_node and column tags.

SQL statement mapping DAD files

To map the relational data to XML we use the following constructs:

text_node

Specifies the text content of the element and the column data in a relational table for bottom-level element_nodes.

column tags

Specifies the DB2 column name that is to be used for the XML element or attribute.

To do the mapping we use the following technique:

- For each element at the lowest level, define `<text_node>` tags indicating that the element contains character data to be extracted from DB2 when composing the document (see Example 8-21).

Example 8-21 Adding text_nodes

```
<root_node>
<element_node name="Order">
   <attribute_node name="key">
   </attribute_node>
      <element_node name="Customer">
```

```
         <element_node name="Name">
            <text_node>
            </text_node>
         </element_node>
         <element_node name="Email">
            <text_node>
            </text_node>
         </element_node>
      </element_node>
      ...
```

- For each bottom-level element_node, define a `<column>` tag. These tags specify from which DB2 column (either an actual table column, or an assigned column name in the query) to extract data when composing the XML document. `<column>` tags are typically inside the `<attribute_node>` or the `<text_node>` tags (see Example 8-22). Remember, the columns defined here must be in the `<SQL_stmt>` SELECT clause when using the SQL statement mapping.

Example 8-22 Adding column tags

```
<root_node>
<element_node name="Order">
   <attribute_node name="key">
      <column name="order_key"/> <!-in an attribute node->
   </attribute_node>
      <element_node name="Customer">
         <element_node name="Name">
            <text_node>
               <column name="customer_name"/> <!-in a text node->
            </text_node>
         </element_node>
         <element_node name="Email">
            <text_node>
               <column name="customer_email"/>
            </text_node>
         </element_node>
      </element_node>
      <...
```

RDB_node DAD files

When using RDB_node mapping, you need to specify an RDB_node for each attribute_node to specify from which table and which column and query condition to use to get the data.

RDB_node

Specifies that we are doing RDB_node mapping for elements and attributes.

text_node

Specifies the text content of the element and the column data in a relational table for bottom-level element_nodes.

table tag

Specifies the table that the values are to be extracted from.

column tag

Specifies the column name in the table the value of the attribute or element is extracted from.

Attribute mapping

- When using RDB_node DAD files and we need to map an XML attribute, we specify the table name and column name having the required data using a `<table>` and `<column>` tag respectively. Both tags have to be specified inside an `<RDB_node>` tag, as shown in Example 8-23.

Example 8-23 Attribute mapping with RDB_node

```
<element_node name="Part">
   <attribute_node name="Key">
      <RDB_node>
         <table name="part_tab"/>
         <column name="part_key"/>
      </RDB_node>
   </attribute_node>
```

Element mapping using text_node

- You need to define an RDB_node for each text_node (representing an XML element), and specify from which table (using the `<table>` tag) and which column (using the `<column>` tag to get the data from.
- You can optionally specify a query condition (using a `<condition>` tag) to filter some of the data. Example 8-24 shows how to code the RDB_node inside a text_node.

Example 8-24 RDB_node in a text_node

```
<element_node name="ExtendedPrice">
   <text_node>
```

```
            <RDB_node>
               <table name="part_tab"/>
               <column name="price"/>
               <condition>price >2500.00</condition>
            </RDB_node>
         </text_node>
      </element_node>
```

4. Controlling the number of documents generated

All the elements and attributes that make up the XML document are specified within a so called root_node. Therefore you must add Add the <root_node> </root_node> tags to define the root element.

You must specify a child element under the root_node, which can only be used once. The element_node under the root_node is actually the root_node of the XML document.

- For SQL composition, the number of documents that a DAD produces can be controlled by the SQL statement in the DAD. The number of documents produced is equal to the number of rows grouped by the first grouping expression.
- For RDB_node, the number of documents produced can be controlled by the options supplied on the root element in the DAD.

Example 8-25 shows how to produce a single document by adding a highest level ‘grouping it all together’ <orders> tag.

Example 8-25 Producing a single XML document

```
<orders>
   <order id=“4711”>
      <signdate>2002-03-18</signdate>
      <amount>24000</amount>
   </order>
   <order id=“4712”>
      <signdate>2002-03-19</signdate>
      <amount>44000</amount>
   </order>
</orders>
```

When certain elements can occur multiple times at the same level within the XML document, you must specify the multi_occurrence attribute and set its value to yes. This is shown in Example 8-26 for the shipment element, as there can be more than one shipment per part.

Example 8-26 DAD file for an XML collection using SQL mapping

```
<root_node>
<element_node name="Order">
   <attribute_node name="key">
      <column name="order_key"/>
   </attribute_node>
      <element_node name="Customer">
         <element_node name="Name">
            <text_node><column name="customer_name"/></text_node>
         </element_node>
         <element_node name="Email">
            <text_node><column name="customer_email"/></text_node>
         </element_node>
      </element_node>
      <element_node name="Part">
         <attribute_node name="color">
            <column name="color"/>
         </attribute_node>
         <element_node name="key">
            <text_node><column name="part_key"/></text_node>
         </element_node>
         <element_node name="Quantity">
            <text_node><column name="quantity"/></text_node>
         </element_node>
         <element_node name="ExtendedPrice">
            <text_node><column name="price"/></text_node>
         </element_node>
         <element_node name="Tax">
            <text_node><column name="tax"/></text_node>
         </element_node>
         <element_node name="Shipment" multi_occurrence="YES">
            <element_node name="ShipDate">
               <text_node><column name="date"/></text_node>
            </element_node>
            <element_node name="ShipMode">
               <text_node><column name="mode"/></text_node>
            </element_node>
         </element_node>
      </element_node>
</element_node>
</root_node>
```

Note that there should not be more than one order that is returned by DB2, as we did not specify multi_occurrence=”yes” on the Order element. If that is the case, we have to change multi_occurrence to yes, and add an extra level to the document, just under the root_node, for example <orders> (note the ’s’ at the end).Example 8-27 shows how the option is specified.

Example 8-27 Using multi-occurrence

```
<root_node>
<element_node name=“orders”>
   <element_node name="Order" multi_occurrence=“yes”>
...
```

5. Outputting document header information

Header information such as XML declarations, DTD references, and processing instructions can be generated through statements in the DAD. Following is an example of how to do so (Example 8-28).

Example 8-28 Specifying header information

```
<prolog>?xml version“1.0”?</prolog>
<doctype>
   !DOCTYPE Order SYSTEM “orders.dtd”
</doctype>
```

6. Validating the generated documents

For both SQL composition and RDB_node mapping, it is possible to validate the generated documents against an XML Schema or a DTD. For XML document validation there are three options:

- Use the validation option in the DAD as follows:

  ```
  <dtdid>order.dtd</dtdid>
  <validation>yes</validation>
  ```

  DB2 XML Extender will perform the validation against the specified DTD stored in the file system, or stored in a special table called the DTD_REF table.

- Use the **`dvalidate()`** user-defined function (UDF), as follows:

  ```
  db2xml.dvalidate( doc, dtd )
  ```

- For Schema validation, the **`svalidate()`** UDF is available, as follows:

```
db2xml.svalidate( doc, xmlschema )
```

7. Transforming the generated documents

It is possible to apply further transformations to the generated documents, for example to convert an XML document into HTML.

There are a number of ways to do transformation.

- Place an XSL processing instruction in the header information.
- Use DB2 XML Extender supplied XSLT UDFs, as follows:

```
XSLTransformToClob( xmldoc, stylesheet, parameters, validate )
```

More information about transformation can be found in 9.2, “Validation” on page 259.

In the previous sections we only showed DAD fragments. Let us now look at two complete DAD files, one using SQL Composition DAD (Example 8-29),

Example 8-29 Full SQL composition DAD

```
<?xml version“1.0”?>
<!DOCTYPE DAD SYSTEM “dad.dtd”>
<DAD>
   <validation>no</validation>
   <Xcollection>
      <SQL_stmt>
         SELECT
            o.oid AS id,
            o.contractdate AS cdate,
            SUM(oi.orderitem) AS total,
         FROM orders AS o, orderItems AS oi
         WHERE oi.oid  o.oid AND status  ‘open’
         ORDER BY id;
      </SQL_stmt>
      <prolog>?xml version”1.0”?</prolog>
      <doctype>
         !DOCTYPE Order SYSTEM gorders.dtdh
      </doctype>
      <root_node>
         <element_node name=”order”>
            <attribute_node name=”id”>
            <column name=”id”/>
            </attribute_node>
         <element_node name=”signdate”>
            <text_node>
               <column name=”cdate”/>
            </text_node>
         </element_node>
         <element_node name=”amount”>
            <text_node>
               <column name=”total”/>
            </text_node>
         </element_node>
         </element_node>
      </root_node>
   </Xcollection>
</DAD>
```

and a complete DAD file using RDB_node mapping (Example 8-30).

Example 8-30 DAD file for an XML Collection using RDB node mapping

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "c:\dxx\dtd\dad.dtd>
<DAD>
   <dtdid>c:\examples\dtd\lineItem.dtd</dtdid>
   <validation>YES</validation>
<Xcollection>
   <prolog>?xml version="1.0"?</prolog>
   <doctype>!DOCTYPE Order SYSTEM "C:\examples\dad\order.dtd"</doctype>
<root_node>
   <element_node name="Order">
      <RDB_node>
         <table name="order_tab"/>
         <table name="part_tab"/>
         <table name="ship_tab"/>
         <condition>order_tab.order_key=part_tab.order_key AND
                    part_tab.part_key=ship_tab.part_key </condition>
      </RDB_node>
      <attribute_node name="Key">
         <RDB_node>
            <table name="order_tab"/>
            <column name="order_key"/>
         </RDB_node>
      </attribute_node>
      <element_node name="Customer">
         <element_node name="Name">
            <text_node>
               <RDB_node>
                  <table name="order_tab"/>
                  <column name="customer_name"/>
               </RDB_node>
            </text_node>
         </element_node>
         <element_node name="Email">
            <text_node>
               <RDB_node>
                  <table name="order_tab"/>
                  <column name="customer_email"/>
               </RDB_node>
            </text_node>
         </element_node>
      </element_node>
      <element_node name="Part">
         <attribute_node name="Key">
            <RDB_node>
               <table name="part_tab"/>
               <column name="part_key"/>
            </RDB_node>
         </attribute_node>
         <element_node name="ExtendedPrice">
            <text_node>
               <RDB_node>
                  <table name="part_tab"/>
                  <column name="price"/>
                  <condition>price >2500.00</condition>
               </RDB_node>
            </text_node>
         </element_node>
         <element_node name="Tax">
            <text_node>
               <RDB_node>
                  <table name="part_tab"/>
                  <column name="tax"/>
               </RDB_node>
            </text_node>
         </element_node>
         <element_node name="Quantity">
            <text_node>
               <RDB_node>
                  <table name="part_tab"/>
                  <column name="qty"/>
               </RDB_node>
            </text_node>
         </element_node>
         <element_node name="Shipment"multi_occurrence="YES">
            <element_node name="ShipDate">
               <text_node>
                  <RDB_node>
                     <table name="ship_tab"/>
                     <column name="date"/>
                     <condition>date >'1966-01-01 '</condition>
                  </RDB_node>
```

```
               </text_node>
            </element_node>
            <element_node name="ShipMode">
               <text_node>
                  <RDB_node>
                     <table name="ship_tab"/>
                     <column name="mode"/>
                  </RDB_node>
               </text_node>
            </element_node>
            <element_node name="Comment">
               <text_node>
                  <RDB_node>
                     <table name="ship_tab"/>
                     <column name="comment"/>
                  </RDB_node>
               </text_node>
            </element_node>
         </element_node><!--end of element Shipment-->
      </element_node><!--end of element Part -->
   </element_node><!--end of element Order -->
</root_node>
</Xcollection>
</DAD>
```

8.2.3 DB2 XML Extender publishing stored procedures

DB2 XML Extender provides four stored procedures to compose XML documents from relational data. The frequency with which you plan to update the XML document is a key factor in selecting the stored procedure that you should use.

- dxxGenXML()
- dxxGenXMLCLOB()
- dxxRetrieveXML()
- dxxRetrieveXMLCLOB()

dxxGenXML and dxxGenXMLCLOB

The *dxxGenXML* stored procedure constructs XML documents using data that is stored in relational tables, which are specified by the <Xcollection> element in the DAD file. This stored procedure inserts each XML document as a row into a results table. You can also open a cursor on the results table and fetch the result set. The results table should be created by the application and should have one VARCHAR, CLOB, XMLVARCHAR, or XMLCLOB column.

Additionally, if the DAD file contains a <validation> element with a value of YES, DB2 XML Extender adds the DXX_VALID column (data type INTEGER) to the results table (if the DXX_VALID column is not in the table yet). DB2 XML Extender inserts a value of 1 for a valid XML document, and 0 for an invalid document. into the DXX_VALID column.

The dxxGenXML stored procedure also allows to specify the maximum number of rows that are to be generated in the results table. This can shorten processing time. The stored procedure returns the actual number of rows in the table, along with any return codes and messages.

The declaration for the dxxGenXML stored procedure follows:

```
dxxGenXML(DAD CLOB(100K)DAD,               /*input */
          resultTabName char(UDB_SIZE ),    /*input */
          overrideType integer,             /*input */
          override varchar(1024),           /*input */
          maxRows integer,                  /*input */
          numRows integer,                  /*output */
          returnCode long,                  /*output */
          returnMsg varchar(1024))          /*output */
```

Example 8-31 shows an example of how to call the dxxgenxml stored procedure. It is un against the SALES_DB sample XML Extender database.

Example 8-31 Sample usage of dxxGenXML()

```
call
db2xml.dxxgenxml(db2xml.xmlclobfromfile('C:\IBM\SQLLIB\samples\db2xml\dad\
                                         getstart_xcollection.dad'
                                        ),
                 'result_tab',
                 0,
                 '',
                 200,
                 ?,
                 ?,
                 ?)
```

Let’s walk though the call statement’s parameters. The first thing to notice it that we need to use a DB2 Extender function (**XMLCLOBFROMFILE**) to include the DAD file from the file system. This function takes a file and converts it into a CLOB. result_tab is the table the XML document is to be written to. Next we assign an override type parameter, and the override text. More details can be found in “Dynamically overriding values in the DAD file” on page 242. We then provide the number of records we want to return. Finally we assign place holders for the three output parameters.

If we run a select statement on the result_tab table, DB2 returns the following result (Example 8-32):

Example 8-32 Result of dxxGenXML

```
<?xml version="1.0"?>
<!DOCTYPE Order SYSTEM "C:\Program
Files\IBM\SQLLIB\samples\db2xml\dtd\getstart.dtd">
<Order key="1">
  <Customer>
    <Name>American Motors</Name>
    <Email>parts@am.com</Email>
  </Customer>
  <Part color="black ">
    <key>68</key>
    <Quantity>36</Quantity>
    <ExtendedPrice>34850.16</ExtendedPrice>
    <Tax>6.000000E-002</Tax>
    <Shipment>
      <ShipDate>1998-08-19</ShipDate>
      <ShipMode>BOAT  </ShipMode>
    </Shipment>
    <Shipment>
      <ShipDate>1998-08-19</ShipDate>
      <ShipMode>AIR   </ShipMode>
    </Shipment>
    <Shipment>
      <ShipDate>1998-08-19</ShipDate>
      <ShipMode>BOAT  </ShipMode>
    </Shipment>
    <Shipment>
      <ShipDate>1998-08-19</ShipDate>
      <ShipMode>AIR   </ShipMode>
    </Shipment>
  </Part>
  <Part color="red   ">
    <key>128</key>
    <Quantity>28</Quantity>
    <ExtendedPrice>38000.00</ExtendedPrice>
    <Tax>7.000000E-002</Tax>
    <Shipment>
      <ShipDate>1998-12-30</ShipDate>
      <ShipMode>TRUCK </ShipMode>
    </Shipment>
    <Shipment>
```

```
        <ShipDate>1998-12-30</ShipDate>
        <ShipMode>TRUCK </ShipMode>
      </Shipment>
    </Part>
  </Order>
```

If your document is only generated occasionally, use the dxxGenXML stored procedure to compose the document. You do not have to enable an XML collection to use this stored procedure. It uses a DAD file as input.

dxxRetrieveXML dxxRetrieveXMLCLOB

The *dxxRetrieveXML* stored procedure works the same way as the dxxGenXML stored procedure, except that it uses the name of an enabled XML collection instead of a DAD file as input. When an XML collection is enabled, a DAD file is stored in the XML_USAGE table. Therefore, DB2 XML Extender retrieves the DAD file, and uses it to compose the document in the same way as the dxxGenXML stored procedure.

The definition of the dxxRetrieveXML stored procedure is as follows:

```
dxxRetrieveXML(collectionName char(UDB_SIZE ),      /*input */
               resultTabName char(UDB_SIZE ),       /*input */
               overrideType integer,                /*input */
               override varchar(1024),              /*input */
               maxRows integer,                     /*input */
               numRows integer,                     /*output */
               returnCode long,                     /*output */
               returnMsg varchar(1024))             /*output */
```

Let us now take a look at an example of composing an XML document using the dxxRetrieveXML stored procedure. We use the SALES_DB database again.

If you ran the example for the dxxGenXML stored procedure before, you need to delete the records from the result_tab. You can do so from the DB2 Command Window using the following SQL statement:

```
db2 delete from result_tab where doc is not null
```

Now that the results table is empty we can prepare the database to run this example. As mentioned before, dxxRetrieveXML requires you to have an XML collection. To enable an XML collection, run the following commend from a DB2 Command Window:

```
dxxadm enable_collection sales_db Orders
       [install directory]\samples\db2xml\dad\getstart_xcollection.dad
```

```
        -l resident -p bartr
```

The collection name is ‘Orders’ (the name is case sensitive).

Example 8-33 shows how to invoke the dxxRetreiveXML() stored procedure to generate an XML document based on an XML Extender collection name.

Example 8-33 Using dxxRetrieveXML

```
call
db2xml.dxxRetrieveXML('Orders','result_tab',0,'NO_OVERRIDE',1000,?,?,?)
```

Notice that the call statement for dxxRetrieveXML is similar to the dxxGenXML stored procedure call statement. In this case we do not have to cast the DAD file to an XMLCLOB because the DAD file is stored in the DB2 database when it XML Extender collection enabled. dxxRetrivevXML manages to find the associated DAD file via the name of the XML collection (Orders). result_tab is the table the XML documents are to be written to. Next, we indicate that we do not want to override a parameter and override text. Overwrites are described in more detail in “Dynamically overriding values in the DAD file” on page 242. We then provide the number of records we want to return. Finally, we assign place holders for the three output parameters.

If we now run a select statement on the result_tab table, we return the following result (Example 8-34)

Example 8-34 Result from dxxRetrieve() retrieved from the result_table

```
<?xml version="1.0"?>
<!DOCTYPE Order SYSTEM
           "C:\ProgramFiles\IBM\SQLLIB\samples\db2xml\dtd\getstart.dtd">
<Order key="1">
  <Customer>
    <Name>American Motors</Name>
    <Email>parts@am.com</Email>
  </Customer>
  <Part color="black ">
    <key>68</key>
    <Quantity>36</Quantity>
    <ExtendedPrice>34850.16</ExtendedPrice>
    <Tax>6.000000E-002</Tax>
    <Shipment>
      <ShipDate>1998-08-19</ShipDate>
      <ShipMode>BOAT  </ShipMode>
    </Shipment>
```

```
      <Shipment>
        <ShipDate>1998-08-19</ShipDate>
        <ShipMode>AIR   </ShipMode>
      </Shipment>
      <Shipment>
        <ShipDate>1998-08-19</ShipDate>
        <ShipMode>BOAT  </ShipMode>
      </Shipment>
      <Shipment>
        <ShipDate>1998-08-19</ShipDate>
        <ShipMode>AIR   </ShipMode>
      </Shipment>
    </Part>
    <Part color="red   ">
      <key>128</key>
      <Quantity>28</Quantity>
      <ExtendedPrice>38000.00</ExtendedPrice>
      <Tax>7.000000E-002</Tax>
      <Shipment>
        <ShipDate>1998-12-30</ShipDate>
        <ShipMode>TRUCK </ShipMode>
      </Shipment>
      <Shipment>
        <ShipDate>1998-12-30</ShipDate>
        <ShipMode>TRUCK </ShipMode>
      </Shipment>
    </Part>
  </Order>
```

If your document is updated frequently, use the dxxRetrieveXML stored procedure to compose the document. Because the same tasks are repeated, improved performance is important.

Dynamically overriding values in the DAD file

For dynamic queries we can use two parameters in the dxxGenXML stored procedure to override conditions in the DAD file. The parameters we use to override conditions are the overrideType and override parameters. These parameters have the following values and rules:

overrideType

This parameter is a required input parameter (IN) that flags the type of the override parameter. overrideType has the following values:

- NO_OVERRIDE Specifies not to override a condition in the DAD file

- SQL_OVERRIDE: Specifies to override a condition in the DAD file with an SQL statement
- XML_OVERRIDE: Specifies to override a condition in the DAD file with RDB_node mapping using an XPath-based condition

When calling a composition stored procedure, the override type parameter has to be set to 0 for NO_OVERRIDE, 1 for SQL_OVERRIDE and 2 for XML_OVERRIDE.

override

This parameter is an optional input parameter (IN) that specifies the override condition for the DAD file. The input value syntax corresponds to the value specified on the overrideType.

- If you specify NO_OVERRIDE, the input value is a NULL string.
- If you specify SQL_OVERRIDE, the input value is a valid SQL statement. If you use SQL_OVERRIDE and an SQL statement, you must use the SQL mapping scheme in the DAD file. The input SQL statement overrides the SQL statement specified by the `<SQL_stmt>` tag in the DAD file.
- If you use XML_OVERRIDE, the input value is a string which contains one or more expressions. If you use XML_OVERRIDE and an expression, you must use the RDB_node mapping scheme in the DAD file. The input XML expression overrides the RDB_node condition specified in the DAD file. The expression uses the syntax shown in Figure 8-4 for the XML expression. It is worth noting that you cannot override the join condition specified in the top RDB_node. You can only override the condition specified in the element node (that matches the specified path).

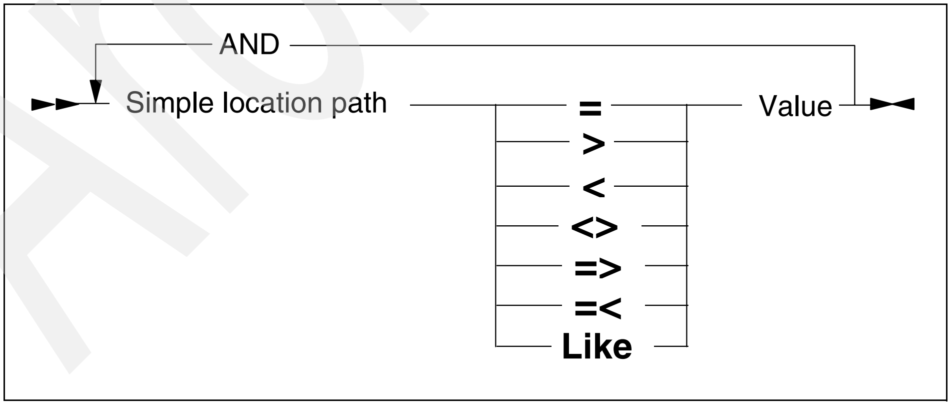


Figure 8-4 XML_OVERRIDE expression syntax

This expression is made up of the following parts:

Simple location path

This is a simple location path using the syntax defined by XPath.

Operators

A space can be used to separate the operator from the other parts of the expression.

Value

This can be a numeric value or a single quoted string.

When the XML_OVERRIDE value is specified, the condition for the RDB_node in the text_node or attribute_node that matches the simple location path is overridden by the specified expression. XML_OVERRIDE is not completely XPath compliant. The simple location path is only used to identify the element or attribute that is mapped to a column.

Similarly, you can use override when you are using the dxxRetrieveXML stored procedure.

Let’s take a look at an example using an SQL_OVERRIDE. We use the SALES_DB that already has an enabled collection. The override principle is the same for all the XML Extender composition (or publishing) stored procedures.

Example 8-35 shows how to override the SQL statement stored in the DAD file which lists prices greater than 20000, to only show prices greater than 35000. The only difference to the dxxGenXML statement in Example 8-31 on page 238 is that the “overrideType” parameter is set to 1, and the statement is included in the “override” parameter.

Example 8-35 dxxGenXML using SQL_OVERRIDE

```
call
 db2xml.dxxGenXML(
   db2xml.xmlclobfromfile('C:\Program Files\IBM\SQLLIB\samples\
                             db2xml\dad\getstart_xcollection.dad'),
   'result_tab',
   1,
   'SELECT o.order_key, customer_name, customer_email, p.part_key,color,
           quantity, price, tax, ship_id, date, mode
      FROM order_tab o, part_tab p,
         table(select substr(char(timestamp(generate_unique())),16) as ship_id,
                      date, mode, part_key
                 from ship_tab) s
      WHERE o.order_key = 1 and p.price > 35000 and
            p.order_key = o.order_key and
            s.part_key = p.part_key
```

```
    ORDER BY order_key, part_key, ship_id',
    1000,?,?,?)
```

The result returned from this statement looks like Example 8-36:

Example 8-36 Result of dxxGenXML() with override

```
<?xml version="1.0"?>
<!DOCTYPE Order SYSTEM "C:\Program Files\IBM\SQLLIB\samples\db2xml\
                         dtd\getstart.dtd">
<Order key="1">
  <Customer>
    <Name>American Motors</Name>
    <Email>parts@am.com</Email>
  </Customer>
  <Part color="red   ">
    <key>128</key>
    <Quantity>28</Quantity>
    <ExtendedPrice>38000.00</ExtendedPrice>
    <Tax>7.000000E-002</Tax>
    <Shipment>
      <ShipDate>1998-12-30</ShipDate>
      <ShipMode>TRUCK </ShipMode>
    </Shipment>
  </Part>
</Order>
```

Notice that the returned XML document only shows orders with a price over 35000.

8.2.4 Writing your own code to publish data as XML

DB2 provides many facilities to publish XML documents, either through its native SQL/XML publishing functions, or via DB2 XML Extender. Using these facilities, you should be able to build whole XML documents, or XML fragments. Unless there is an XML structure you are unable to produce using the supplied functions, it is best to avoid building your own publishing applications.

8.3 Best practices

The preferred tool for XML publishing is using SQL/XML. Generally speaking, it is more flexible and versatile than the DB2 XML Extender publishing functions.

However, if for some reason you cannot use SQL/XML to get the job done, you can try to accomplish your goal using XML Extender, and as a last resort, if XML Extender is also not up to the job, write your own application to publish relational data as XML.

8.3.1 SQL/XML

There are a few limitations in SQL/XML that you should be aware of, and that can determine whether or not to use these functions during XML publishing. The major limitations we found with SQL/XML during the writing of this chapter are:

- On the distributed platforms (Linux, Unix and Windows) using DB2 V8 Fixpak 4, when using the XMLAGG function we are unable to use a subselect statement. This is supported on the zSeries® platform. Therefore, we are unable to produce Example 8-11 on page 216 on DB2 for LUW.
- When writing a SQL/XML statement and prefixing it with the DB2 command, you must escape double quotation marks bordering element and attribute names. If you run the statement without escaping the quotes, DB2 returns all element and attribute names in upper case. Example 8-37 shows a statement using correct escaping of the double quotes.

  Scripts produce the correct result without the escape characters.

Example 8-37 Escaping element and attribute names

```
SELECT xml2clob(
   XMLELEMENT (NAME \"EmployeeSalary\",
      XMLATTRIBUTES (e.empno AS \"id\"),
         XMLELEMENT (NAME \"Firstname\", e.firstnme),
         XMLELEMENT (NAME \"Lastname\", e.lastname),
         XMLELEMENT (NAME \"TotalSalary\",(e.salary+e.bonus+e.comm)
         )
      )
   )
FROM employee e WHERE SEX  = 'F'
```

8.3.2 XML Extender

When publishing using DB2 XML Extender, we need to be diligent and plan our publishing method based on the recommendations in the section 8.2.3, “DB2 XML Extender publishing stored procedures” on page 237. This governs which stored procedures you will use to generate your document.

9

Additional XML functionality

This chapter describes additional XML functionality, such as:

- Transformation of XML documents using XSL and SAX
- Validation of XML documents and DAD files
- Importing and exporting XML documents using DB2 XML Extender UDFs

9.1 Transformation

In general, there are two commonly used techniques to transform XML documents. The first one is to use the Extensible Stylesheet Language (XSL). The second one is to use the Simple API for XML, better know as SAX.

9.1.1 XSL, stylesheets and transformation

A stylesheet, in a generic sense, guides the transformation of data from one format to another. Extensible Stylesheet Language or XSL is the language for expressing stylesheets in XML. XSL helps in restructuring, restyling, or converting XML data to another form.

XSL consists of three parts:

- XSLT, the language for transforming XML documents based on rule matching
- XPath, an expression language used by XSLT for addressing parts of an XML document
- XSL Formatting Objects, an XML vocabulary for specifying formatting semantics

Figure 9-1 on page 249 gives an overview of how XSL works.

The source XML document that needs to be transformed is supplied to an XSLT engine. The XSLT engine refers to the transformation rules specified in XSLT and identifies the XML nodes from the source XML document. It then performs actions/transformations specified in the action part of the transformation rules on these identified XML nodes. The application of actions on all such XML nodes of the source XML document results in the transformed document. This transformed document can be another XML document or any other type of document.

If this transformed document needs to be formatted, it is further passed to the formatting engine. The formatting engine reads the formatting rules from the supplied formatting document written in XSL-FO and applies the rules to the transformed document. The formatting engine can use other resources such as images and fonts to create the formatted document.

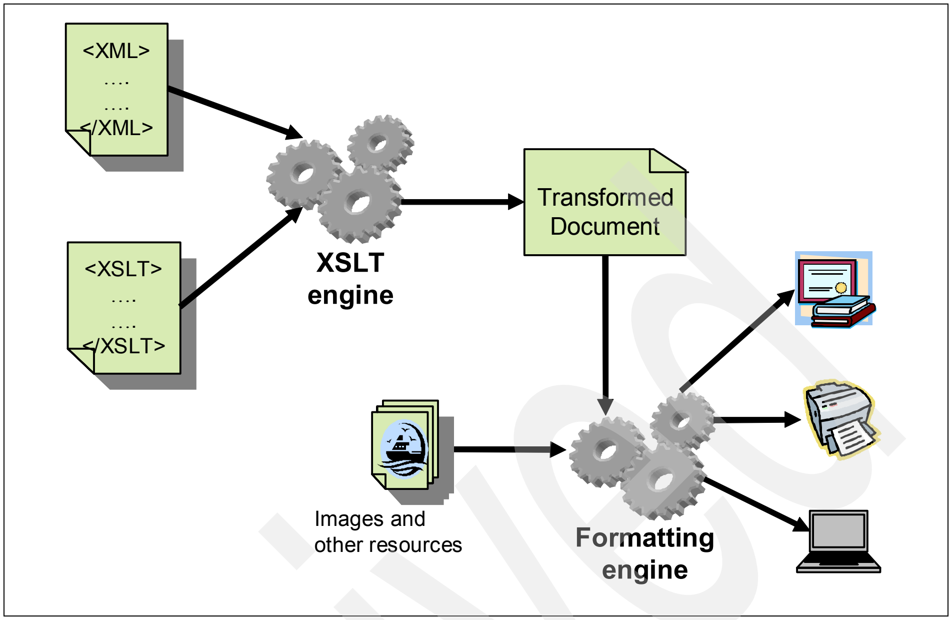


Figure 9-1 XSL architectural overview

In the database world, XSL is primarily used for restructuring or restyling XML data. When working with databases you rarely come across situations wherein you need to apply different formatting mechanisms to the XML data. Hence in this section we only deal with XSLT.

XSLT uses XML syntax to define the transformation rules (thus, an XSLT document is a valid XML document). An XSLT document contains one or more *template rules*. A template rule consists of a pattern and an action. The *pattern* specifies the XML entities or the nodes to which the template rule applies. A pattern is expressed using XPath expressions. The *action* part of the template rule specifies the action to be taken when the pattern matches. The action can be as simple as outputting some markup, add new data, and/or copy some data out of the source XML document, or it can be a more complex action.

A typical XSL stylesheet is shown in Example 9-1:

Example 9-1 Simple XSL stylesheet

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet>
     <!-- other directives -->
     <xsl:template match="title">
           <xsl:apply-templates/>
```

```
            <!-- Any other Action element -->
        </xsl:template>
             …
             …
</xsl:stylesheet>
```

The XSLT engine uses both the XML document to be transformed, and the XSLT stylesheet as input. The engine reads through the XML document and tries to match the patterns defined in the template rules against each XML node it comes across. If a pattern matches, its associated action is executed. This is known as invoking the template rule.

A simple XSL stylesheet

Let us take a sample XML snippet and try to write XSLT rules to transform this snippet into another XML snippet of our choice. Suppose that an insurance company ITSO Insurance, Inc. stores the details of its agents in XML in the following format (Example 9-2):

Example 9-2 Source XML document

```
<agent id="1234">
      <name> James Bont </name>
      <area> London </area>
</agent>
```

Now an insurance brokerage firm XYZ Ltd. that caters to many insurance companies wants to collect details about all the agents of all the companies. However, the brokerage firm XYZ Ltd. stores the details of each insurance agent in the following format (Example 9-3):

Example 9-3 Target XML document

```
<insuranceAgent name="James Bont" location=" London "/>
```

To transform the XML data stored in ITSO Insurance, Inc., to the format XYZ Ltd. expects, we can use the following simple XSL stylesheet (Figure 9-4 on page 251):

Example 9-4 Simple XSL stylesheet

```
1  <?xml version="1.0"?>
2  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                                version="1.0">

3    <xsl:template match="/agent" >
4          <xsl:element name="insuranceAgent">
5                 <xsl:apply-templates/>
6            </xsl:element>
7     </xsl:template>

8     <xsl:template match="/agent/name">
9          <xsl:attribute name="name">
10                <xsl:apply-templates/>
11         </xsl:attribute>
12   </xsl:template>

13   <xsl:template match="/agent/area">
           <xsl:attribute name="location">
                  <xsl:apply-templates/>
           </xsl:attribute>
14         </xsl:template>
```

Simple stylesheet dissected

Let us now look at Example 9-4, and dissect it line by line to understand how XSL transformation works.

1 The first line states that the XSL Stylesheet is an XML 1.0 document.

2 An XSL stylesheet contains exactly one `<xsl:stylesheet>` element and this encloses the entire stylesheet. In our example it appears on the second line.

3 One or more `<xsl:template>` elements can appear in a stylesheet. The first template rule has the XPath expression “/agent” as its matching criteria. This expression addresses all the `<agent>` element nodes of the source XML document. For more information about XPath, you can go to the following Web site:

  http://www.w3.org/TR/xpath

4 Specifies the action to be taken when an `<agent>` node is encountered in the source XML document.
As per our transformation requirements (to transform the XML document of Example 9-2 on page 250 into Example 9-3 on page 250), we have to create a new XML element, `<insuranceAgent>`. Then we have to extract the values of `<name>` and `<area>` child elements of the `<agent>` element, and copy them into the attributes “name” and “location” respectively of the new `<insuranceAgent>` element.

To create our new <insuranceAgent> element., we use an XSL directive <xsl:element>.

5 A very interesting thing is happening in this line. By using <xsl:apply-templates/>, we are asking the XSLT engine to continue applying template rules recursively down the hierarchy. If we do not tell this to the XSLT engine it just does not continue to traverse any children further, and no templates that are targeted for the children of this node will ever be invoked. By now you must have started wondering how this template rule ever got applied, as it is a child of the root node, and we never asked the root node to apply its template. In fact, we did not define a template rule targeted for the root node at all! The answer is the following built-in template rule is defined for the root node:

```
<xsl:template match="*|/">
     <xsl:apply-templates/>
</xsl:template>
```

Thus by saying <xsl:apply-templates/> we are ensuring that the XSLT engine tries to match patterns for <name> and <area> elements as well.

6 After templates are applied to all the children of this node, this line is executed, and the closing tag for the <insuranceAgent> element is created.

7 This line signifies the end of the first template rule body.

8 Here we encounter the start of a new template rule. This template is targeted for the <name> element, a child of the <agent> element.

9 Create the “name” attribute.

10 We ask the XSLT engine to apply the templates down the hierarchy again. In our case, the node down the hierarchy from the <name> node, is a “text” node. We have a built-in template rule for “text” nodes as well:

```
<xsl:template match="text() | @*">
     <xsl:value-of  select="."/>
</xsl:template
```

This rule also caters to any attribute value. The action part of this built-in rule specifies to write the contents of a “text” node to the output XML document. So with this, we populate the value of the attribute node that we created in the previous line.

11 This line completes the creation of the XML ”name” attribute node.

12 This line signifies the end of second template rule.

13 The following lines are doing the same as described above; the only change here is that the name of the attribute node being created, and the name of the source element node are different, location and area respectively.

14 This line indicates the end of stylesheet.

Running the sample

To run the sample we just discussed, you need an XSLT engine. Apache's Xalan is one of the more widely used XSLT engines. It implements XSLT 1.0 and XPath 1.0. It can be used from the command line, in an applet or a servlet, or as a module in a program. Please refer to the Xalan documentation at:

`http://xml.apache.org/xalan-j/` or
`http://xml.apache.org/xalan-c/index.html`

for further information.

9.1.2 Why transform?

While dealing with XML in databases, there are many situations wherein the XML data has to be restructured. In some cases, the output format has to change as well, for example from an XML document into HTML. In this section we look at some of the candidate scenarios that require such transformations.

Before storing the data into relational tables

In real life scenarios, data comes from various sources and in various formats. No matter how much we try to standardize the structure of data being exchanged using XML, we still find minor variations in the "standard" structures of XML data, from one source to another. Sometimes, even though the XML data from various sources conforms to one standard schema, we come across situations where the structure of the XML data stored in the host database system differs slightly from the standard for various reasons. In such cases we have to restructure the incoming XML documents to conform to the host database schema so that it can be stored.

In some other cases, the database system imposes some restrictions on the structure of incoming XML data so that it can be decomposed/shredded before storing the data in relational tables. DB2 XML Extender uses DAD files to guide the shredding of XML data into relational tables. Prior to DB2 V8.1 FixPak3 or V7 FixPak 11, one of the restrictions imposed by the DAD file was that the incoming XML document should not contain two elements with the same names, even though their XPath expressions are different. For example, consider the XML snippet shown in Example 9-5 on page 254.

Example 9-5   Sample XML file

```
<insurancePolicy id="1234">

   <insurer>
      <name> Bertram Zooster </name>
      <address> Droney Club </address>
   </insurer>

   <nominee>
       <name> Jeevey </name>
       <relationToInsurer> valet </relationToInsurer>
   </nominee>

   <policyDetails>
      ...
   </policyDetails>

</insurancePolicy >
```

The preceding piece of XML contains two `<name>` elements. This is perfectly valid in XML, since the two elements appear under different parent nodes. However, DB2 XML Extender considers them as duplicate elements. In this case we have to transform the input XML document before passing it to DB2 XML Extender. The piece of code given in Example 9-6 on page 255 shows the restructured XML data. Note that the two `<name>` elements have been renamed as `<insurance_name>` and `<nominee_name>`.

Example 9-6 Transformed XML document

```
   <insurancePolicy id=”1234”>

      <insurer>
         <insurer_name> Bertram Zooster </insurer_name>
         <address> Droney Club </address>
      </insurer>

      <nominee>
          <nominee_name> Jeevey </nominee_name>
          <relationToInsurer> valet </relationToInsurer>
      </nominee>

      <policyDetails>
         ...
      </policyDetails>

</insurancePolicy >
```

After publishing the data from relational tables

To retrieve data stored in relational tables and publish it as XML, we can use both SQL/XML and DB2 XML Extender. SQL/XML statements in case of SQL/XML, or DAD files in case of DB2 XML Extender, guide the (re)construction of the XML data from the relational data. However, due to certain limitations, such as those imposed by the DAD, as we discussed earlier, we end up creating XML documents that do not conform to the schema the client application is expecting. In such cases, we can use XSL transformation to restructure the output XML document.

Also in some cases, we need to present the XML data to a Web client. In this case we can also use XSL to transform the XML document into HTML just before returning the results to the Web client.

DB2 XML Extender provides two user-defined functions that perform XSL transformation on an XML document that is stored as a CLOB object. These help to transform XML document objects from within the DB2 environment.

- **`XSLTransformToClob()`** reads an XML document as a CLOB locator and a stylesheet as a CLOB or from a file, and returns the document as a CLOB.
- **`XSLTransformToFile()`** reads an XML document as a CLOB and a stylesheet as a CLOB or from a file. The XSLTransformToFile() user-defined function(UDF) then writes the results from the stylesheet and XML document

into a file. When a directory and a file extension are given as parameters, the UDF will create a file with a unique filename in this directory.

9.1.3 Transforming XML documents with SAX

If your transformation is simple and can be performed while reading through the document from start to finish, then you may be able to write a simple SAX program to perform the transformation. The advantage of SAX is that it is generally faster than XSLT and, because it works with documents linearly, it can be used with arbitrarily large documents.

For example, DAD documents do not allow a leaf element type to be mapped differently in different contexts. Unfortunately, this is a common occurrence. For example, an XML schema that describes bibliographies may have several element types, such as Book, Article, and Paper, that all have the Title element type as a child. To work around this limitation, you only need to change the name of the Title element type to a context-dependent name, such as BookTitle, ArticleTitle, and PaperTitle. Each of these can then be mapped separately.

One way to do this is to create a SAX XMLFilter that changes the names. For example, the NameChanger sample program takes a list of element type names to change and a corresponding list of new names to change into. The list of names to change provides both parent and child names so that NameChanger can change child names on a per-parent basis.

> **Note:** An XMLFilter is a SAX program that serves as both a SAX application and a SAX parser. As a SAX application, it implements the ContentHandler interface and listens for SAX events. As a SAX parser, it implements the XMLReader interface and fires new SAX events. What is important about XMLFilters is that they are not required to fire the same events that they receive. Thus, they can be used to filter XML documents.

The following code fragment (Example 9-7 on page 257) is from the NameTester application. This is a sample application that uses the NameChanger application to change the names in an XML document depending on their context. This code accepts the name of XML input and output files, and the names of Java properties files containing the element type and attribute names to change. The NameTester and NameChanger applications are available for download. See Appendix F, “Additional material” on page 655 for details.

Example 9-7 NameTester code snippet

```
public void run(String oldXMLFilename, String newXMLFilename,
                String elemFilename, String attrFilename)
      throws Exception
{
 XMLReader   xmlReader;
 NameChanger changer;
 // 1. Get a new XMLReader and set namespace processing off.
 xmlReader = getXMLReader();
 xmlReader.setFeature("http://xml.org/sax/features/namespaces", false);
 xmlReader.setFeature("http://xml.org/sax/features/namespace-prefixes",
                      true);
 // 2. Get a new NameChanger and set the ContentHandler to this
 // NameTester object.
 changer = new NameChanger(xmlReader);
 changer.setContentHandler(this);
 // 3. Set the new element type and attribute names.
 changer.setElementTypeNames(getProperties(elemFilename));
 changer.setAttributeNames(getProperties(attrFilename));
 // 4. Open the output file for writing, parse the input file, and
 //    close the output file.
 writer = new FileWriter(newXMLFilename);
 changer.parse(new InputSource(new FileInputStream(oldXMLFilename)));
 writer.close();
}
```

The code instantiates an XMLReader (1), instantiates a NameChanger and hooks this up to the XMLReader (2), sets the names to change (3), and opens a Writer over the output file and calls the parse method on the XMLReader (4).

The actual work is done in the NameChanger code. This simply passes events to the ContentHandler (in this case, the NameTester application) except that it checks if the current element type name or attribute name is supposed to be changed when that element type or attribute occurs in the current parent.

For example, here is the code for startElement (Example 9-8 on page 258). Notice that it calls getNewQName and getNewAttributes to get the (possibly) changed names, then calls super.startElement to pass the startElement event on to the NameTester.

Example 9-8 startElement in NameChanger

```
public void startElement(String uri, String localName, String qName,
                         Attributes attrs)
   throws SAXException
{
   String     newQName;
   Attributes newAttrs;
   // Get the new QName. This may be the same as the
   // existing QName.
   newQName = getNewQName(qName);
   // Get attributes that use the new names. These
   // may use existing names.
   newAttrs = getNewAttributes(qName, attrs);
   // Pass on the startElement event with the new names.
   super.startElement(uri, "", newQName, newAttrs);
   // Push the local name of the current element onto the stack.
   parentStack.push(qName);
}
```

The ContentHandler methods in NameTester simply write the (possibly) new elements, attributes, and PCDATA to the output document. The startElement method is shown in Example 9-9.

Example 9-9 startElement in NameTester

```
public void startElement (String uri, String localName, String qName,
                          Attributes attrs)
  throws SAXException
{
   int i;
   // Start the element.
   try
   {
      writer.write('<');
      writer.write(qName);
      // Append the attributes. Note that this includes xmlns attributes.
      for (i = 0; i < attrs.getLength(); i++)
      {
         writer.write(' ');
         writer.write(attrs.getQName(i));
         writer.write("=\"");
         appendEscapedString(attrs.getValue(i));
         writer.write('"');
      }
      // Close the element.
```

```
         writer.write('>');
      }
      catch (IOException io)
      {
         throw new SAXException(io);
      }
   }
```

SAX applications can also be used to do things like inserting default values for elements required by the database but not present in the XML document, changing attributes to child elements, and combining element values, such as combining the values of a FirstName element and a LastName element into a Name element.

9.2 Validation

Validation of an XML document is the process of verifying if the document conforms to a given “schema.” An XML document is said to be valid, if its structure satisfies all the constraints mandated by the corresponding schema. The XML schema dictates the structure of an XML document. The schema for XML documents is either written in Document Type Definition language (DTD), or in XML Schema Definition language (XSD). XSD is becoming more popular because of its superior expressiveness over of DTD.

Generally, XML documents are validated before storing them in the database, or after publishing the stored data. Though DB2 does not impose any restriction on storing XML documents that are not valid, it may affect the working of the applications that rely on this data. Hence, unless you are storing XML documents for archival purposes, it is recommended that you first validate them before storing them into DB2.
In cases where you are sure that the XML document being stored is valid, or you expect validation to severely impact performance, you may consider the option of storing or publishing without validation.

9.2.1 Validating XML documents using UDFs

DB2 XML Extender offers two user-defined functions (UDFs) which validate XML documents against either an XML Schema or a DTD. These functions cannot only be used to validate XML documents during storing and publishing but can also be invoked at any other time if needed. The functions are:

- **db2xml.svalidate()**: Validates an XML document against a specified XSD
- **db2xml.dvalidate()**: Validates an XML document against a specified DTD

svalidate() function

This function validates an XML document against a specified schema written in XSD (or the schema named in the XML document). The UDF returns 1 if the document is valid, or 0 if not. This function assumes an XML document and a schema exist, either in the file system, or as a CLOB in DB2.

Before executing the SVALIDATE() function, ensure that XML Extender is enabled for your database. If the XML document fails the validation, an error message is written to the XML Extender trace file, provided that tracing is enabled before executing the SVALIDATE() function.

The function can be invoked as:

- `db2xml.svalidate(xmlObj, schemaDoc)`
- `db2xml.svalidate(xmlObj)`

Table 9-1 lists the parameters and their allowed types for this function

Table 9-1 SVALIDATE parameters

Parameter	Data type	Description
xmlObj	VARCHAR(256) CLOB (2G)	Fully qualified file path of the XML document to be verified CLOB containing the XML document to be verified
schemaDoc *(optional parameter)*	VARCHAR(256) CLOB (2G)	Fully qualified file path of the schema document CLOB containing the schema to verify against

dvalidate() function

This function validates an XML document against a specified DTD (or the DTD named in the XML document). The UDF returns 1 if the document is valid, or 0 if not. This function assumes an XML document and a schema exist in the file system, or as a CLOB in DB2.

Before executing the DVALIDATE() function, ensure that XML Extender is enabled for your database. If the XML document fails the validation, an error message is written to the XML Extender trace file)provided that the trace was enabled before executing the SVALIDATE() command).

The function can be invoked as:

- `db2xml.dvalidate(xmlObj, dtdDoc)`
- `db2xml.svalidate(xmlObj)`

Table 9-2 lists the parameters and their allowed types for this function.

Table 9-2 DVALIDATE parameters

Parameter	Data type	Description
xmlObj	VARCHAR(256)	Fully qualified file path of the XML document to be verified
	CLOB (2G)	CLOB containing the XML document to be verified
dtdDoc *(optional parameter)*	VARCHAR(256)	Fully qualified file path of the DTD document
	CLOB (2G)	CLOB containing the DTD document (either from the DTD_REF table or from a regular table)

Automatic XML document validation against a DTD

When you are using DB2 XML Extender functionality to store the XML documents, either as an XML column, or an XML collection, you can automatically validate the documents without using the previously mentioned UDFs. You can do so by specifying `YES` for the `<validation>` tag in the DAD file. To have a document validated when it is stored into DB2, you must specify the DTD you want to validate against within the `<dtdid>` element or in the `<!DOCTYPE>` specification in the original document.
To have a document validated when it is composed from an XML collection in DB2, you must specify a DTD within the `<dtdid>` element or within the `<!DOCTYPE>` element in the DAD file (Example 9-10).

Example 9-10 Specifying validation during composition

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DAD SYSTEM "C:\SQLLIB\samples\db2xml\dtd\dad.dtd">
<DAD>
   <validation>YES</validation>
   <Xcollection>
     .....
```

DB2 XML Extender does not support the validation of data after it has been inserted into DB2.

9.2.2 Validating the DAD file

The `<!DOCTYPE ...>` tag of the DAD file contains the location of a DTD file against which the DAD document itself is to be validated. The XML parser uses this DTD to perform *syntactic validation* of the DAD file.

DAD Checker is a tool that *verifies the semantics* of a DAD file and provides error messages and recommendations for correcting the DAD. DAD Checker is available for free and can be downloaded from DB2 XML Extender Web site that can be found at:

```
http://www-306.ibm.com/software/data/db2/extenders/xmlext/index.html
```

DAD Checker is a Java application that is started from the command line. When invoked, it produces two output files that contain errors, warnings and success indicators. The two files are equivalent.
One is a plain text file that you use to check for errors or warnings, the other is an XML file, 'errorsOutput.xml', which communicates the results of the DAD Checker application to other applications. The name of the output text file is user-defined. If no name is specified, the standard output is used.

Installing the DAD Checker

After you download the DADChecker.zip file, extract all files into a directory of your choice. Make sure you have a JRE or JDK V1.3.1 or later installed on your system.

Running the DAD Checker

As mentioned before,the DAD checker is a Java program, that can run on JDK version 1.3.1. and later. To run the DAD checker:

From a command line change to the /bin subdirectory in the directory where you installed the DAD checker

```
CD \[dadchecker-dir]\bin
```

Execute the following commands:

```
setcp -- (This command executes a .bat file that set the CLASSPATH)

java dadchecker.Check_dad_xml [-dad | -xml] [-all] [-dup dupName]
                              [-enc encoding] [-dtd dtdPath]
                              [-doc xmlDocument]
                               [-db driver dbURL userID password]
                               [-out outputFile] fileToCheck
```

The different parameters have the following meaning:

fileToCheck: Specifies the path of the file to be checked. This parameter is required. It must be the final parameter.

-dad: Indicates that the file that is to be checked is a DAD file. This is the default option.

-xml: Indicates that the file that is to be checked is an XML document rather than a DAD file.
For this kind of check, only the output text file is generated: no output XML file is provided.

For large XML documents (several MB), the Java Virtual Machine might run out of memory, producing a java.lang.OutOfMemoryError exception. In such cases, the -Xmx option can be used to allocate more memory to the Java Virtual Machine. Refer to the JDK documentation for details.

-all: The output will show all occurrences of tags that are in error.

-dup dupName: For DAD files, only the duplicate tags whose name attribute values are dupName are displayed. For XML documents, only the duplicate tags or attributes whose names are dupName are displayed.

-enc encoding: Specifies the encoding of the file that is to be checked. encoding can be either a MIME encoding or a Java encoding. This option is used when the encoding declaration in the XML file does not match its actual encoding. It allows you to specify the actual encoding of the document and specifies that the XML parser should ignore the encoding declaration in the file.

-dtd dtdPath: Overrides the DTD declared in the XML document. The dtdPath parameter specifies the path for a DTD in the file system to be used to validate the DAD. The XML parser will then use dtdPath to locate the DTD instead of the DTD declared in the file.

The validation against the DTD is dynamic: it will only be done if a DTD is declared in the file. Consequently, using this option has no effect when a DTD is not declared in the file.

-doc xmlDocument: Specifies an XML document whose structure is to be checked against the DAD's structure. The DAD checker will check that the attributes and leaf tags of xmlDocument are mapped in the DAD. This option cannot be used when checking an XML document with the -xml option.

-db driver dbURL userID password: Specifies the name of the driver to access DB2 with JDBC, the URL of the database to which the DAD checker connects, the user ID and the password needed to connect to the database. Here is an example of use of this option: -db COM.ibm.db2.jdbc.app.DB2Driver jdbc:db2:sales_db "" ""

-out outputFile: Specifies the output text file name. If omitted, the standard output is used.

Note that a second output file, errorsOutput.xml is also created in the same directory as the DAD file, except when the -xml option is used. This file contains in XML form the same information as the output text file except all parser warnings and errors.

-help: Displays command line option information.

-version: Dsplays version information. We used version 1.3

Example 9-11 shows sample output from running the DAD checker utility against a DAD.

Example 9-11 DAD checker sample output

```
C:\dadcheck\bin>java dadchecker.Check_dad_xml
      C:\Progra~1\IBM\SQLLIB\samples\db2xml\dad\getstart_xcollection.dad

Checking DAD document:
C:\Progra~1\IBM\SQLLIB\samples\db2xml\dad\getstart_xcollection.dad

1 duplicate naming conflict(s) were found
A total of 2 tags are in error (total occurrences: 2)

The following tags are duplicates:

<DAD>
  <Xcollection>
    <root_node>
      <element_node name="Order">
        <element_node name="Part">
5              <element_node name="key">  line(s): 26
                 <text_node>
                   <column name="part_key">
----------------
```

```
<DAD>
  <Xcollection>
    <root_node>
      <element_node name="Order">
4            <attribute_node name="key">  line(s): 11
               <column name="order_key">
----------------

----------------------------------------------------

******************************************************
******************************************************
No type attributes are missing for <column> tags.
******************************************************
******************************************************
All <column> tags are properly enclosed.
******************************************************
******************************************************
The 'name' attributes for the <table> and <column> tags are all non empty
strings.
******************************************************
******************************************************
No <element_node> tags have been found with the same names and different
mappings.
******************************
No <attribute_node> tags have been found with the same names and different
mappings.
******************************************************
******************************************************
No missing multi_occurrence="YES" has been found.
******************************************************
******************************************************
FIXPAK 3 or earlier only:
no <attribute_node> tag mapping order problems were found.
```

For more information, such as the various kinds of error checks performed and sample files, please refer to the documentation that comes with the DAD checker download.

9.3 Importing and exporting XML documents using UDF

Using DB2 XML Extender is the preferred way of dealing with XML documents in DB2. However if for some reason if you decide not to use XML Extender, you can still define your table columns using UDTs that support XML data types such as

XMLCLOB, XMLVarchar and XMLFile. The following UDFs can then be used to convert XML data types to and from DB2 base data types.

9.3.1 Importing or storing XML documents

To store the XML data in XML UDTs, you can either use the default casting functions of a UDT directly in INSERT or SELECT statements, or use the UDFs provided by the XML Extender that take XML documents from sources other than the UDT base data type and convert them to the specified UDT.

XMLCLOBFromFile()

This UDF reads an XML document from a (server) file and returns an XMLCLOB type.

- `XMLCLOBFromFile(`*fileName*`)` where *fileName* is the fully qualified server file name of type `VARCHAR(512)`.

XMLFileFromCLOB()

`XMLFileFromCLOB` consumes an XML document as a CLOB locator, writes it to an external (server) file, and returns the file name and path as an XMLFILE type.

- `XMLFileFromCLOB (`*buffer*`,` *fileName*`)` where *buffer* is the CLOB containing the XML document, and *fileName* is the fully qualified (server) file name.

XMLFileFromVarchar()

This UDF 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 type.

- `XMLFileFromCLOB (`*buffer*`,` *fileName*`)` where *buffer* is the memory buffer of type VARCHAR(3k) that contains the XML document and *fileName* is the fully qualified (server) file name.

XMLVarcharFromFile()

This UDF reads an XML document from a server file, and returns the document as an XMLVARCHAR type.

- **`XMLVarcharFromFile`**`(`*fileName*`)` where *fileName* is the fully qualified (server) file name.

9.3.2 Exporting or retrieving XML documents

To retrieve XML data stored in columns that contain XML data types, you can either use the default casting functions, or the overloaded content() UDF. The following are the different forms of the overloaded content() function.

Content (XMLFile xmlObj)

This form of content() takes an object of type XMLFile, retrieves the data from the (server) file specified in the XMLFile object and returns the result as a CLOB locator.

Content (XMLVARCHAR xmlObj , VARCHAR (512) fileName)

This overloaded form of content() retrieves the XML content that is stored as an XMLVARCHAR type, and stores it in an external (server) file specified in *fileName*. If a file with the specified name already exists, the contents of the file is overwritten.

Content (XMLCLOB xmlObj , VARCHAR(512) fileName)

This overloaded form of content() retrieves the XML content that is stored as an XMLCLOB type, and stores it in an external (server) file specified in *fileName*. If a file with the specified name already exists, the contents of the file is overwritten.

DB2 XML Extender also provides functions to extract values from XML documents and convert them to other data types such as INTEGER, REAL, CLOB etc. For more details on these functions please refer to Chapter 5, “Working with XML documents stored in XML columns” on page 113, and *DB2 XML Extender Administration and Programming V8*, SC27-1234.

10

Web services in DB2

In this chapter, we explore the facilities provided by DB2 for Web services. We look at DB2 both as a Web service provider and as a Web service consumer.

As a Web service provider, DB2 can build Web services from SQL statements and from stored procedures.

As a Web service consumer, DB2 can build user-defined functions (UDF) that access Web services and return results as scalar values or tables.

10.1 Introduction to Web services

We start this chapter with a short introduction to Web services. This section is an extract from the IBM Redbook *WebSphere V5 Web Services Handbook*, SG24-6891.

10.1.1 Motivation

Although the rush of the dot-com era seems to have faded, there has been a strong trend in recent years for companies to increasingly integrate existing systems in order to implement IT support for business processes that cover the entire business value chain. Today, interactions already exist using a variety of schemes that range from very rigid point-to-point electronic data interchange (EDI) interactions to open Web auctions. Many companies have already made some of their IT systems available to all of their divisions and departments, or even their customers or partners on the Web. However, techniques for collaboration vary from one case to another and are thus proprietary solutions; systems often collaborate without any overarching vision or architecture.

Thus, there is an increasing demand for technologies that support the connecting or sharing of resources and data in a very flexible and standardized manner. Because technologies and implementations vary across companies and even within divisions or departments, unified business processes could not be smoothly supported by technology. Integration has been developed only between units that are already aware of each other and use the same static applications.

Furthermore, there is a need to further structure large applications into building blocks in order to use well-defined components within different business processes. A shift toward a *service-oriented* approach will not only standardize interaction, but also allow for more flexibility in the process. The complete value chain within a company is divided into small modular functional units, or services. A service-oriented architecture, therefore, has to focus on how services are described and organized to support their dynamic, automated discovery and use.

Companies and their sub-units should be able to easily provide services. Other business units can use these services to implement their business processes. This integration can be ideally performed during the runtime of the system, not just at design time.

10.1.2 Requirements for a service-oriented architecture

For an efficient use of a service-oriented scenario, a number of requirements have to be fulfilled:

- Interoperability between different systems and programming languages

  The most important basis for a simple integration between applications on different platforms is a communication protocol, which is available for most systems and programming languages.

- Clear and unambiguous description language

  To use a service offered by a provider, it is necessary to be able to access the provider system, *and* the syntax of the service interface must be clearly defined in a platform-independent fashion.

- Retrieval of the service

  To allow a convenient integration at design time or even at runtime of the system, we require a mechanism that provides search facilities to retrieve suitable available services. Such services should be classified into computer-accessible, hierarchical categories, or taxonomies, based upon what the services in each category do and how they can be invoked.

10.1.3 Service-oriented architecture overview

This section offers a short introduction to the key concepts of a service-oriented architecture.

Each component in a service-oriented architecture can play one (or more) of three roles: *service provider*, *broker*, and *requestor*, which perform the operations shown in Figure 10-1 on page 272.

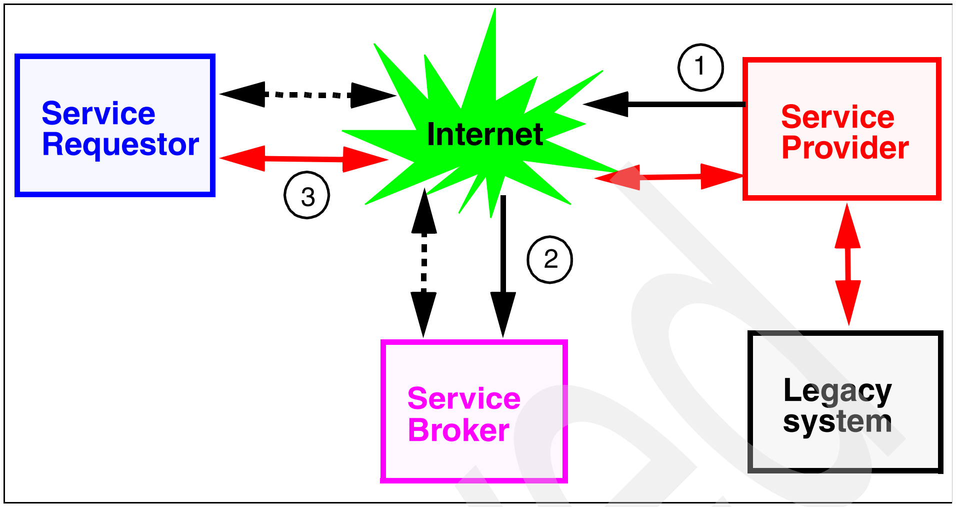


Figure 10-1 Web services roles and operations

1. The *service provider* creates a Web service and possibly publishes its interface and access information to the service registry.

   Each provider must decide which services to expose, how to implement trade-offs between security and easy availability, how to price the services (or, if they are free, how to exploit them for other value). The provider also has to decide what category the service should be listed in for a given broker service and what sort of trading partner agreements are required to use the service.

2. The *service broker* (also known as the service registry) is responsible for making the Web service interface and implementation access information available to any potential service requestor.

   The implementers of a broker have to make a decision about the scope of the broker. Public brokers are available all over the Internet, while private brokers are only accessible to a limited audience, for example users of a company-wide intranet. Furthermore, the width and breadth of the offered information has to be decided. Some brokers will specialize in breadth of listings. Others will offer high levels of trust in the listed services. Some will cover a broad landscape of services and others will focus on a given industry. Brokers will also be available that simply catalog other brokers. Depending on the business model, a broker may attempt to maximize look-up requests, number of listings, or accuracy of the listings.

3. The *service requestor* locates entries in the broker registry using various find operations and then binds to the service provider in order to invoke one of its Web services.

One important issue for users of services is the degree to which services are statically chosen by designers compared to those dynamically chosen at runtime. Even if most initial usage is largely static, any dynamic choice opens up the issues of how to choose the best service provider and how to assess quality of service. Another issue is how the user of services can assess the risk of exposure to failures of service suppliers.

10.1.4 Characteristics of the Web service architecture

The presented service-oriented architecture employs a loose coupling between the participants. Such a loose coupling provides greater flexibility:

- In this architecture, a client is not coupled to a server, but to a service. Thus, the integration of the server to use takes place outside of the scope of the client application programs.
- Old and new functional blocks are encapsulated into components that work as services.
- Functional components and their interfaces are separated. Therefore, new interfaces can be plugged in more easily.
- Within complex applications, the control of business processes can be isolated. A business rule engine can be incorporated to control the workflow of a defined business process. Depending on the state of the workflow, the engine calls the respective services.
- Services can be incorporated dynamically during runtime.
- Bindings are specified using configuration files and can thus easily be adapted to new needs.

10.1.5 Web services approach for a SOA architecture

Web services are a rather new technology that implements the above service-oriented architecture. During the development of this technology, a major focus was put on making functional building blocks accessible over standard Internet protocols that are independent from platforms and programming languages.

Web services are self-contained, modular applications that can be described, published, located, and invoked over a network. Web services perform encapsulated business functions, ranging from simple request-reply to full business process interactions.

These services can be new applications or just wrapped around existing legacy systems to make them network-enabled. Services can rely on other services to achieve their goals.

The following are the core technologies used for Web services. These technologies are covered in detail in the subsequent chapters.

- **XML** (eXtensible Markup Language) is the markup language that underlies most of the specifications used for Web services. XML is a generic language that can be used to describe any kind of content in a structured way, separated from its presentation to a specific device.
- **SOAP** (formerly referred to as *Simple Object Access Protocol*, or *Service-Oriented Architecture Protocol*—in fact, similarly to JDBC, it is no longer an acronym) is a network, transport, and programming language-neutral protocol that allows a client to call a remote service. The message format is XML.
- **WSDL** (Web services description language) is an XML-based interface and implementation description language. The service provider uses a WSDL document in order to specify the operations a Web service provides, as well as the parameters and data types of these operations. A WSDL document also contains the service access information.
- **UDDI** (universal description, discovery, and integration) is both a client-side API and a SOAP-based server implementation that can be used to store and retrieve information on service providers and Web services.

Figure 10-2 on page 275 shows the relationship between the core elements of the SOA.

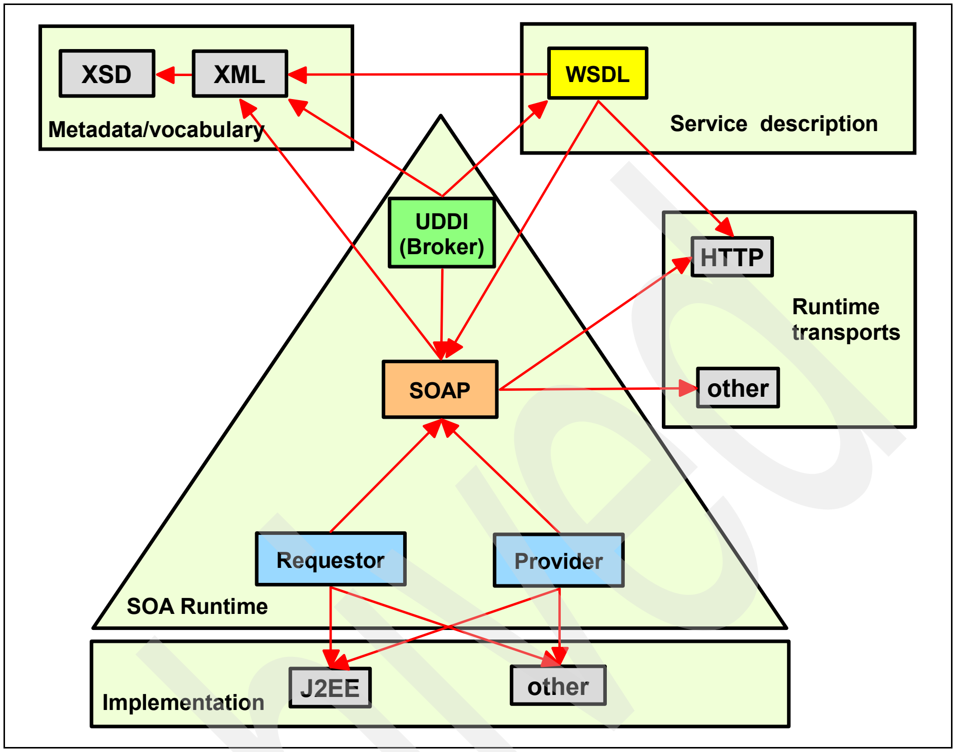


Figure 10-2 Main building blocks in an SOA approach based on Web services

- All elements use XML including XML namespaces and XML Schemas.
- Service requestor and provider communicate with each other.
- WSDL is one alternative to make service interfaces and implementations available in the UDDI registry.
- WSDL is the base for SOAP server deployment and SOAP client generation.

10.1.6 Properties of the service-oriented architecture

The service-oriented architecture offers the following properties:

- Web services are self-contained.

  On the client side, no additional software is required. A programming language with XML and HTTP client support is enough to get you started. On the server side, merely a Web server and a SOAP server are required. It is possible to Web services enable an existing application without writing a single line of code.

- Web services are self-describing.

  Neither the client nor the server knows or cares about anything besides the format and content of request and response messages (loosely coupled application integration). The definition of the message format travels with the message; no external metadata repositories or code generation tool are required.

- Web services can be published, located, and invoked across the Web.

  This technology uses established lightweight Internet standards such as HTTP. It leverages the existing infrastructure. Some additional standards that are required to do so include SOAP, WSDL, and UDDI.

- Web services are language-independent and interoperable.

  Client and server can be implemented in different environments. Existing code does not have to be changed in order to be Web service enabled.

- Web services are inherently open and standard-based.

  XML and HTTP are the major technical foundation for Web services. A large part of the Web service technology has been built using open-source projects. Therefore, vendor independence and interoperability are realistic goals this time.

- Web services are dynamic.

  Dynamic e-business can become reality using Web services because, with UDDI and WSDL, the Web service description and discovery can be automated.

- Web services can be composed.

  Simple Web services can be aggregated to more complex ones, either using workflow techniques or by calling lower-layer Web services from a Web service implementation. Web services can be chained together to perform higher-level business functions. This shortens development time and enables best-of-breed implementations.

- Web services build on proven mature technology.

  There are a lot of commonalities, as well as a few fundamental differences to other distributed computing frameworks. For example, the transport protocol is text based and not binary.

- Web services are loosely coupled.

  Traditionally, application design has depended on tight interconnections at both ends. Web services require a simpler level of coordination that allows a more flexible re-configuration for an integration of the services in question.

- Web services provide programmatic access.

  The approach provides no graphical user interface; it operates at the code level. Service consumers have to know the interfaces to Web services but do not have to know the implementation details of services.

- Web services provide the ability to wrap existing applications.

  Already existing stand-alone applications can easily be integrated into the service-oriented architecture by implementing a Web service as an interface.

10.1.7 More information

General introductions to Web services can be found at:

`http://www.ibm.com/developerworks/webservices/`
`http://xml.watson.ibm.com/`

The following Web site provides a collection of IBM resources on the topic at hand. For example, you can find an introduction to the SOA in a white paper titled *Web Services Conceptual Architecture* (WSCA 1.0):

`http://www.ibm.com/software/solutions/webservices/resources.html`

More information is provided in the article *Energize e-business with Web services from the IBM WebSphere software platform* at:

`http://www.ibm.com/developerworks/library/ibm-lunar.html`

10.2 DB2 as Web service provider

Web services are XML-based application functions that can be invoked over the Internet. The Web Services Object Runtime Framework (WORF) that ships with DB2 for Linux, UNIX and Windows V8.1 provides an environment to easily create simple Web services that access DB2. The Web Services Object Runtime Framework (WORF) will also be made available on DB2 for z/OS with the PTF for APAR PQ91315 (still open at the time of writing this redbook). WORF uses Apache SOAP 2.2 or later and the document access definition extension (DADX). A DADX document specifies how to create a Web service using a set of operations that are defined by SQL statements (including stored procedure calls) and, optionally, XML Extender document access definition (DAD) files. The Web services that are created from a DADX file are called DADX Web services or DB2 Web services.

WORF can use all DB2 XML publishing (composition) functions, namely SQL/XML and XML Extender discussed in Chapter 8, “Publishing data as XML” on page 209, or use the WORF XML composition when using non-SQL/XML SELECT statements.

Figure 10-3 on page 278 represents the architecture of WORF.

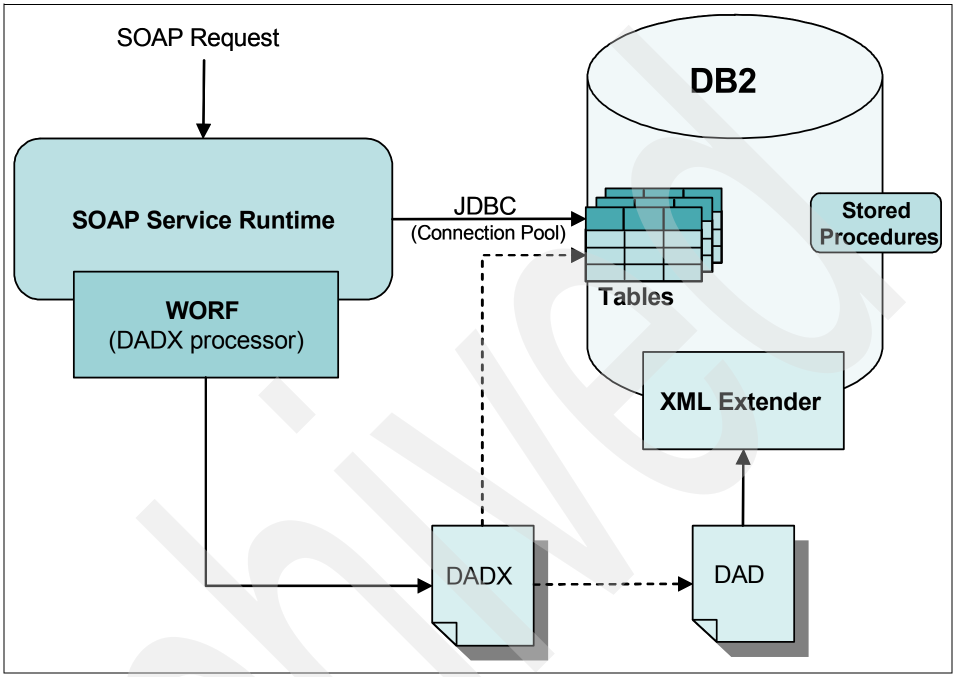


Figure 10-3 WORF architecture

10.2.1 Web Services Object Runtime Framework

WORF provides the following features:

- Resource-based deployment and invocation
- Automatic service redeployment at development time when defining resources change
- HTTP GET and POST bindings in addition to SOAP
- Automatic WSDL and XSD generation, including support for UDDI best practices
- Automatic documentation and test page generation

In the following sections, we explain how to:

- Install WORF on IBM WebSphere Application Server

- Create a simple Web service that exposes the result of SQL statements, and generating a Web application that hosts a Web service
- Deploy the Web application in WebSphere
- Test the Web service

10.2.2 Installing WORF on IBM WebSphere Application Server

In this section, we assume the following installation setup:

- WebSphere Application Server 5.0 is installed in `C:\WebSphere\AppServer`. We call this location `<washome>`. SOAP 2.2 ships by default with WebSphere Application Server V4.01 onwards. Ensure that you have soap.jar in <washome>\lib.
- DB2 8.1 Fixpack 2 is installed in C:\SQLLIB. We call this location `<db2home>`. WORF requires JDBC 2.0, which is the default in DB2 8.1. For DB2 7.2, select JDBC 2.0 by running the <db2home>\java12\usejdbc2.bat file.
- The DB2 `SAMPLE` database is created. If not, you can create it using the First Steps wizard (**Start -> Programs -> IBM DB2 -> Set-up Tools -> First Steps**).

To install WORF on WebSphere Application Server, complete the following steps:

1. Locate dxxworf.zip in your <db2home>\samples\java\Websphere directory. If you cannot find it there, you can download it from the following Web site. However, it should be noted that this download ONLY applies to DB2 V7.2 (FixPak 7). The Web site also contains valuable information about DB2 Web services implementations. WORF ships with DB2 UDB V8.1 and as part of WebSphere Studio.

   `http://www.ibm.com/developerworks/db2/zones/webservices/worf/`

2. Unzip the dxxworf.zip to a directory, such as <washome>\worf. The directory has the following contents:
   - readme.html—explanation of files
   - lib\worf.jar—WORF runtime for WebSphere Application Server
   - lib\worf-servlets.jar—servlets used for DB2 Web services that have to be included in your Web applications using WORF
   - lib\services.war—sample Web application with DB2 Web services
   - schemas—directory with XML Schemas for the DADX and NST XML files
   - tools—directory with DAD and DADX syntax checkers
3. Copy worf.jar to <washome>\lib

4. Start the WebSphere Administration Server.

10.2.3 Creating DB2 Web services

In this section, we show you how to create a simple DB2 Web service based on the SAMPLE database. For simplicity we only consider exposing SQL statements as Web services.

Creating the DADX document

A DADX file defines a Web service by specifying a set of operations. *Operations* are similar to methods that you can invoke. The definition of an operation consists of a list of parameters and an action to be performed. The action to perform is defined using SQL statements or DAD file references. DADX uses XML syntax to define the Web service.

The operations in a DADX Web service can be defined by the following operation types:

- SQL operations
  - <query>—queries the database
  - <update>—performs an update, insert or delete operation against the database
  - <call>—calls stored procedures
- XML collection operations (requires DB2 XML Extender)
  - <retrieveXML>—generates XML documents
  - <storeXML>—stores XML documents

Let us look at a sample DADX file. Example 10-1 is a sample DADX file that exposes two operations, listSales and listDepartments, as Web service operations. For simplicity we use only the SQL operations in this DADX example.

Example 10-1 List.dadx

```
<?xml version="1.0" encoding="UTF-8"?>
<DADX xmlns="http://schemas.ibm.com/db2/dxx/dadx">
   <operation name="listDepartments">
      <documentation>  Lists all the departments in the DEPARTMENT table of
                       the SAMPLE database </documentation>
      <query>
         <SQL_query>SELECT * FROM department</SQL_query>
      </query>
   </operation>

   <operation name="listSales">
      <query>
```

```
         <SQL_query>SELECT * FROM sales WHERE SALES_PERSON = :name</SQL_query>
         <parameter name="name" type="xsd:string"/>
      </query>
   </operation>
</DADX>
```

Note: The second operation, `listSales`, has the salesperson's name as a parameter.

Creating the Web application

To invoke the Web service defined in the DADX document, we have to deploy it in the Application Server as part of a Web application. In this section we create a Web application that can host our DB2 Web service. Here we show you how to create a Web application manually, without using any of the tools. This is mainly to show which are the components that are required and how to obtain the information and configure the setup. In real life, you would most likely use a tool like WebSphere Studio to create you Web service. This is explained in 12.3, "Web services tools" on page 375.

Starting from a Web application skeleton

A skeleton of a simple Web application consists of a WEB-INF folder and a welcome page in HTML. The WEB-INF folder contains a file named web.xml, also called the deployment descriptor that contains the details about deployment of various resources of the Web application. WEB-INF contains two more folders named classes and lib. Both these folders are in the classpath of the class loader. Generally, we keep all the JAR files in the lib folder and other invokable resources under classes.

Before building our sample Web application let us create a skeleton Web application by creating the directory structure as shown in Example 10-2. Keep the index.html and web.xml blank for the time being. WORFTest is the base directory for our Web application for the rest of this chapter.

Example 10-2 Skeleton Web application

```
WORFTest
|
|-- index.html
|
|-- WEB-INF
    |
    |-- web.xml
    |
    |-- classes
    |
    |-- lib
```

Defining the group for Web services

We create a group for each set of related Web services. When we say related, in this context, we mean Web services that access the same database. As part of the group definition we create a connection configuration for the database used by the Web services. Each such group forms a directory in the file system. These groups are created under the /WEB-INF/classes/groups directory. The DADX files are stored in the directories defined for these groups.

Let us create a group for our Web service and configure it for the SAMPLE database. Let us call our group the `TestGroup`.

1. Create the groups directory: WORFTest/WEB-INF/classes
2. Create the TestGroup directory: WORFTest/WEB-INF/classes/groups
3. Create the group.properties file in TestGroup and add the lines listed in Example 10-3 on page 283
4. Store the List.dadx file under WORFTest/WEB-INF/classes/groups/TestGroup

Example 10-3 Sample group.properties file

```
initialContextFactory=
datasourceJNDI=
dbDriver=COM.ibm.db2.jdbc.app.DB2Driver
dbURL=jdbc:db2:sample
userID=database-user-id
password=password-for-the-above-user-id
namespaceTable=
autoReload=true
reloadIntervalSeconds=5
groupNamespaceUri=
enableXmlClob=true
```

You can leave `userID` and `password` empty if your logon ID and the WebSphere ID are authorized for DB2. It is also possible to use a data source instead of a JDBC driver, but this requires more configuration. Groups with data sources are covered in “Creating a DADX group configuration” on page 375.

Define the servlet for the group

Now we have to inform the application server about the existence of this group, so that at runtime the correct group is looked up for DADX documents. We do this by creating a servlet for each group. This servlet is an instance of the `DxxInvoker` servlet that is part of the worf-servlets.jar. The name of this servlet instance should be the same as the name of the Web service group, which is `TestGroup` in our case. When a request is issued to serve a DADX document, defined under a particular group, the servlet instance associated with this group gets invoked.The `DxxInvoker` servlet determines where to find DADX files by looking for a directory that matches its servlet name.

1. Copy the worf-servlets.jar into WORFTest/WEB-INF/lib. This jar file is part of dxxworf.zip we unzipped in “Creating the Web application” on page 281.
2. Our next step is to create an instance of the `DxxInvoker` servlet named `TestGroup`. We do this by adding the lines listed in Example 10-4 to our web.xml.

Example 10-4 Servlet configuration information that goes into web.xml

```
<web-app>
   <servlet>
      <servlet-name>TestGroup</servlet-name>
      <servlet-class>
         com.ibm.etools.webservice.rt.dxx.servlet.DxxInvoker
      </servlet-class>
      <init-param >
         <param-name>faultListener</param-name>
         <param-value>org.apache.soap.server.DOMFaultListener</param-value>
      </init-param>
      <load-on-startup>-1</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>TestGroup</servlet-name>
      <url-pattern>/testing/*</url-pattern>
   </servlet-mapping>
</web-app>
```

Note: For every group you want to create, you have to follow the steps mentioned above.

If you analyze the definition of our servlet 'TestGroup' you see that the *url-pattern* for this servlet is `/testing/*`. Hence, when WebSphere Application Server gets the request for /testing/List.dadx, it invokes the TestGroup servlet. This servlet instance looks for a directory called TestGroup (its own name) under /WEB-INF/classes/groups directory. In this directory it searches for List.dadx and invokes the Web service operation.

Packaging the Web application

Now is the time to create our welcome file associate it with the Web application. enter the text shown in Example 10-5 into WORFTest/index.html file and save it.

Example 10-5 Contents of index.html

```
<html>
<head> <title>Simple DB2 Web Services </title> </head>
<body>
   <h1> Simple DB2 Web Services </h1>
   <hr>
   <br>Test the <a href="testing/List.dadx/listDepartments">
         listDepartments</a> Web Service
   <br>Test the <a href="testing/List.dadx/listSales?name=LEE">
         listSales      </a> Web Service for LEE <br>
   <br>Generate <a href="testing/List.dadx/WSDL"> WSDL</a>
   <br>Generate <a href="testing/List.dadx/XSD">  XSD</a>
</body>
</html>
```

> **Note:** We are using a fixed name as the name parameter for this simple test. We could use Java script or a servlet to pass the name from a form.
>
> Here we call a Web service from a static HTML page. We can do this because WORF supports GET bindings. (Plain Apache SOAP or Apache Axis do not have GET bindings). As WORF also supports POST bindings, you can have the user to enter the name instead of have a fixed name if the user creates an HTML form.

Now, to associate the index.html file as the welcome file for the Web application, we insert the following lines into our web.xml between `<webapp>` and `</webapp>`.

```
   <welcome-file-list>
      <welcome-file>index.html </welcome-file>
   </welcome-file-list>
```

Now the completed web.xml file should look as shown in Example 10-6.

Example 10-6   Contents of web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.2//EN" "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">
   <web-app>
      <servlet >
         <servlet-name>TestGroup</servlet-name>
         <servlet-class>
            com.ibm.etools.webservice.rt.dxx.servlet.DxxInvoker
         </servlet-class>
         <init-param >
            <param-name>faultListener</param-name>
            <param-value>org.apache.soap.server.DOMFaultListener</param-value>
         </init-param>
         <load-on-startup>-1</load-on-startup>
      </servlet>
      <servlet-mapping>
         <servlet-name>TestGroup</servlet-name>
         <url-pattern>/testing/*</url-pattern>
      </servlet-mapping>
      <welcome-file-list>
         <welcome-file>index.html </welcome-file>
      </welcome-file-list>
   </web-app>
```

With this step, we come to the end of our Web application construction. By now you should have the directory structure as shown in Example 10-7 ready with each file containing the contents discussed in the previous sections. We can now proceed to package the Web application as a Web archive or WAR file by entering this command (from the WORFTest directory):

```
jar -cvf SimpleWorfTest.war
```

Note. The . at the end of the command is required.

Example 10-7 Final directory structure of the sample Web application

```
WORFTest
|
|-- index.html
|
|-- WEB-INF
    |
    |-- web.xml
    |
    |-- lib
    |    |
    |    |-- worf-servlets.jar
    |
    |-- classes
         |
         |-- groups
              |
              |--TestGroup
                  |
                  |-- group.properties
                  |-- List.dadx
```

In case you want to set up the security for the Web service provider application, you need to do so at this point. In our example we did not want to complicate things by adding security. Please refer to the following article on the Web for more details:

http://www-106.ibm.com/developerworks/db2/library/techarticle/dm-0404wollscheid/index.html

Now we have created a Web application that can be deployed to WebSphere Application Server.

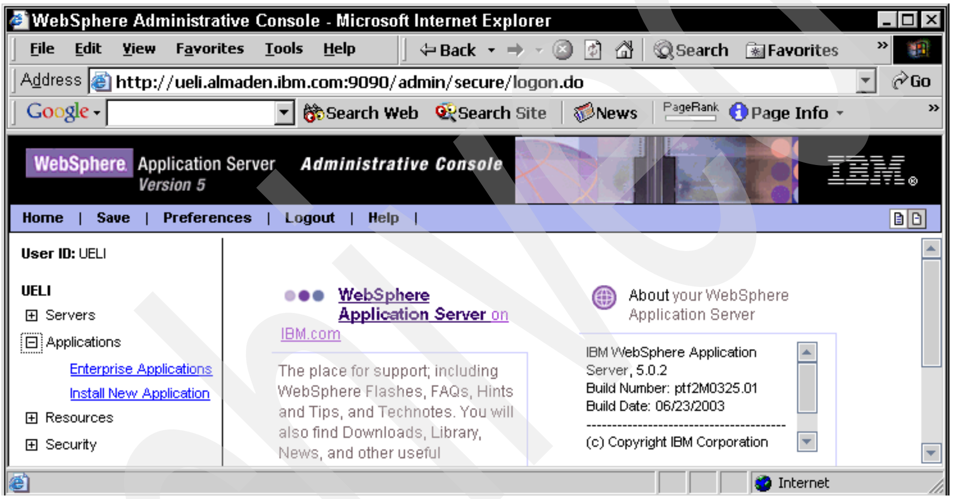
10.2.4 Deploying the Web application

In this section, we show you how to deploy the Web application on WebSphere 5.0 or 5.0.2.

1. Make sure the WebSphere server is started.
2. Start the WebSphere Administrative Console from the WebSphere program group or by typing the following URL in the Web browser's address bar:

   ```
   http://localhost:9090/admin
   ```

3. Enter your user ID and the Administrative Console opens (Figure 10-4).



WebSphere Administrative Console - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://ueli.almaden.ibm.com:9090/admin/secure/logon.do
Go
Google
Search Web Search Site News PageRank Page Info
WebSphere Application Server Version 5
Administrative Console
IBM
Home | Save | Preferences | Logout | Help |
User ID: UELI
UELI
Servers
Applications
Enterprise Applications
Install New Application
Resources
Security
WebSphere Application Server on IBM.com
About your WebSphere Application Server
Internet

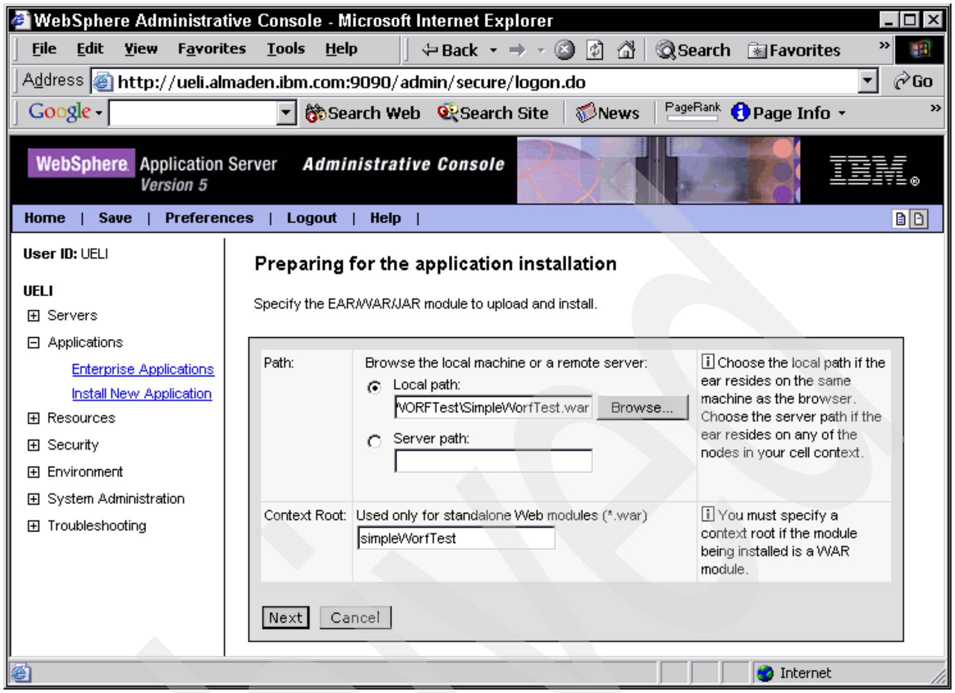
The place for support, including WebSphere Flashes, FAQs, Hints and Tips, and Technotes. You will also find Downloads, Library, News, and other useful

IBM WebSphere Application Server, 5.0.2
Build Number: ptf2M0325.01
Build Date: 06/23/2003

(c) Copyright IBM Corporation

Figure 10-4 WebSphere Administrative Console startup page

4. Select **Install New Applications** under *Applications*. Provide the path of the Web application you created and also provide a context name in the appropriate input field. (Figure 10-5 on page 288). Note that the context root is part of the URL that you specify to invoke the Servlet.


WebSphere Administrative Console - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://ueli.almaden.ibm.com:9090/admin/secure/logon.do Go
Google Search Web Search Site News PageRank Page Info
WebSphere Application Server Version 5 Administrative Console
IBM
Home | Save | Preferences | Logout | Help |
User ID: UELI
UELI
Servers
Applications
Enterprise Applications
Install New Application
Resources
Security
Environment
System Administration
Troubleshooting
Next
Cancel
Browse...
Internet

Preparing for the application installation

Specify the EAR/WAR/JAR module to upload and install.

Path: Browse the local machine or a remote server:

● Local path: \WORFTest\SimpleWorfTest.war

○ Server path:

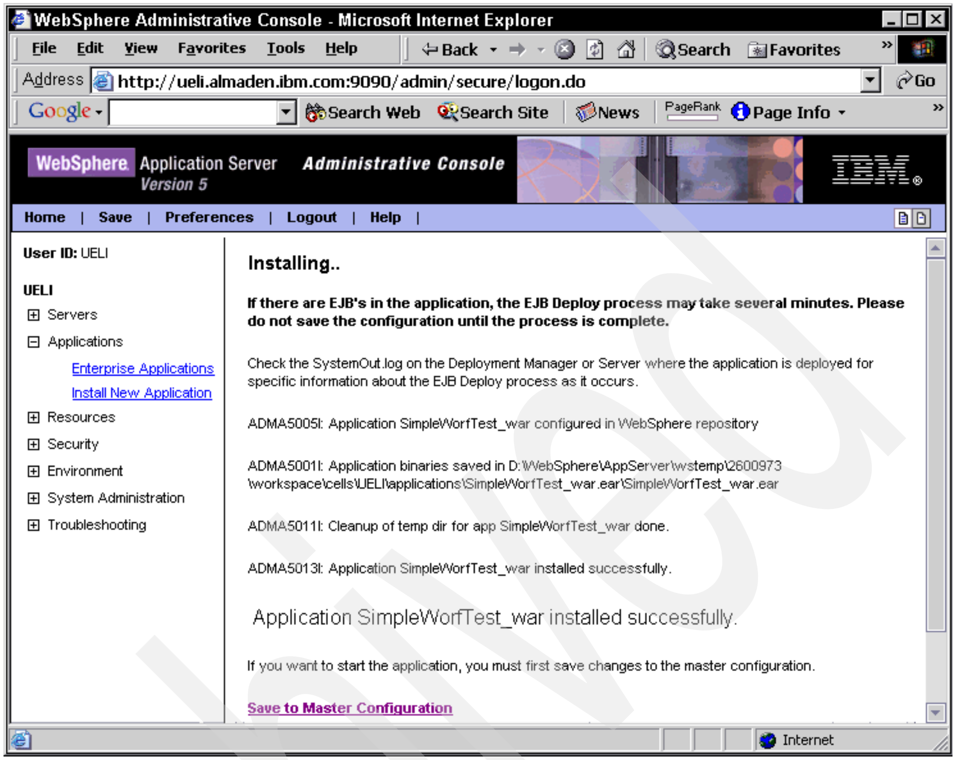
Choose the local path if the ear resides on the same machine as the browser. Choose the server path if the ear resides on any of the nodes in your cell context.

Context Root: Used only for standalone Web modules (*.war) simpleWorfTest

You must specify a context root if the module being installed is a WAR module.

Figure 10-5 Install an enterprise application

5. Click **Next** and keep accepting the default values for the next few panels:
   - The default virtual host is `default_host`.
   - The application name is `SimpleWorfTest_war`.
   - The `SimpleWorfTest.war` maps to the `default_host` virtual host.
   - The module is installed on the selected server.
   - The summary panel recapitulates the options.
   - Click **Finish** and the enterprise application is installed. The installation messages are shown in Figure 10-6 on page 289.



WebSphere Administrative Console - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://ueli.almaden.ibm.com:9090/admin/secure/logon.do
Go
Google Search Web Search Site News PageRank Page Info
WebSphere Application Server Version 5 Administrative Console
IBM
Home | Save | Preferences | Logout | Help |
User ID: UELI
UELI
Servers
Applications
Enterprise Applications
Install New Application
Resources
Security
Environment
System Administration
Troubleshooting
Internet

Installing..

If there are EJB's in the application, the EJB Deploy process may take several minutes. Please do not save the configuration until the process is complete.

Check the SystemOut.log on the Deployment Manager or Server where the application is deployed for specific information about the EJB Deploy process as it occurs.

```
ADMA5005I: Application SimpleWorfTest_war configured in WebSphere repository

ADMA5001I: Application binaries saved in D:\WebSphere\AppServer\wstemp\2600973
\workspace\cells\UELI\applications\SimpleWorfTest_war.ear\SimpleWorfTest_war.ear

ADMA5011I: Cleanup of temp dir for app SimpleWorfTest_war done.

ADMA5013I: Application SimpleWorfTest_war installed successfully.
```

Application SimpleWorfTest_war installed successfully.

If you want to start the application, you must first save changes to the master configuration.

Save to Master Configuration

Figure 10-6 Installation messages

6. Select **Save to Master Configuration**. Click **Save** on the confirmation panel.

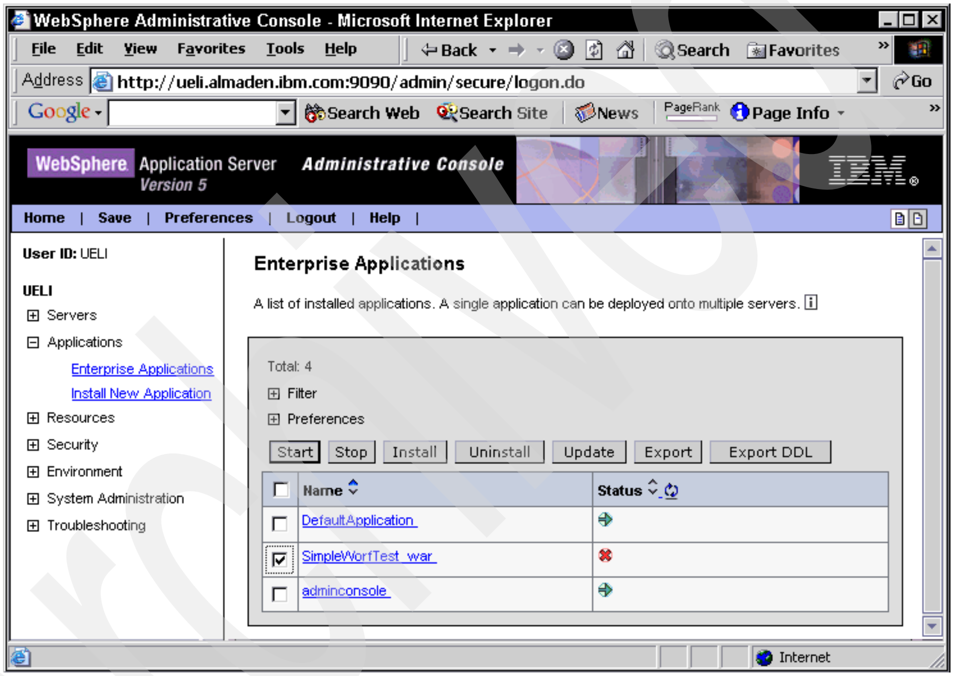
   This saves the details about the Web application into the WebSphere repository. Once the save action is completed, the window shown in Figure 10-4 on page 287 is displayed again. Now you can start the Web application to test our Web services.

> **Note:** If you have never used DB2 with you WebSphere Application Server before, you must set up the correct CLASSPATH for the server before you can start the application.
>
> To verify that DB2 JDBC drivers are included as part of the CLASSPATH, you use the WebSphere Administrative Console. Go to **Servers -> Application Servers -> <name of your server> -> Process Definition -> Java Virtual Machine**. The correct value, which in our case is C:\SQLLIB\java\db2java.zip, should then be entered into the CLASSPATH field.

10.2.5 Starting the enterprise application

In this section, we assume that you completed all the instructions detailed in the previous sections and your Web application is successfully deployed and your WebSphere Administrative client is displaying the main page as shown in Figure 10-4 on page 287.

1. Select **Enterprise Applications** under Applications.
2. In the Enterprise Applications panel (Figure 10-7) select the **SimpleWorfTest_war** application (that you just installed) and click **Start**.


WebSphere Administrative Console - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://ueli.almaden.ibm.com:9090/admin/secure/logon.do
Go
Google Search Web Search Site News PageRank Page Info
WebSphere Application Server Version 5 Administrative Console
IBM
Home | Save | Preferences | Logout | Help |
User ID: UELI
UELI
Servers
Applications
Enterprise Applications
Install New Application
Resources
Security
Environment
System Administration
Troubleshooting
Total: 4
Filter
Preferences
Start Stop Install Uninstall Update Export Export DDL
Internet

Enterprise Applications

A list of installed applications. A single application can be deployed onto multiple servers.

☐	Name	Status
☐	DefaultApplication	➜
☑	SimpleWorfTest_war	✖
☐	adminconsole	➜

Figure 10-7 Enterprise Application administration page

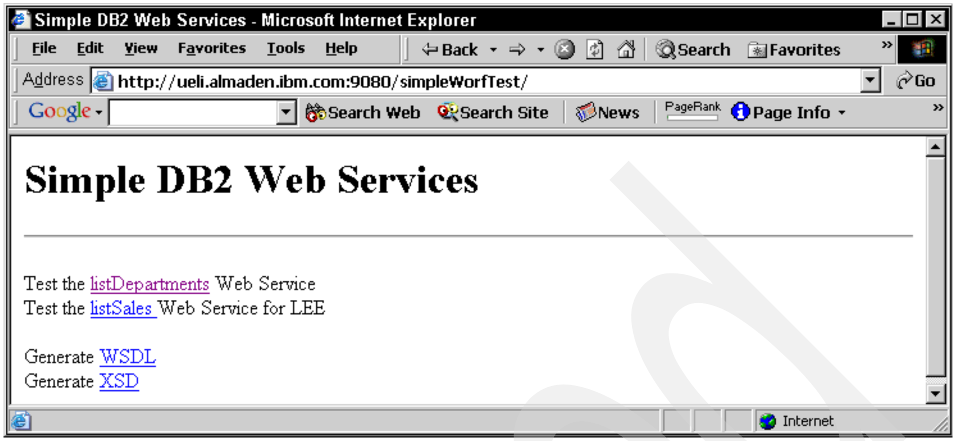
If you are able to start the Web application a green arrow replaces the red cross.

10.2.6 Testing the DB2 Web services

Now it is time for us to see our DB2 Web services in action. In this section we test the DB2 Web services that we developed and deployed.

1. Open a Web browser and type the following URL in the address bar (Figure 10-8 on page 291):

```
http://localhost:9080/simpleWORFTest/
```



Simple DB2 Web Services - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://ueli.almaden.ibm.com:9080/simpleWorfTest/
Go
Google Search Web Search Site News PageRank Page Info
Internet

Simple DB2 Web Services

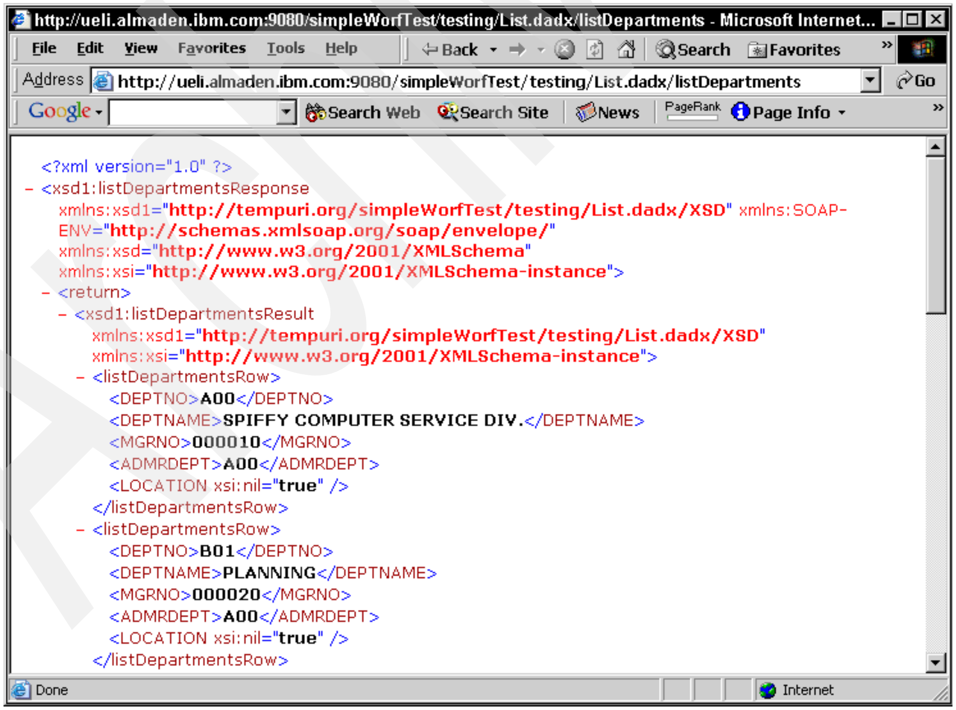
Test the listDepartments Web Service
Test the listSales Web Service for LEE

Generate WSDL
Generate XSD

Figure 10-8 Web application welcome page

2. Select **listDepartments** to test the Web service that lists the departments of the SAMPLE database (Figure 10-9). The SQL statement for this operation is:

```
select * from DEPARTMENT
```



http://ueli.almaden.ibm.com:9080/simpleWorfTest/testing/List.dadx/listDepartments - Microsoft Internet...
File Edit View Favorites Tools Help
Back Search Favorites
Address http://ueli.almaden.ibm.com:9080/simpleWorfTest/testing/List.dadx/listDepartments
Go
Google Search Web Search Site News PageRank Page Info
Done
Internet

```
  <?xml version="1.0" ?>
- <xsd1:listDepartmentsResponse
    xmlns:xsd1="http://tempuri.org/simpleWorfTest/testing/List.dadx/XSD" xmlns:SOAP-
    ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  - <return>
    - <xsd1:listDepartmentsResult
        xmlns:xsd1="http://tempuri.org/simpleWorfTest/testing/List.dadx/XSD"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      - <listDepartmentsRow>
          <DEPTNO>A00</DEPTNO>
          <DEPTNAME>SPIFFY COMPUTER SERVICE DIV.</DEPTNAME>
          <MGRNO>000010</MGRNO>
          <ADMRDEPT>A00</ADMRDEPT>
          <LOCATION xsi:nil="true" />
        </listDepartmentsRow>
      - <listDepartmentsRow>
          <DEPTNO>B01</DEPTNO>
          <DEPTNAME>PLANNING</DEPTNAME>
          <MGRNO>000020</MGRNO>
          <ADMRDEPT>A00</ADMRDEPT>
          <LOCATION xsi:nil="true" />
        </listDepartmentsRow>
```

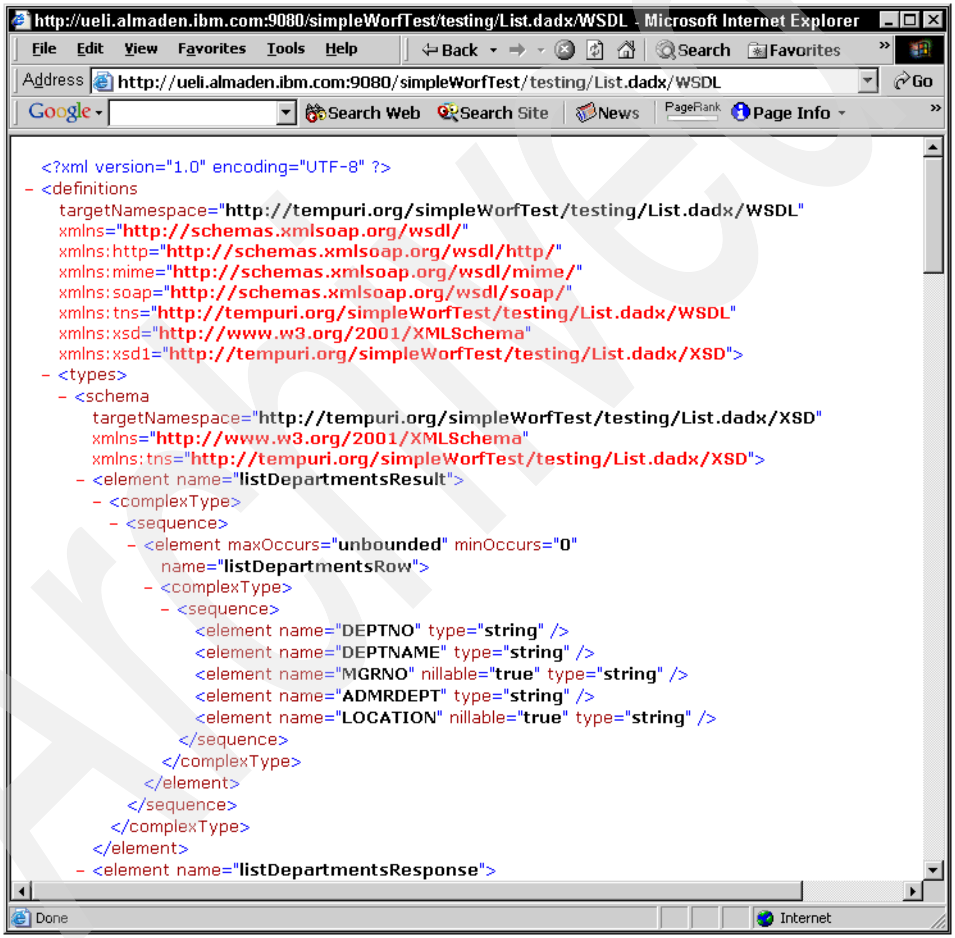
Figure 10-9 DB2 Web service: department list

3. Similarly you can also select **listSales** to invoke the `listSales` operation of the sample Web service. You can also achieve the same result by directly typing the URL:

   `http://localhost:9080/simpleWorfTest/testing/List.dadx/`**`listSales`**

4. As mentioned in the introduction WORF automatically generates the WSDL and XSD for the Web service hosted. You can view the WSDL for the Web service defined by `List.dadx` by selecting the **WSDL** link on the welcome page (Figure 10-10). You can also directly invoke this by entering the URL:

   `http://localhost:9080/simpleWorfTest/testing/List.dadx/`**`WSDL`**


http://ueli.almaden.ibm.com:9080/simpleWorfTest/testing/List.dadx/WSDL - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Address http://ueli.almaden.ibm.com:9080/simpleWorfTest/testing/List.dadx/WSDL
Go
Google
Search Web
Search Site
News
PageRank
Page Info
Done
Internet

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <definitions
    targetNamespace="http://tempuri.org/simpleWorfTest/testing/List.dadx/WSDL"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://tempuri.org/simpleWorfTest/testing/List.dadx/WSDL"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://tempuri.org/simpleWorfTest/testing/List.dadx/XSD">
  - <types>
    - <schema
        targetNamespace="http://tempuri.org/simpleWorfTest/testing/List.dadx/XSD"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://tempuri.org/simpleWorfTest/testing/List.dadx/XSD">
      - <element name="listDepartmentsResult">
        - <complexType>
          - <sequence>
            - <element maxOccurs="unbounded" minOccurs="0"
                name="listDepartmentsRow">
              - <complexType>
                - <sequence>
                    <element name="DEPTNO" type="string" />
                    <element name="DEPTNAME" type="string" />
                    <element name="MGRNO" nillable="true" type="string" />
                    <element name="ADMRDEPT" type="string" />
                    <element name="LOCATION" nillable="true" type="string" />
                  </sequence>
                </complexType>
              </element>
            </sequence>
          </complexType>
        </element>
      - <element name="listDepartmentsResponse">
```

Figure 10-10 DB2 Web service: WSDL

5. In the same way, you can view the XSD file:

   `http://localhost:9080/simpleWorfTest/testing/List.dadx/XSD`

10.2.7 DADX and auto-deploy feature of WORF

If you want to add another operation to the same Web service, all you have to do is add another operation element into the `List.dadx` file and save the file. WORF will automatically pick up the new operation when you invoke it, if the group properties (Example 10-3 on page 283) specify:

```
autoReload=true
```

Also if you want to create another related Web service with one or more operations in it, you can create a new DADX file similar to Example 10-1 on page 280 and place it in the same directory as the deployed List.dadx:

```
<Washome>\installedApps\<your-node-name>\SimpleWorfTest_war.ear
         \SimpleWorfTest.war\WEB-INF\classes\groups\TestGroup
```

You do not have to redeploy your Web application. WORF will pick up the new DADX automatically when the DADX is invoked.

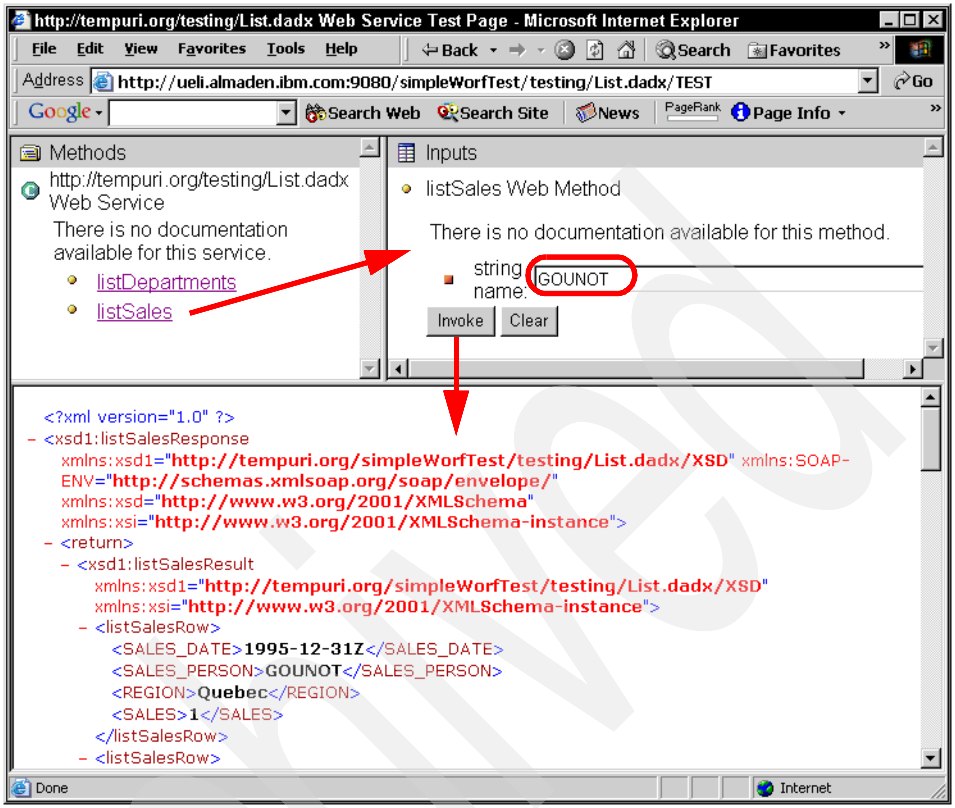
In the preceding sections, we showed you how to expose an SQL query statement as a Web service. In fact, WORF is capable of exposing any SQL statement, as well as features provided by DB2 XML Extender. If you are interested in learning more about other features of DADX, you can install the sample Web application services.war, that is part of DB2 Web services provider package as mentioned in 10.2.2, “Installing WORF on IBM WebSphere Application Server” on page 279. This sample Web application has many more examples of DADX files that you can play around with.

Tip: In case the *services* application does not work, you may try renaming soap-ibm.xml as soap.xml, and dds-example.xml as dds.xml in the services.war folder after installing the services.war file.

10.2.8 WORF test facility

To run the WORF test facility (Figure 10-11 on page 294), start a browser and enter the URL:

```
http://localhost:9080/simpleWorfTest/testing/List.dadx/TEST
```


http://tempuri.org/testing/List.dadx Web Service Test Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Address
Go
Google
Search Web
Search Site
News
PageRank
Page Info
Methods
Inputs
Invoke
Clear
Done
Internet

http://ueli.almaden.ibm.com:9080/simpleWorfTest/testing/List.dadx/TEST

http://tempuri.org/testing/List.dadx Web Service

There is no documentation available for this service.

- listDepartments
- listSales

listSales Web Method

There is no documentation available for this method.

string name: GOUNOT

```
<?xml version="1.0" ?>
- <xsd1:listSalesResponse
    xmlns:xsd1="http://tempuri.org/simpleWorfTest/testing/List.dadx/XSD" xmlns:SOAP-
    ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  - <return>
    - <xsd1:listSalesResult
        xmlns:xsd1="http://tempuri.org/simpleWorfTest/testing/List.dadx/XSD"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      - <listSalesRow>
          <SALES_DATE>1995-12-31Z</SALES_DATE>
          <SALES_PERSON>GOUNOT</SALES_PERSON>
          <REGION>Quebec</REGION>
          <SALES>1</SALES>
        </listSalesRow>
      - <listSalesRow>
```

Figure 10-11 WORF test facility

1. Select one of the operations (**listSales** in our example).
2. Enter the parameter (salesperson name, `LEE` or `GOUNOT`) and click **Invoke**.

The resulting XML is shown in the bottom pane.

This concludes our examples with DB2 as a Web service provider.

10.3 DB2 as Web service consumer

Web services are increasingly used to integrate information processing within and between enterprises. When building service-based applications, Web services often have to be integrated with relational data. To accomplish this, applications must access both Web services and database management systems.

IBM DB2 Web service consumer user-defined functions (UDFs) are now available to help with this task. These new Web service consumer UDFs enable databases to directly invoke Web services using SQL. This eliminates the need to transfer data between Web services and the database. The result is increased productivity and better performance. The Web services consumer converts existing WSDL interfaces into DB2 table or scalar functions.

In the sections that follow, we show you how to enable DB2 as a Web service consumer and write a sample UDF that acts as a consumer to a standard Web service that fetches delayed stock quote given the stock symbol.

10.3.1 Prerequisites

You should have DB2 XML Extender installed before you begin enabling the DB2 Web service consumer UDFs. DB2 XML Extender is packaged with DB2 UDB for Linux, UNIX and Windows V8.1. See “DB2 V8.1 with FixPak 2” on page 546 for installation instructions.

The Web services consumer code is packaged with DB2 UDB V8.1 FixPak 2 and later. The Web services consumer functionality will also be made available on DB2 for z/OS with the PTF for APAR PQ91316 (still open at the time of writing this redbook).

The following discussion assumes that you have DB2 UDB version 8.1 FixPak 2. The commands to enable the Web services consumer UDFs may differ if you are on a different DB2 level. Please refer to the documentation or the following Web site, for more information:

http://www7b.software.ibm.com/dmdd/zones/webservices/wsconsumer/

10.3.2 Enabling DB2 Web service consumer UDFs

To enable the Web service consumer UDFs you first have to enable the database for XML Extender, then run the commands to enable the Web service consumer.

Follow the instructions in “Enabling a database for XML Extender” on page 546 and “Enabling a database with Web service consumer UDFs” on page 546.

10.3.3 Using the Web service consumer UDFs

In this section, we show you how to use the Web service consumer UDFs. We first introduce the Web service consumer UDFs and explain their parameters. We then use the Delayed Stock Quote Web service, provided by *xmethods*, as an example. Finally, we show you how to find out the values of different parameters for the consumer UDFs.

10.3.4 Web service consumer UDFs

The db2xml.soaphttp DB2 UDF is a function that composes a SOAP request, post the request to the service endpoint, receives the SOAP response, and returns the content of the SOAP body. The function is overloaded depending on the soap body to return a VARCHAR or CLOB.

```
db2xml.soaphttpv returns VARCHAR():
       db2xml.soaphttpv (endpoint_url VARCHAR(256),
                         soap_action VARCHAR(256),
                         soap_body VARCHAR(3072)) | CLOB(1M))
                      RETURNS VARCHAR(3072)

db2xml.soaphttpc returns CLOB():
       db2xml.soaphttpc (endpoint_url VARCHAR(256),
                         soapaction VARCHAR(256),
                         soap_body VARCHAR(3072) | CLOB(1M))
                      RETURNS CLOB(1M)

db2xml.soaphttpcl returns CLOB() as locator:
       db2xml.soaphttpcl(endpoint_url VARCHAR(256),
                         soapaction VARCHAR(256),
                         soap_body varchar(3072))
                      RETURNS CLOB(1M) as locator
```

DB2 requires the following information to build a SOAP request and receive a SOAP response:

- Service endpoint, for example:

  `http://services.xmethods.net/soap/servlet/rpcrouter`

- SOAP action URI reference (it is optional and may be a null string)
- XML content of the SOAP body, which are:
  - Name of operation with request namespace URI
  - Encoding style
  - Input arguments

10.3.5 From WSDL to Web service consumer function

The WSDL for the *Delayed Stock Quote Request* Web service listed in Example 10-8 on page 297 describes the details of the Web service interface. It provides the information on how to connect to the Web service and invoke the operation. This WSDL is also available at:

`http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl`

Note again that in this section we show how to create the functions manually to give you a better understanding what pieces are required and how they fit together. WebSphere Studio has tooling that allow you to generate Web service consumer function from WSDL very easily. This is described in more detail in 16.5.1, “Creating a scalar Web service UDF” on page 519.

Example 10-8 WSDL for the Delayed Stock Quote Request Web service

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

<definitions name='net.xmethods.services.stockquote.StockQuote'
   targetNamespace='http://www.themindelectric.com/wsdl
                          /net.xmethods.services.stockquote.StockQuote/'
   xmlns:tns='http://www.themindelectric.com/wsdl
                          /net.xmethods.services.stockquote.StockQuote/'
   xmlns:electric='http://www.themindelectric.com/'
   xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
   xmlns:xsd='http://www.w3.org/2001/XMLSchema'
   xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
   xmlns:wsdl='http://schemas.xmlsoap.org/wsdl/'
   xmlns='http://schemas.xmlsoap.org/wsdl/'>

<message name='getQuoteResponse1'>
   <part name='Result' type='xsd:float'/>
</message>

<message name='getQuoteRequest1'>
   <part name='symbol' type='xsd:string'/>
</message>

<portType name='net.xmethods.services.stockquote.StockQuotePortType'>
   <operation name='getQuote' parameterOrder='symbol'>
      <input message='tns:getQuoteRequest1'/>
      <output message='tns:getQuoteResponse1'/>
   </operation>
</portType>

<binding name='net.xmethods.services.stockquote.StockQuoteBinding'
          type='tns:net.xmethods.services.stockquote.StockQuotePortType'>
   <soap:binding style='rpc'
                transport='http://schemas.xmlsoap.org/soap/http'/>
   <operation name='getQuote'>
      <soap:operation soapAction='urn:xmethods-delayed-quotes#getQuote'/>
      <input>
         <soap:body use='encoded' namespace='urn:xmethods-delayed-quotes'
                   encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
      </input>
      <output>
         <soap:body use='encoded' namespace='urn:xmethods-delayed-quotes'
                   encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
      </output>
   </operation>
</binding>

<service name='net.xmethods.services.stockquote.StockQuoteService'>
   <documentation>net.xmethods.services.stockquote.StockQuote web
                     service</documentation>
   <port name='net.xmethods.services.stockquote.StockQuotePort'
         binding='tns:net.xmethods.services.stockquote.StockQuoteBinding'>
      <soap:address location='http://66.28.98.121:9090/soap'/>
   </port>
</service>

</definitions>
```

Let us analyze the WSDL and find out how to map the elements of this WSDL to parameters of Web service consumer functions.

The `service` section of the WSDL contains the port definition for the SOAP interface:

```
   <port name='net.xmethods.services.stockquote.StockQuotePort'
            binding='tns:net.xmethods.services.stockquote.StockQuoteBinding'>
         <soap:address location='http://66.28.98.121:9090/soap'/>
   </port>
```

The location of `soap:address` shows the service endpoint of the Web service. This is the *first* parameter of the Web service consumer functions. If there are multiple ports for different bindings, you have to find the port with a SOAP binding, for example:

```
   <soap:binding transport="http://schemas.xmlsoap.org/soap/http " .../>
```

The `binding` section of the WSDL lists all the operations of the service.

```
   <binding name='net.xmethods.services.stockquote.StockQuoteBinding'
            type='tns:net.xmethods.services.stockquote.StockQuotePortType'>
   <soap:binding style='rpc'transport='http://schemas.xmlsoap.org/soap/http'/>
      <operation name='getQuote'>
         <soap:operation soapAction='urn:xmethods-delayed-quotes#getQuote'/>
            ...
            ...
      </operation>
   </binding>
```

The `soapAction` of the desired Web service operation is the value of the *second* parameter of the Web service consumer functions. In our case we are interested in invoking the getQuote operation and hence the relevant soapAction as described in the WSDL is urn:xmethods-delayed-quotes#getQuote.

Now we are left with finding the structure of the soap body, the *third* argument of the Web service consumer functions. The portType definition provides you this information. The portType may define many operations. Each operation typically contains an input and an output message element. Occasionally it may contain fault elements as well. For simplicity we consider only input and output elements in our discussion.

```
<portType name='net.xmethods.services.stockquote.StockQuotePortType'>
   <operation name='getQuote' parameterOrder='symbol'>
      <input message='tns:getQuoteRequest1'/>
      <output message='tns:getQuoteResponse1'/>
   </operation>
</portType>
```

You can construct the structure of the soap body using the operation element of the portType. The name of the operation becomes the top-level node. You then have to flatten out the input or output message element to get the rest of the structure. Thus in our case the getQuote forms the top-level element and the flattening of input message getQuoteRequest1 yields the second level element symbol. In many cases this flattening of input or output message may yield quite complex structure. Thus in our case the structure of the soap body becomes:

```
<stock:getQuote xmlns:stock="urn:xmethods-delayed-quotes"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <symbol xsi:type="xsd:string">Stock symbol to be supplied</symbol>
</stock:getQuote>
```

This is the third parameter of the Web service consumer functions for our example.

Similarly you can find out the structure of the result by flattening the output message element. In our case it looks as follows:

```
<n:getQuoteResponse xmlns:n="urn:xmethods-delayed-quotes">
   <Result xsi:type="xsd:float">Quote value returned</Result>
</n:getQuoteResponse>
```

Table 10-1 on page 300 summarizes our discussion and provides hints for finding the different parameter values for the Web service consumer functions.

Table 10-1 From WSDL to Web service consumer function parameters

No	Parameter	XPath to look for
1	Service endpoint URL	`/definition/port/soap:address/@location`
2	SOAP action	`/definitions/binding/soap:binding/operation/ soap:operation/@soapAction`
3	SOAP body	`/definitions/portType/operation`

10.3.6 Testing the Web service consumer functions

Let us now test the Web service consumer functions by accessing the stock quote of IBM:

1. Connect to the database that you enabled for SOAP functions (`SAMPLE` or `INSURA` in our case). To enable these functions, see a “Enabling a database with Web service consumer UDFs” on page 546
2. Type the following command in a DB2 command line and execute it:

```
VALUES db2xml.soaphttpv ('http://66.28.98.121:9090/soap',
   'urn:xmethods-delayed-quotes#getQuote',
   varchar ('<stock:getQuote xmlns:stock="urn:xmethods-delayed-quotes"
        SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
            <symbol xsi:type="xsd:string"> IBM </symbol>
         </stock:getQuote>'))
```

Note: If you want to enter this as a DB2 command, the easiest is to put the command in a file (stockquote.txt) with an ending semicolon, and execute: `db2 -tf stockquote.txt`

3. You should be able to see the XML result, similar to this:

```
<n:getQuoteResponse xmlns:n="urn:xmethods-delayed-quotes">
   <Result xsi:type="xsd:float">83.52</Result></n:getQuoteResponse>
```

4. If you are running DB2 UDB V8.1, you can also use an SQL/XML query to construct the soap body. The example that follows is the same as the previous, except that it is written in SQL/XML and uses the overloaded `soaphttpv` function that takes a soap body as a CLOB:

```
   VALUES db2xml.soaphttpv
      ('http://66.28.98.121:9090/soap',
       'urn:xmethods-delayed-quotes#getQuote',
       xml2clob
        (XMLElement(name "getQuote",
                 XMLAttributes('urn:xmethods-delayed-quotes' AS "xmlns:stock"),
                 XMLElement(name "symbol", 'IBM') )
        )
      )
```

10.3.7 Creating a wrapper UDF to Web services consumer function

In order for applications to work with DB2 and Web services seemlessly, we make Web services calls using the DB2 UDFs that we installed and enabled.

However, invoking these UDFs clutters the application code and reduces the readability and understandability of the code. Also, every time we call the Web service we have to pass the URL of the Web service and the operation name. Therefore, we can wrap the call to a Web service in a UDF and invoke this UDF to invoke a Web service operation.

In our case, instead of calling the `db2xml.soaphttpv(...)` for a stock quote request, we create a wrapper UDF that takes only the stock symbol as parameter and internally calls the `db2xml.soaphttpv` function.

Let us define our wrapper UDF `getStockQuote`, which takes only one parameter (`symbol`) as a `VARCHAR(256)` and returns `Result` as `FLOAT`. Example 10-9 shows the code for our `getStockQuote` UDF. The result of the call to `db2xml.soaphttpv` is a CLOB; we extract the value contained in the *Result* element by using the XPath `/*/Result`.

Example 10-9 GetStockQuote UDF

```
CREATE FUNCTION itso.getStockQuote (symbol VARCHAR(100))
    RETURNS DECIMAL(5,2) SPECIFIC xmethods_getQuote
    LANGUAGE SQL CONTAINS SQL
    EXTERNAL ACTION NOT DETERMINISTIC
    RETURN
    db2xml.extractREAL(
      db2xml.xmlclob(
         db2xml.soaphttpv(
             'http://66.28.98.121:9090/soap',
             'urn:xmethods-delayed-quotes#getQuote',
             varchar(
                '<m:getQuote xmlns:m="urn:xmethods-delayed-quotes"
                   SOAP-ENV:encodingStyle=
                                "http://schemas.xmlsoap.org/soap/encoding/">
                   <symbol xsi:type="xsd:string">' || symbol || '</symbol>
                </m:getQuote>'
             )
          )
      ), '/*/Result'
   )
```

This UDF can be invoked in a DB2 command window as follows:

```
DB2 VALUES itso.getStockQuote('IBM')
```

In Example 10-9 on page 301 we saw how to invoke a consumer UDF for Web services returning one value. DB2 also allows to write UDFs that invoke Web services returning multiple results. In such cases we return a table with each row containing one result element.

For example, take the Web service *Xignite News* provided by `http://www.xignite.com/xnews.asmx`. The WSDL for this Web service is available at `http://www.xignite.com/xnews.asmx?WSDL`. One of the operations provided by this Web service is `GetStockHeadlines`, with two parameters (list of symbols, number of headlines wanted). The service returns an XML document in the format:

```
<GetStockHeadlinesResult>
<StockNews>
   <Outcome>Success</Outcome><Headline>a headline</Headline>
   <Ticker>stocksymbol</Ticker><Date>date</Date><Time>time</Time>
   <Source>source of headline</Source><Url>url for information</Url>
</StockNews>
<StockNews>......</StockNews>
</GetStockHeadlinesResult>
```

We write a UDF to access this Web service operation and extract the headline element (Example 10-10).

Example 10-10 Web service consumer UDF wrapper returning a table

```
CREATE FUNCTION itso.GetStockHeadlines (Symbols VARCHAR(100),
                                      HeadlineCount INTEGER)
    RETURNS TABLE (StockNews VARCHAR(3000))
    LANGUAGE SQL CONTAINS SQL
    EXTERNAL ACTION NOT DETERMINISTIC
    RETURN
    Select * from Table (db2xml.extractVarchars(
      DB2XML.XMLCLOB(
         db2xml.soaphttpc(
            'http://www.xignite.com/xnews.asmx',
            'http://www.xignite.com/services/GetStockHeadlines',
            '<m:GetStockHeadlines xmlns:m="http://www.xignite.com/services/"
               SOAP-ENV:encodingStyle=
                         "http://schemas.xmlsoap.org/soap/encoding/">'
               || '<m:Symbols>' || Symbols || '</m:Symbols>'
               || '<m:HeadlineCount>' || rtrim( char(HeadlineCount) ) ||
                                                   '</m:HeadlineCount>'
            || '</m:GetStockHeadlines>'
         )
      ), '//StockNews/Headline'
  )) as X ;
```

In this wrapper UDF, we extract only the Headline elements from the result of the Web service, construct a table from these elements, and return the table.

You can invoke this UDF as follows:

```
db2 select substr(stocknews,1,79) as headlines
      from table ( itso.getStockheadlines('IBM',10) ) as y
```

By issuing this query, we are seeking for 10 or less news headlines related to IBM stock. A sample result is shown in Example 10-11:

Example 10-11 Sample result

```
HEADLINES
-------------------------------------------------------------------------
UPDATE - AMD shares jump on hopes for new chips
Open Text says poised for consolidating market
Horror Story
[external] HP Revenue May Come In A Little Light
The Web-Content Management Boom
State of the Art: Server Load Balancing
[external] Apple's G5 In Stores Today
IBM Cuts 600 Jobs from Microchip Division
[external] IBM Claims Wins Over H-P in Server Wars
UPDATE - Post-9/11 steps help business computers in outage
```

11

XML wrapper

In this chapter, we discuss the XML wrapper functionalities; this includes:

- An introduction to the XML wrapper
- A detailed discussion on how to set up, and use the XML wrapper
- A discussion about using the XML wrapper from the DB2 Control Center
- A set of best practices when using the XML wrapper

11.1 Introducing the XML wrapper

The XML wrapper technology lets you view an XML data source as a relational data source by creating a wrapper, for the XML data source, in your federated system. In Chapter 4, "Storing whole XML documents" on page 93 and Chapter 6, "Shredding XML into relational tables" on page 143, we discussed the storage options available in DB2 XML Extender while working with XML documents. We suggested to:

- Use XML collection when:
  - Partial updates of your XML document are very frequent
  - The structure of your XML document is fairly regular
- Use XML column when:
  - You need to store your XML document as a whole, and/or
  - The structure of XML documents is not regular except for a few elements

Now, what if your requirement is such that neither of the above mentioned storage options is adequate? For instance, this could be the case if:

- Your XML document needs to be stored as a whole and
- The XML documents are stored in a relational column of another database and we do not want to replicate them in our database and be burdened with synchronizing our XML documents with the ones stored in the other database.
- Some or most of your XML documents are not stored in a local file system or a database but are available on some Web site as Web pages (maybe owned by you or someone else, to which you have read only access).
- The volume of existing XML documents is so huge that you don't want to load them into a relational database, but would still like to view the elements of these XML documents as relational entities.

If you are facing some or all of the above mentioned requirements, then you have found the right page of the right book! In the following sections, we provide an introduction to XML wrapper and present a few examples to showcase some of the commonly used features of XML wrapper. We also show you how to create XML wrapper definitions using DB2 Control Center.

11.2 XML wrapper explained

In this section, we assume that you are familiar with the concept of data federation and DB2 wrappers. For more details, see *Data Federation with IBM DB2 Information Integrator V8.1*, SG24-7052.

XML wrapper is a powerful tool to map XML documents into the relational world of DB2 II federated systems without the need to load/unload data. Figure 11-1 gives an overview of the XML wrapper architecture.

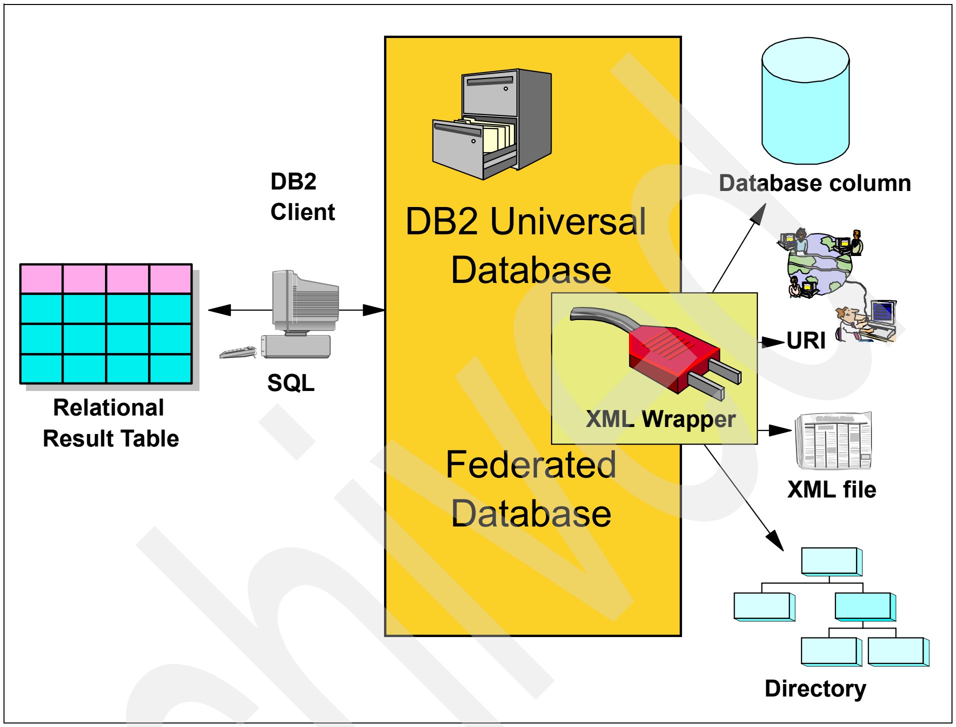


Figure 11-1 Overview of XML wrapper

The XML wrapper functionality ships as part of the DB2 Information Integrator product, and its functionality is not available on the z/OS or iSeries™ platform. Information on how to install DB2 Information Integrator can be found in, "DB2 Information Integrator" on page 547.

As can be seen from Figure 11-1, an XML wrapper is built around an XML document, a directory in the file system, a database column or a Universal Reference Identifier (URI). The wrapper maps the entities of the XML world (elements and attributes) to those in the relational world (tables and columns), thereby acting as a mediator between the XML document and the DB2 database. In fact there is no restriction in DB2 that a wrapper be associated with exactly one XML document. The wrapper can be associated with:

- An individual XML document stored in file system
- A set of XML documents contained in a directory
- XML document stored in a database column
- XML document accessible through a URI

When we create XML-to-relational mapping using the XML wrapper, we observe the following:

- Hierarchical relationships in the XML data are preserved in the parent-child relationship of nicknames
- Individual XML elements map to columns through standard XPath syntax
- Sequences of XML elements map to nicknames through XPath syntax

11.2.1 Using the XML wrapper

In order to view the XML data as relational data, you need to do the following:

1. Register the XML wrapper with the federated system
2. Create a server using the wrapper registered above
3. Create nicknames specifying the XML-to-relational mapping
4. Create federated views for non-root nicknames

You can then start using the nicknames and views just like any other relational entity. However, using these nicknames and views, you can only extract the XML data for processing with read-only access. That is, you cannot insert or update XML data using the XML wrapper.

11.2.2 Registering the XML wrapper

A wrapper is a mechanism that federated servers use to communicate with and retrieve data from a category of data sources. wrappers are installed on your system as library files. You can register the wrapper using the CREATE WRAPPER SQL statement.

For example, to register the XML wrapper called `my_wrap` on a *Windows system* from the default library file, `db2lsxml.dll`, issue the following SQL statement on the DB2 command line:

```
CREATE WRAPPER my_wrap LIBRARY 'db2lsxml.dll'
```

You may optionally set the DB2_DJ_COMM environment variable to improve query performance.This variable determines whether the federated server loads the wrapper during initialization. To set the DB2_DJ_COMM DB2 profile variable, use the db2set command with the wrapper library that corresponds to the wrapper that you specified in the associated CREATE WRAPPER statement. In our case we issue the following command:

```
db2set DB2_DJ_COMM='db2lsxml.dll'
```

Ensure that there are no spaces on either side of the equal sign (=).

Processor usage of your system increases when the federated server loads the wrapper libraries during database startup. To avoid excessive usage, specify only the libraries that you intend to access.

11.2.3 Creating the server

After you register the wrapper, you should create a corresponding server. A server defines a data source to a federated database. You can create the server using the CREATE SERVER SQL statement.

For example, to create a server called `xml_server` using our wrapper my_wrap, you issue the following SQL statement on the DB2 command line:

```
CREATE SERVER xml_server WRAPPER my_wrap
```

Note that we do not specify the target data source (XML data source in our case) anywhere, when defining the server. We do this when we define the nicknames.

11.2.4 Creating nicknames

In order to map your XML data to DB2's relational entities, you create nicknames that correspond to the tree structure of your XML data source. A *root nickname* is the top-level nickname in the nickname hierarchy. (It may or may not reference the root element in the XML document.) *Parent nicknames* map to the parent elements of the XML document. *Child nicknames* correspond to the elements that are nested within the element for the parent nickname.

In this section, we take a simple XML document, listed in Example 11-1, as our target data source, and explain different ways of creating nicknames.

Example 11-1 Customers.xml

```
<?xml version="1.0" ?>

   <customer id='123'>
      <name>Shrinivas Kulkarni</name>
      <address>IBM Bangalore</address>

      <order>
         <amount>200</amount>
         <date>2003-03-01</date>
         <item quant='12'>
            <name>Notebook</name>
         </item>
         <item quant='4'>
            <name>Colored Pen</name>
         </item>
      </order>

      <order>
      </order>

      <payment>
         <number>BGM1877</number>
         <date>2003-04-01</date>
      </payment>
   </customer>
```

To create the root nickname called `customers`, issue the following command:

Example 11-2 Create the root nickname: customers

```
CREATE NICKNAME customers
(
  id VARCHAR(5) OPTIONS (XPATH '@id'),
  cname VARCHAR(50) OPTIONS (XPATH './name/text()'),
  cid VARCHAR(20) OPTIONS (PRIMARY_KEY 'YES')
)
FOR SERVER xml_server
  OPTIONS (XPATH '/customer',
           FILE_PATH 'C:\Shrini\Customers.xml')
```

This statement creates the nickname `customers` over the XML file specified in the `FILE_PATH` (`C:\Shrini\Customers.xml`). Let us analyze the statement.

We are creating a nickname called `customers` containing the columns `id`, `name` and `cid`. As `OPTIONS` for the `id` and `name` columns, we provide the information to populate the values these columns will contain, from the XML document mentioned in the `FILE_PATH`. We provide this information using XPATH syntax. These XPATH values(`./name` and `@id`) are relative to the XML element specified by the XPATH clause in the OPTIONS of the FOR SERVER clause (/customer).

Tip: We use the "/customer" XPATH expression. We could have used the "//customer" XPATH expression. It is correct and convenient. However, it is not very efficient. The self-or-descendant operator "//" is very costly. (The XPath processor has to recursively parse the entire document). Therefore, if possible, we encourage you use absolute XPath expressions.

Only the root nicknames can contain XML document location information. All other child nicknames refer to this or other parent nicknames (generally through a foreign key).

Note that the `cid` primary key column does not actually exist in the XML document. It is generated by the system when the XML document is accessed to reflect the parent-child relationships inside the rest of the XML document.

We can now create nicknames for the children of the `customers` nickname, that is, `orders`, `payments`, and `items`.

Issue the following statement to create the `orders` nickname.

Example 11-3 Create the non-root nickname: orders

```
CREATE NICKNAME orders
(
   amount   INTEGER       OPTIONS(XPATH './amount/text()'),
   date     VARCHAR(10)   OPTIONS(XPATH './date/text()'),
   oid      VARCHAR(20)   OPTIONS(PRIMARY_KEY 'YES'),
   cid      VARCHAR(20)   OPTIONS(FOREIGN_KEY 'CUSTOMERS')
)
FOR SERVER xml_server
   OPTIONS(XPATH './order')
```

Note that the `oid` primary key column and `cid` foreign key column do not actually exist in the XML document. They are generated by the wrapper when the XML document is accessed to reflect the parent-child relationships inside the XML document.

Issue the following command to create the `payments` nickname.

Example 11-4 Create the non-root nickname: payments

```
CREATE NICKNAME payments
(
   number   INTEGER       OPTIONS(XPATH './number/text()'),
   date     VARCHAR(10)   OPTIONS(XPATH './date/text()'),
   cid      VARCHAR(16)   OPTIONS(FOREIGN_KEY 'CUSTOMERS')
)
FOR SERVER xml_server
OPTIONS( XPATH './payment')
```

Issue the following command to create the `items` nickname.

Example 11-5 Create the non-root nickname: ITEMS

```
CREATE NICKNAME items
(
   cname        VARCHAR(20)   OPTIONS(XPATH './name/text()'),
   quantity    INTEGER       OPTIONS(XPATH './@quant'),
   oid         VARCHAR(16)   OPTIONS(FOREIGN_KEY 'ORDERS')
)
FOR SERVER xml_server
OPTIONS( XPATH './item')
```

Now all the XML elements of our customers.xml document (XML data source for the XML wrapper) are mapped to relational entities, and we can query the XML data source just as if it were a relational data source.

For example, we can issue the following SQL statement from the DB2 command line (Example 11-6) to retrieve all items ordered by all customers.

Example 11-6 Results of SQL query using XML wrapper

```
db2 select c.cname, i.name, i.quantity
      from customers c, orders o, items i
      where c.cid = o.cid
        and o.oid = i.oid

CNAME                 NAME                         QUANTIFY
--------------------- ---------------------------- ----------
Shrinivas Kulkarni    Notebook                             12
Shrinivas Kulkarni    Colored Pen                           4

  2 record(s) selected.
```

11.2.5 Accessing non-root nicknames

We know that in a federated system, a nickname is just like a database table or a view, and we can perform any legal operation that we do on a regular database table or a view. So logically, the nicknames we defined in the previous section should be no exception. Is this the case? Let us try the very simple query given below:

```
SELECT o.date, o.amount FROM orders o
```

Naturally, we would expect two rows: one containing null values and another containing valid values “2003-03-01” and “200” for DATE and AMOUNT columns respectively. However, when we run this query, we get the following message (Example 11-7):

Example 11-7 Error message for accessing a non-root nickname

```
DBA2191E SQL execution error.

com.ibm.db.DataException: A database manager error occurred.
: [IBM][CLI Driver][DB2/NT]
SQL0901N  The SQL statement failed because of a non-severe system error.
Subsequent SQL statements can be processed.
(Reason "sqlno_crule_save_plans    [100]:rc(0)".)
SQLSTATE=58004
```

What could have gone wrong? Well, the answer is that you cannot access non-root nicknames stand-alone. If you remember, while creating the non-root nicknames, we never mentioned the location of our XML document. So DB2 does not know from where to fetch the XML data and hence it fails, throwing the above error message. The next question is: what do we do if we want the above query and similar ones to work?

The solution is to perform a *join* with the root nickname as part of the query. So our revised query looks like this (Example 11-8):

Example 11-8 Querying non-root nickname with JOIN with root nickname

```
SELECT o.date, o.amount
  FROM orders o, customers c
  WHERE o.cid = c.cid
```

Now try running the query given in Example 11-8 on page 313. You should see the proper results, as given in Table 11-1.

Table 11-1 Results of query

DATE	AMOUNT
2003-03-01	200
-	-

11.2.6 Creating federated views for non-root nicknames

In the previous section, we discussed that nicknames cannot appear in a query without a join to the root nickname. To avoid having to code this join in each query, we can simplify our queries by defining *federated views* so that not every query containing a non-root nickname must explicitly code the join operation. Defining federated views ensures that the queries that join pieces of an XML nickname hierarchy but do not include the root nickname, run properly. The same is true for queries that join columns but, not code the join on the "special" PRIMARY_KEY and FOREIGN_KEY columns.

To define federated views that include all required predicates and a full path to the top node, follow these steps:

1. Define a view for each non-root nickname as a join of all the nicknames on the path to the root
2. In the WHERE clause, make the join predicates over the PRIMARY_KEY and FOREIGN_KEY columns of the subsequent nicknames
3. In the SELECT list, include all the columns of the non-root nickname except the column that is designated with the FOREIGN_KEY nickname column option.
4. In the SELECT list, include the column of the parent nickname designated with the PRIMARY_KEY option.

Following the above guidelines, we define our view for the non-root nickname orders as follows:

Example 11-9 Creating federated view for non-root nicknames

```
CREATE VIEW order_view AS
       SELECT o.amount, o.date, o.oid, c.cid
         FROM customers c, orders o
         WHERE c.cid = o.cid
```

Now we can run the following query to get the results shown in Table 11-1 on page 314.

```
SELECT o.date, o.amount FROM order_view o
```

Similarly, you can define a view to easily access the items nickname as follows (Example 11-10):

Example 11-10 Creating items_view

```
CREATE VIEW items_view AS
   SELECT i.name, i.quantity, o.oid
     FROM customers c, orders o, items i
     WHERE c.cid = o.cid
       and o.oid = i.oid
```

11.2.7 Options for specifying the XML data source for nicknames

When creating a root nickname using the CREATE NICKNAME statement, we can specify the location of the XML data source in multiple ways. In “Creating nicknames” on page 309, we showed how to specify an individual XML document as a data source. In this sub-section we explain different ways of specifying the XML data source.

XML data source as a hard-coded file name

This is the simplest form of XML data source that we discussed in “Creating nicknames” on page 309. Sample code for creating a nickname for an individual XML file is given in Example 11-11.

Example 11-11 Create the root nickname using the FILE_PATH option

```
CREATE NICKNAME customers
(
  id VARCHAR(5) OPTIONS (XPATH '@id'),
  cname VARCHAR(50) OPTIONS (XPATH './name/text()'),
  cid VARCHAR(20) OPTIONS (PRIMARY_KEY 'YES')
)
FOR SERVER xml_server
  OPTIONS (XPATH '/customer',
           FILE_PATH 'C:\Shrini\Customers.xml')
```

The full XML file (directory and file name) is specified in the FILE_PATH option.

XML data source as a set of XML files in a fixed directory

We can also specify the name of a file system directory as XML data source. All XML files in this directory with a .xml extension make up the data source. Whenever a query is run using this nickname, all the XML files in this directory are parsed and analyzed to produce the results of the query. If the XML data having similar structure is scattered among multiple files, then you should probably consider using this option. Sample code for creating a nickname for a set of XML files in a directory is provided in Example 11-12.

Example 11-12 Create the root nickname using the DIRECTORY_PATH option

```
CREATE NICKNAME customers
(
   id          VARCHAR(5)   OPTIONS(XPATH './@id')
   cname        VARCHAR(16)  OPTIONS(XPATH './name/text()'),
   cid         VARCHAR(16)  OPTIONS(PRIMARY_KEY 'YES')
)
FOR SERVER xml_server
   OPTIONS(XPATH '/customer'
           DIRECTORY_PATH 'C:\Shrini')
```

The directory containing the XML files is specified in the `DIRECTORY_PATH` option. The XML wrapper uses only those files with an .xml extension that are located in the directory that you specify. The XML wrapper ignores all other files in this directory.

If you specify either the `DIRECTORY_PATH` or `FILE_PATH` nickname option, you should not specify a DOCUMENT column.

XML data source as a parameterized file name

With the DOCUMENT option, we can create a wrapper around an individual XML file. However, in this case, the name of the file is not statically bound when the nickname is created, but will be provided at the runtime as a parameter. Example 11-13 on page 317 shows a sample of how to create a nickname using this option.

Example 11-13 Create the root nickname using the DOCUMENT option FILE

```
CREATE NICKNAME customers
(
   doc      VARCHAR(100)   OPTIONS(DOCUMENT 'FILE'),
   cname     VARCHAR(16)    OPTIONS(XPATH './name/text()'),
   cid      VARCHAR(16)    OPTIONS(PRIMARY_KEY 'YES')
)
   FOR SERVER xml_server
     OPTIONS(XPATH '/customer')
```

`doc` is an additional column in the nickname that is used to reference the actual XML file at runtime. When you run a query against the customers nickname, you can specify the location of the XML document in the WHERE clause:

```
SELECT * FROM customers WHERE doc = 'C:\Shrini\Customers.xml'
```

XML data source as a parameterized directory name

This DOCUMENT option is similar to the one discussed above, with the only difference that a directory name is specified instead of a file name, and the wrapper looks up for all the XML files contained in the specified directory . Example 11-14 shows an example of how to create a nickname using this option.

Example 11-14 Create the root nickname using the DOCUMENT option DIRECTORY

```
CREATE NICKNAME customers
(
   doc      VARCHAR(100)   OPTIONS(DOCUMENT 'DIRECTORY'),
   cname     VARCHAR(16)    OPTIONS(XPATH './name/text()'),
   cid      VARCHAR(16)    OPTIONS(PRIMARY_KEY 'YES')
)
   FOR SERVER xml_server
     OPTIONS(XPATH '/customer')
```

You can then run the following query against the customers nickname, specifying the directory containing the XML documents in the WHERE clause:

```
SELECT * FROM customers WHERE doc = 'C:\Shrini’
```

XML data source as a URI

The XML wrapper also allows you to specify a URI as an XML data source. The URI address indicates the (remote) location of the XML file, for example on the Web. Example 11-15 shows sample code that creates a nickname using this option.

Example 11-15 Create the root nickname using the DOCUMENT option URI

```
CREATE NICKNAME customers
(
   doc      VARCHAR(100)   OPTIONS(DOCUMENT 'URI'),
   cname     VARCHAR(16)    OPTIONS(XPATH './name/text()'),
   cid      VARCHAR(16)    OPTIONS(PRIMARY_KEY 'YES')
)
   FOR SERVER xml_server
     OPTIONS(XPATH '/customer')
```

You can then run the following query on the customers nickname to retrieve the XML data from the remote location:

```
SELECT * FROM customers WHERE doc = 'http://www.mycorp.com/results.xml'
```

XML data source as a database column

The XML wrapper lets you access a column of a relational database as an XML data source. The column that contains the XML document is specified at runtime. Example 11-16 shows sample code that creates a nickname using this option.

Example 11-16 Create the root nickname using the DOCUMENT option COLUMN

```
CREATE NICKNAME customers
(
   doc      VARCHAR(100)   OPTIONS(DOCUMENT 'COLUMN'),
   cname     VARCHAR(16)    OPTIONS(XPATH './name/text()'),
   cid      VARCHAR(16)    OPTIONS(PRIMARY_KEY 'YES')
)
   FOR SERVER xml_server
     OPTIONS(XPATH '/customer')
```

You can then run the following query on the `customers` nickname to retrieve the XML data (Example 11-17 on page 319).

Example 11-17 Query using DOCUMENT option COLUMN

```
SELECT *
FROM customers c, xml_data x
WHERE c.doc = x.data
```

In the above query, xml_data is a valid table or a view or a nickname, and data is relational column that contains XML data.

You can also run the following query (Example 11-18) to get similar results.

Example 11-18 Query using COLUMN option with XML data as part of the query

```
SELECT *
FROM customers c
WHERE c.doc = '<?xml version="1.0" encoding="UTF-8"?>
        <customer id='123'>
           <name>Shrinivas Kulkarni</name>
        </customer>
        <customer id='456'>
           <name>Bart Steegmans</name>
        </customer>'
```

Restriction: When using the DOCUMENT ‘COLUMN’ option, you can only process a single XML document at a time (in a single query). You have to make sure that your query results in a single row being retrieved from the table that contains the column with the XML documents. This restriction may be lifted in a future release.

11.2.8 Altering XML nicknames

You can use the **ALTER NICKNAME** statement to modify the federated database representation of a data source. Use this statement to:

- Change local data type of a column:

  `alter nickname customers alter column cname local type varchar (45)`

- Change local column name:

  `alter nickname customers alter column cname local name CUSTOMER_NAME`

- Add the streaming option (since DB2 V8.1 FixPak 3):

  `alter nickname customers options (ADD STREAMING 'YES')`

- Drop the streaming option:

  ```
  alter nickname customers options (DROP STREAMING)
  ```

More information on the streaming option can be found in “Handling large XML documents” on page 332.

11.3 Working with XML wrapper via DB2 Control Center

In this section we show you how to create an XML wrapper, server and nicknames using the DB2 Control Center GUI. Manually creating nicknames is a tedious and laborious task if the structure of the XML document is complex. You also need to understand XPATH really well if you want to map the XML elements and attributes correctly to nickname columns. The DB2 Control Center automates most of the work, leaving you to just selecting which elements and attributes to map, rather than how to map them.

1. Launch the DB2 Control Center. Click **All Cataloged Systems** -> **<NodeName>** -> **Instances** -> **DB2** -> **Databases** -> **<databasename>**. You should see a window similar to the one shown in Figure 11-2 on page 321.

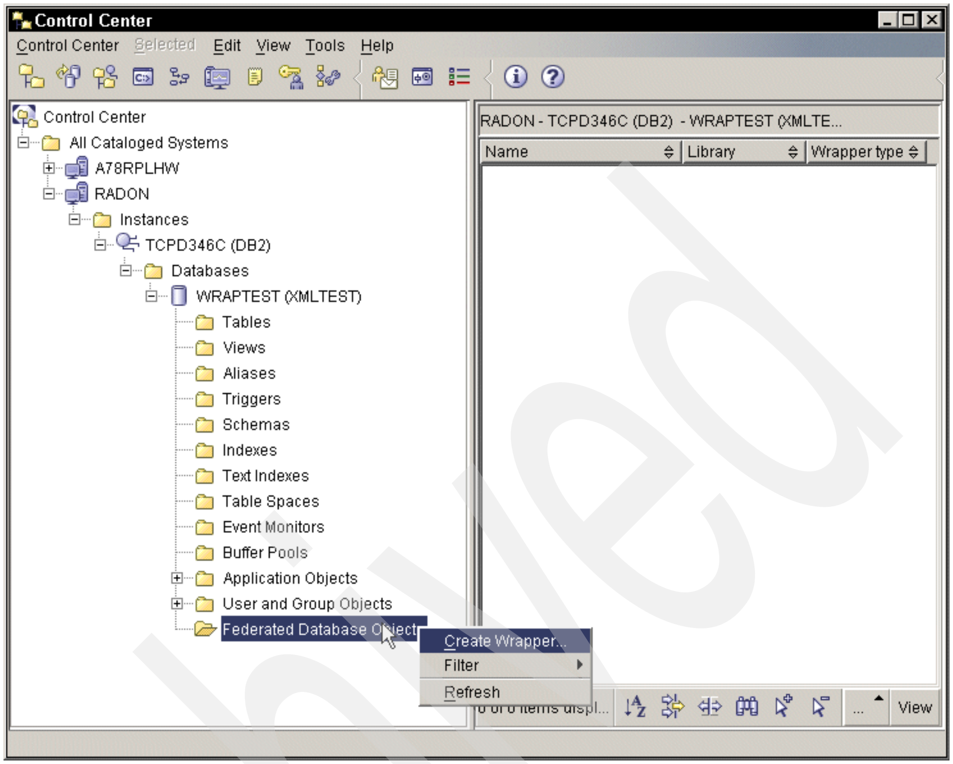
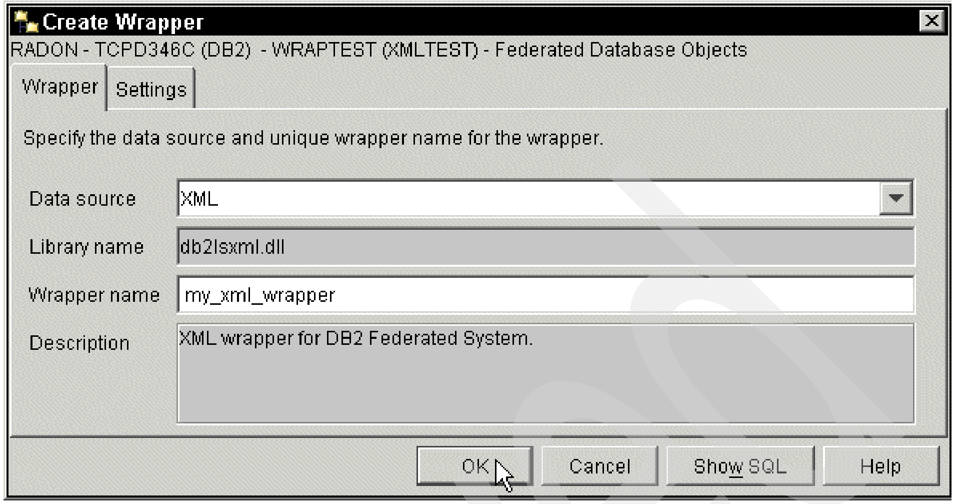


Figure 11-2 Creating nicknames using DB2 Control Center

2. Right-click **Federated Database Objects**. Click **Create Wrapper...** This opens up the dialog shown in Figure 11-3 on page 322.



RADON - TCPD346C (DB2) - WRAPTEST (XMLTEST) - Federated Database Objects

Specify the data source and unique wrapper name for the wrapper.

Data source: XML

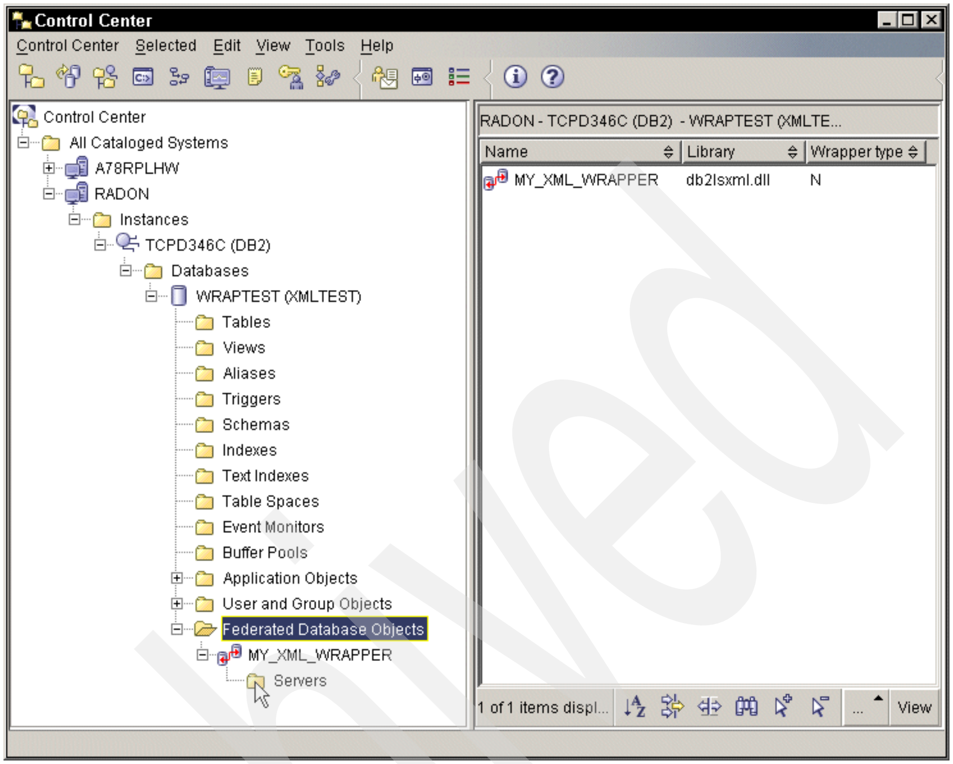
Library name: db2lsxml.dll

Wrapper name: my_xml_wrapper

Description: XML wrapper for DB2 Federated System.

Figure 11-3 Create wrapper dialog

3. Select the **Data source** value as `XML` from the pull-down menu. The selection automatically fills the Library name field with `db2lsxml.dll`. Enter the wrapper name of choice in the Wrapper name text field. The figure shows the value `my_xml_wrapper` being entered. Now click the **OK** button to create the XML wrapper. If the wrapper is successfully created, you should see a window similar to the one shown in Figure 11-4 on page 323.

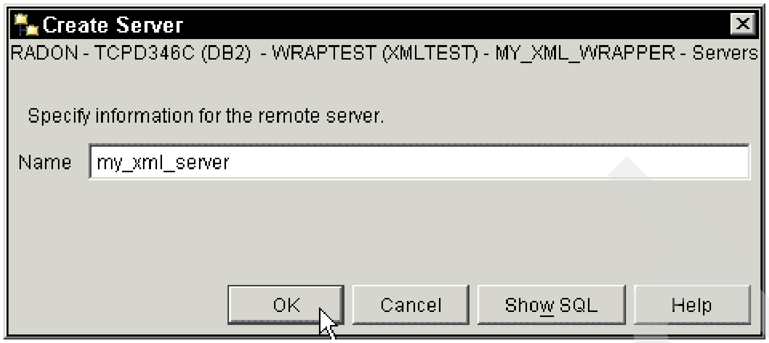


Name	Library	Wrapper type
MY_XML_WRAPPER	db2lsxml.dll	N

Figure 11-4 XML wrapper successfully created

If not, you may want to right-click the **Federated Database Objects** again and select the **Refresh** option.

4. Now right-click **Servers**, listed under the XML wrapper we just created. Select **Create...** and click it to open the Create Server dialog as shown in Figure 11-5 on page 324.



Create Server

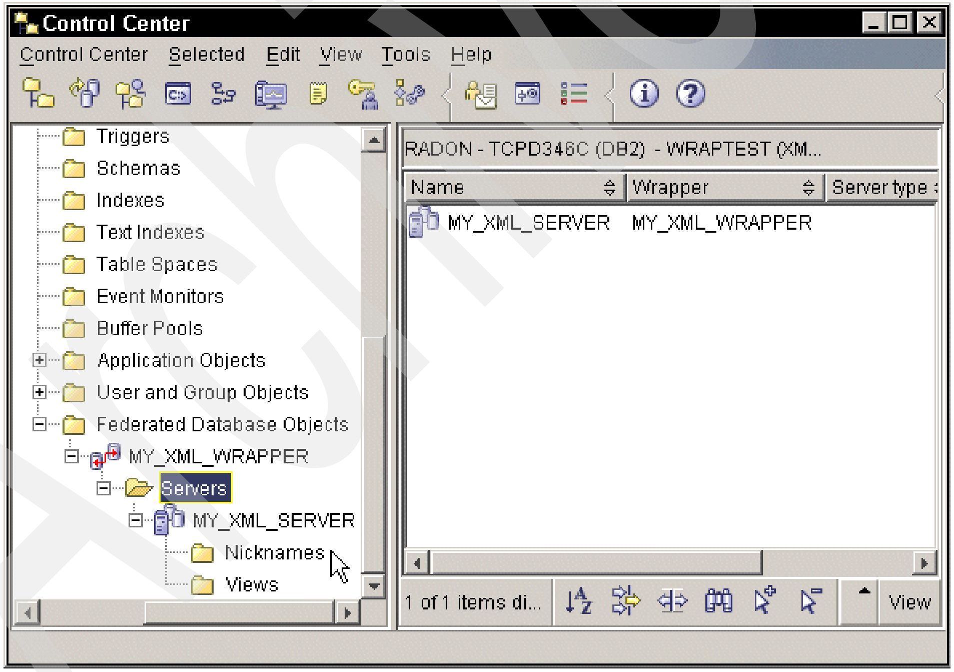
RADON - TCPD346C (DB2) - WRAPTEST (XMLTEST) - MY_XML_WRAPPER - Servers

Specify information for the remote server.

Name: my_xml_server

Figure 11-5 Create Server dialog

5. Enter the name of the server in the Name field. We use `my_xml_server`. Then click **OK**. The server gets created and you should see a window similar to the one shown in Figure 11-6.



Control Center

RADON - TCPD346C (DB2) - WRAPTEST (XM...

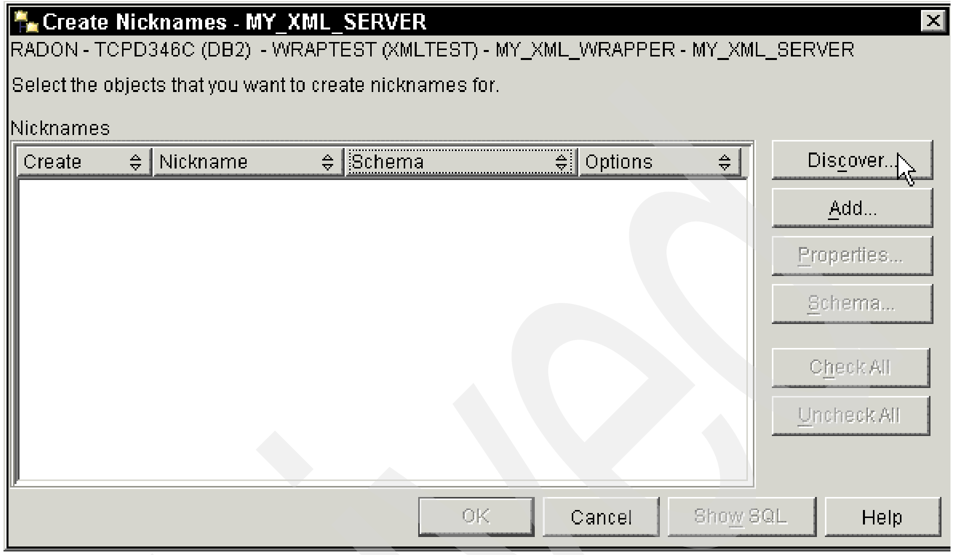
Name	Wrapper	Server type
MY_XML_SERVER	MY_XML_WRAPPER	

1 of 1 items di...

Figure 11-6 Server created successfully

Now we can create the nicknames on this server. We show how to create one nickname. You can create the rest of the nicknames similarly. Here we show how to create the root nickname for our sample XML file we used in “Creating nicknames” on page 309.

6. Right-click **Nicknames**. Select **Create** and click it to open up the Create Nicknames dialog as shown in Figure 11-7.



Create Nicknames - MY_XML_SERVER

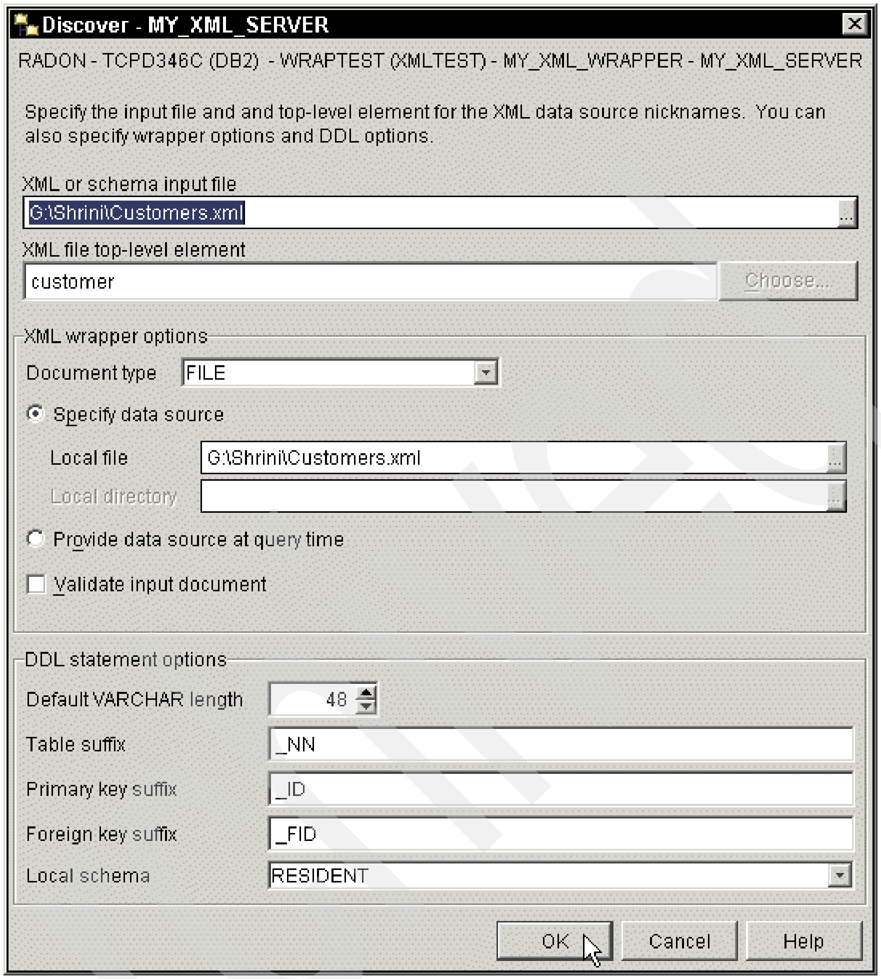
RADON - TCPD346C (DB2) - WRAPTEST (XMLTEST) - MY_XML_WRAPPER - MY_XML_SERVER

Select the objects that you want to create nicknames for.

Figure 11-7 Create Nicknames dialog

We can either specify the XML element to relational mapping manually by clicking the **Add** button or use the tool to automatically discover possible mappings by clicking the **Discover** button. Let us see how much our mapping work is simplified by the automated tool.

7. Click the **Discover** button. It opens up the window shown in Figure 11-8 on page 326.



Discover - MY_XML_SERVER

RADON - TCPD346C (DB2) - WRAPTEST (XMLTEST) - MY_XML_WRAPPER - MY_XML_SERVER

Specify the input file and and top-level element for the XML data source nicknames. You can also specify wrapper options and DDL options.

XML or schema input file: G:\Shrini\Customers.xml

XML file top-level element: customer

XML wrapper options

Document type: FILE

● Specify data source

Local file: G:\Shrini\Customers.xml

Local directory:

○ Provide data source at query time

□ Validate input document

DDL statement options

Default VARCHAR length: 48

Table suffix: _NN

Primary key suffix: _ID

Foreign key suffix: _FID

Local schema: RESIDENT

Figure 11-8 Discover XML-to-relational mapping automatically

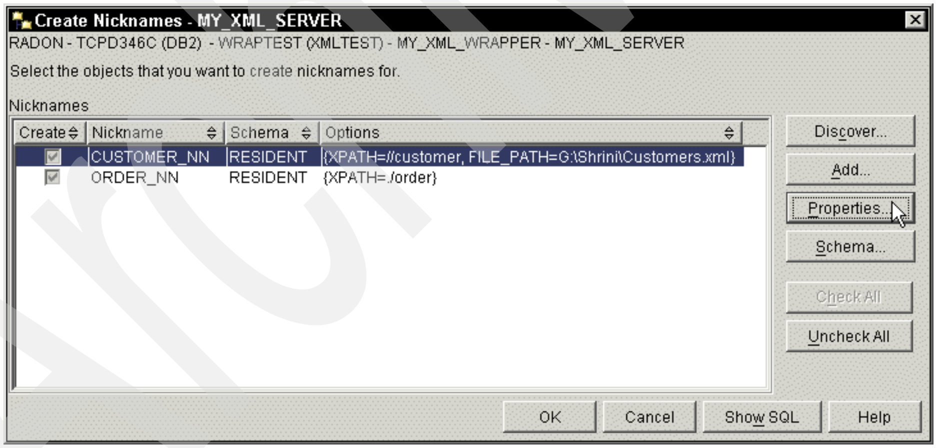
We could also have gone to this window by right-clicking the name of the XML server name **MY_XML_SERVER**, and selecting **Discover**.

In this window, we specify which type of XML data source we want to use, and provide the DB2 Control Center with the information it needs to determine how to set up the nicknames.

– We choose either an XML file, or an XML Schema file that describes your XML document. The tool then parses through the contents of this file and determines the possible XML-to-relational mappings. (Note that DTDs are currently not supported.)

- After parsing through the XML elements, the top level XML element name is displayed in the “ XML file top-level element ” field. If the tool finds more than one top-level XML element, the tool allows you to choose the element for which you want to create nickname.
- We can choose different XML wrapper options from the “`XML wrapper options` ” section of the window. We can choose one of the four values for DOCUMENT `type` option, namely DIRECTORY, FILE, COLUMN, URI. The meanings of these is similar to the ones we explained in “Options for specifying the XML data source for nicknames” on page 315. We can also choose if we want to specify the name of the FILE or DIRECTORY at query execution time by selecting the radio button named “**Provide data source at query time** ”.

8. After entering all the desired values for the window shown in Figure 11-8 on page 326, click **OK**. If you used the sample XML file shown in Example 11-1 on page 309 for discovering the mappings, you should see a window as shown in Figure 11-9.
9. Note that the window in Figure 11-9 also has a **Show SQL** button. It allows us to look at the actual SQL statements that will be executed when we click **OK**. We can also select all SQL statements (Ctl+a) and copy them to the clipboard (Ctl+c) for reuse, or manual execution later on.



Create Nicknames - MY_XML_SERVER

RADON - TCPD346C (DB2) - WRAPTEST (XMLTEST) - MY_XML_WRAPPER - MY_XML_SERVER

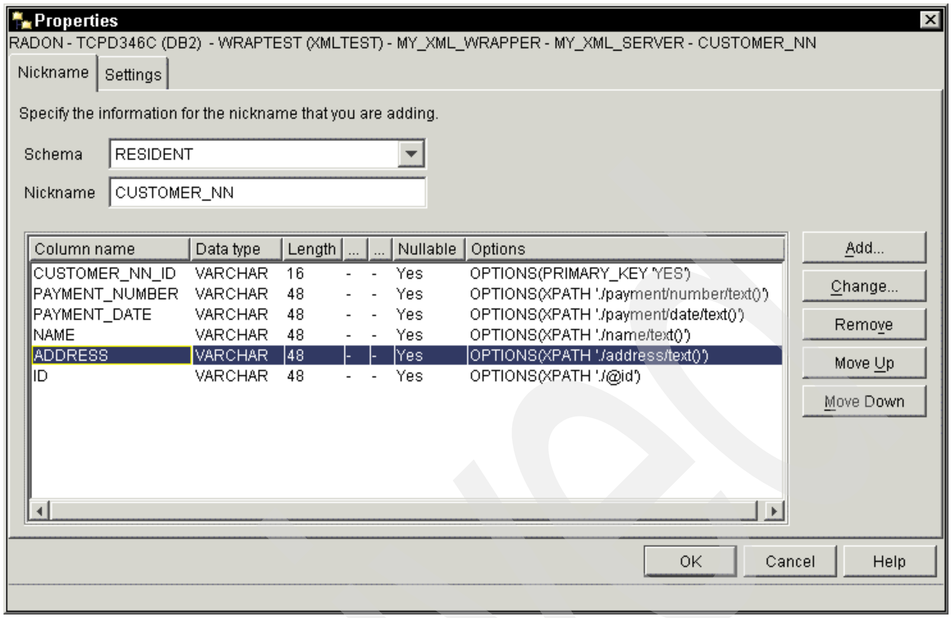
Select the objects that you want to create nicknames for.

Nicknames

Create	Nickname	Schema	Options
☑	CUSTOMER_NN	RESIDENT	{XPATH=//customer, FILE_PATH=G:\Shrini\Customers.xml}
☑	ORDER_NN	RESIDENT	{XPATH=./order}

Figure 11-9 XML-to-relational mappings discovered automatically

10. Now highlight the **CUSTOMER_NN** nickname and click the **Properties** button to find out the mapping details, such as which columns mapped to what XPATH. You see a window like the one shown in Figure 11-10 on page 328.



Properties

RADON - TCPD346C (DB2) - WRAPTEST (XMLTEST) - MY_XML_WRAPPER - MY_XML_SERVER - CUSTOMER_NN

Nickname | Settings

Specify the information for the nickname that you are adding.

Schema: RESIDENT

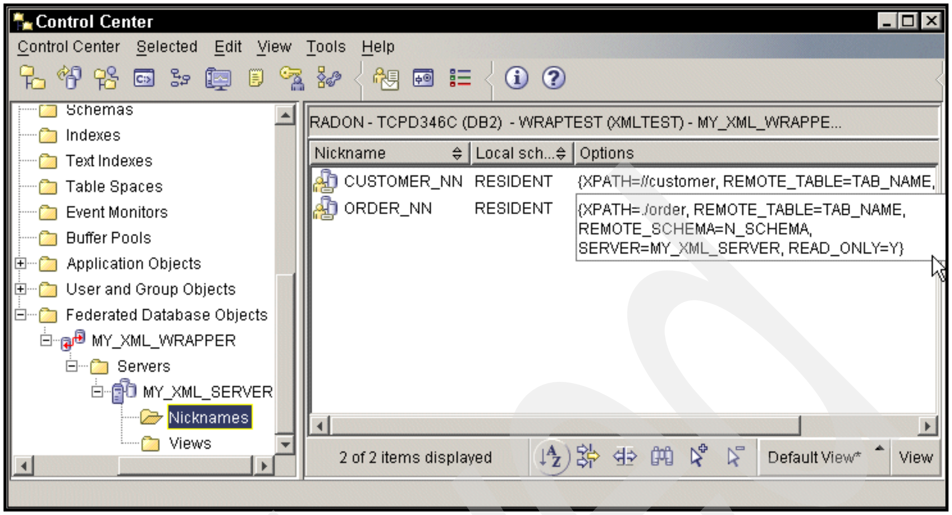
Nickname: CUSTOMER_NN

Column name	Data type	Length	...	...	Nullable	Options
CUSTOMER_NN_ID	VARCHAR	16	-	-	Yes	OPTIONS(PRIMARY_KEY 'YES')
PAYMENT_NUMBER	VARCHAR	48	-	-	Yes	OPTIONS(XPATH './payment/number/text()')
PAYMENT_DATE	VARCHAR	48	-	-	Yes	OPTIONS(XPATH './payment/date/text()')
NAME	VARCHAR	48	-	-	Yes	OPTIONS(XPATH './name/text()')
ADDRESS	VARCHAR	48	-	-	Yes	OPTIONS(XPATH './address/text()')
ID	VARCHAR	48	-	-	Yes	OPTIONS(XPATH './@id')

Figure 11-10 Nickname properties

11. We can change the data types of the mapped columns. When we use an XML Schema as input for the mapping, DB2 Control Center is able to determine the data types for the columns in the nicknames, based on the schema information. When we use an actual XML document as input, the tool maps all elements and attributes to VARCHAR, for lack of other information. Using this window, we have the possibility to change the length as well as the data type of the columns that will be used for the nicknames (by selecting a column and using the **Change...** button).

12. Click **OK** when done. This takes us back to the window shown in Figure 11-9 on page 327. Click **OK** again. We see the window as shown in Figure 11-11 on page 329. Now we have created two nicknames: CUSTOMER_NN and ORDERS_NN.



RADON - TCPD346C (DB2) - WRAPTEST (XMLTEST) - MY_XML_WRAPPE...

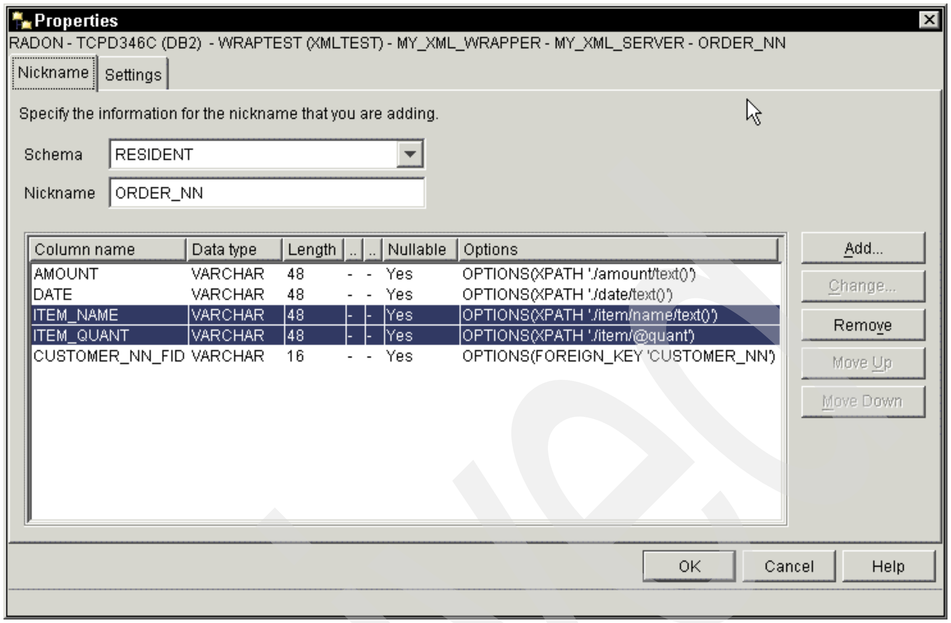
Nickname	Local sch...	Options
CUSTOMER_NN	RESIDENT	{XPATH=//customer, REMOTE_TABLE=TAB_NAME,
ORDER_NN	RESIDENT	{XPATH=./order, REMOTE_TABLE=TAB_NAME, REMOTE_SCHEMA=N_SCHEMA, SERVER=MY_XML_SERVER, READ_ONLY=Y}

2 of 2 items displayed

Figure 11-11 Nicknames Created

Notice that the tool created only two nicknames whereas we had created four nicknames (`customers`, `orders`, `items`, `payments`) from the same XML file in “Creating nicknames” on page 309. Did the tool miss the other mappings even for such a simple XML file? The answer is *no*. All the elements in the sample XML file have been mapped. However the tool managed to map all the elements in only two nicknames.

Let us look at the columns contained in the ORDER_NN nickname and find out what they are mapped to. Figure 11-12 on page 330 shows the properties of the ORDERS_NN nickname. The two highlighted lines defining the columns ITEM_NAME and ITEM_QUANT were actually part of our `ITEMS` nickname in “Creating nicknames” on page 309. Similarly, if you look at Figure 11-10 on page 328, lines 2 and 3 define the columns PAYMENT_NUMBER and PAYMENT_DATE, and account for our other missing nickname, namely PAYMENTS.



Properties

RADON - TCPD346C (DB2) - WRAPTEST (XMLTEST) - MY_XML_WRAPPER - MY_XML_SERVER - ORDER_NN

Specify the information for the nickname that you are adding.

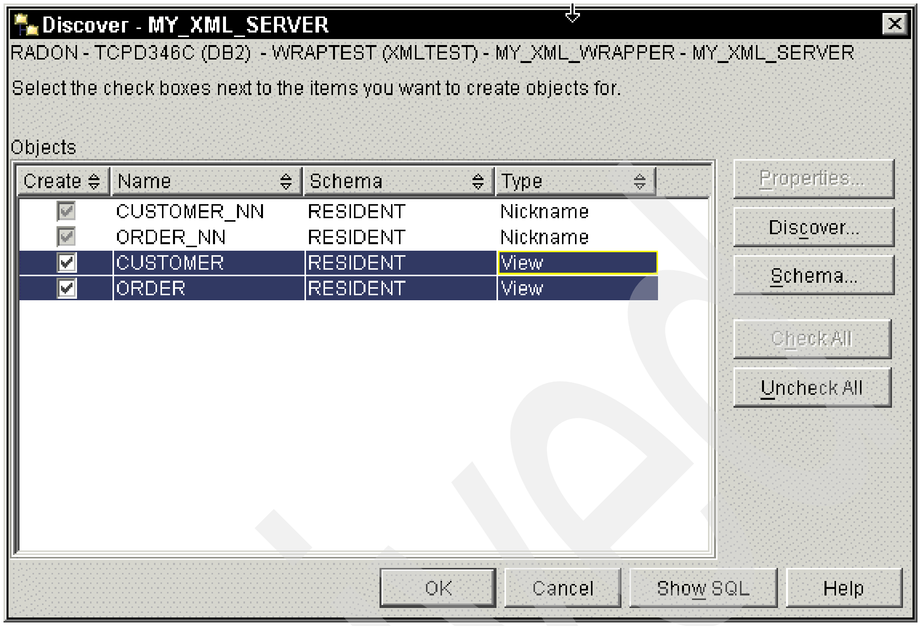
Schema: RESIDENT

Nickname: ORDER_NN

Column name	Data type	Length	..	..	Nullable	Options
AMOUNT	VARCHAR	48	-	-	Yes	OPTIONS(XPATH './amount/text()')
DATE	VARCHAR	48	-	-	Yes	OPTIONS(XPATH './date/text()')
ITEM_NAME	VARCHAR	48	-	-	Yes	OPTIONS(XPATH './item/name/text()')
ITEM_QUANT	VARCHAR	48	-	-	Yes	OPTIONS(XPATH './item/@quant')
CUSTOMER_NN_FID	VARCHAR	16	-	-	Yes	OPTIONS(FOREIGN_KEY 'CUSTOMER_NN')

Figure 11-12 Properties of ORDERS_NN nickname

In “Creating federated views for non-root nicknames” on page 314, we discussed the need for creating federated views. In fact we can create all the possible nicknames and views in one shot by right-clicking your XML server name and then selecting the **Discover** option. This results in identifying the nicknames and views as shown in Figure 11-13 on page 331.



Discover - MY_XML_SERVER

RADON - TCPD346C (DB2) - WRAPTEST (XMLTEST) - MY_XML_WRAPPER - MY_XML_SERVER

Select the check boxes next to the items you want to create objects for.

Objects

Create	Name	Schema	Type
☑	CUSTOMER_NN	RESIDENT	Nickname
☑	ORDER_NN	RESIDENT	Nickname
☑	CUSTOMER	RESIDENT	View
☑	ORDER	RESIDENT	View

Figure 11-13 Discovering possible nicknames and views combined

11.4 Best practices

Consider the following items when deciding whether or not to use the XML wrapper :

- The complexity of the XML document
- The size of the XML document (limited by the available virtual memory)
- XML files must be accessible to the DB2 Information Integrator server (same machine or on a network accessible shares/mounts).
- INSERT, UPDATE, DELETE are not supported when using the XML wrapper
- Whether namespace support is required. Currently XML wrapper does not support namespaces. Namespace support may be made available in a future release.

Since DB2 is a cost engine, some specific parameters were added for the XML wrapper:

- **INSTANCE_PARSE_TIME** (for root-nicknames only): Time required in milliseconds to parse the XML document
- **XPATH_EVAL_TIME**: Time required in milliseconds to evaluate a nickname XPath expression

You can use the default values or modify them at CREATE NICKNAME time to optimize your queries.

You can also set the DB2_DJ_COMM profile variable to load the wrapper during the database startup:

```
db2set DB2_DJ_COMM='libdb2lsxml.a' (AIX)
db2set DB2_DJ_COMM='libdb2lsxml.sl' (HP-UX)
db2set DB2_DJ_COMM='libdb2lsxml.so' (Solaris, and Linux)
db2set DB2_DJ_COMM='db2lsxml.dll' (Windows)
```

Handling large XML documents

When an XML document's size or complexity exceeds a threshold, the memory requirement to process a query can exceed the virtual storage available. When this occurs, the query will terminate and the XML wrapper returns:

```
SQL0901N The SQL statement failed because of a non-severe system
error.Subsequent SQL statements can be processed.(Reason "Unspecified
exception while parsing input document".)SQLSTATE=58004
```

To avoid this problem, you can use the STREAMING nickname option for the XML wrapper (available starting with DB2 V8.1 FixPak 3). The STREAMING option (see Example 11-19) specifies whether the XML source document is separated into logical fragments that correspond to the node that matches the XPath expression of the nickname. The XML wrapper then parses and processes the XML source data fragment by fragment, reducing the total memory required to read the document. You can specify streaming for any XML source document (FILE, DIRECTORY, URI, or COLUMN). This option is accepted only for columns of the root nickname (the nickname that identifies the elements at the top level of the XML document). The default streaming value is NO.

Note that you should not set the STREAMING parameter to YES, if you also set the VALIDATE parameter to YES. If you set both parameters to YES, you will receive an error message. Example 11-19 shows a create nickname statement using the STREAMING option.

Example 11-19 XML - Create nickname statement

```
CREATE NICKNAME customers
(
  id VARCHAR(5) OPTIONS (XPATH '@id'),
  cname VARCHAR(50) OPTIONS (XPATH './name/text()'),
  cid VARCHAR(20) OPTIONS (PRIMARY_KEY 'YES')
)
FOR SERVER xml_server
  OPTIONS ( XPATH '/customer'
           ,FILE_PATH 'C:\Shrini\Customers.xml'
           ,STREAMING 'YES')
```

Part 3

XML tools for database systems

12

XML and database tools in Application Developer

In this chapter, we introduce the various XML and database tools that are provided by Application Developer.

We walk through some simple examples based on the INSURA database that is used in the scenario.

Note: To run the examples, you must have created the INSURA database as described in “Setting up the INSURA database” on page 402. You do not have to go through the whole scenario.

12.1 Application Developer tools overview

In this section, we list the tools provided by Application Developer for XML and database activities.

12.1.1 Perspectives

Application Developer includes three perspectives that are most often used when authoring, generating or transforming XML files:

- **Data**—The Data perspective contains various tools to import and export data as XML from all databases having a JDBC driver.
- **XML**—The XML perspective contains various tools author, edit and transform XML related files (XML, XML Schema, DTD, XSL stylesheet, and so forth).
- **XSL Debug**—The XSL Debug perspective enables you to debug XSL transformations in a symbolic debugger.

12.1.2 Authoring and generation tools

Application Developer provides a comprehensive visual XML development environment. The tool set includes components for building DTDs, XML Schemas, XML files , and XSL files:

- **XML editor**—The XML editor is a tool for creating and viewing XML files. You can use it to create new XML files, either from scratch, existing DTDs, or existing XML Schemas. You can also use it to edit XML files, associate them with DTDs or schemas, and validate them.
- **DTD editor**—The DTD editor is a tool for creating and viewing DTDs. Using the DTD editor, you can create DTDs, generate XML Schema files, and generate Java beans. You can also use the DTD editor to generate a default HTML form based on the DTDs you create.
- **XML Schema editor**—The XML Schema editor is a tool for creating, viewing, and validating XML Schemas. You can use the XML Schema editor to perform tasks such as creating XML Schema components, importing and viewing XML Schemas, generating DTDs and relational table definitions from XML Schemas, and generating Java beans for creating XML instances of an XML Schema.
- **XSL editor**—The XSL editor can be used to create new XSL files or to edit existing ones. You can use content assist and various wizards to help you create or edit the XSL file. Once you have finished editing your file, you can also validate it. You can also associate an XML instance file with the XSL source file you are editing and use that to provide guided editing when defining constructions such as an XPath expression.

- **XPath Expression wizard**—You can use the XPath expression wizard to create XPath expressions. XPath expressions can be used to search through XML documents, extracting information from the nodes (such as an element or attribute).
- **XSL tools**—You can use the XSL debugging and transformation tool to apply XSL files to XML files, transforming them into new XML, HTML, or text files. After the transformation has taken place, the XSL Debug perspective enables you to visually step through an XSL transformation script, highlighting the transformation rules as they are fired. You can use the views in the XSL Debug perspective to help you debug the XML or XSL files.
- **SQL Query wizard and SQL Builder**—You can use either the SQL Query wizard or SQL Builder to create SQL statements for XML generation or for database applications.
- **User-defined function**—You can build and deploy SQL user-defined functions that can be used in SQL statements.
- **Stored procedures**—Application Developer includes the stored procedure builder in a fully integrated way. Java and SQL stored procedures can be built and deployed.
- **XML and SQL Query wizard**—You can use the XML and SQL Query wizard to create an XML file from the results of an SQL query or take an XML file and store it in a relational table. When creating an XML file from an SQL query, you can optionally choose to create an XML Schema or DTD file that describes the structure that the XML file has for use in other applications. Two Java class libraries `SQLToXML` (sqltoxml.jar) and `XMLToSQL` (xmltosql.jar) are included so you can use them in your applications at runtime.   These JAR files are located in:

  ```
  <wsadhome>\wstools\eclipse\plugins\com.ibm.etools.sqltoxml\jars
  ```
- **XML to XML mapping editor**—The XML to XML mapping editor is a tool used to map one or more source XML files to a single target XML file. You can add XPath expressions, groupings, Java methods or conversion functions to your mapping. Mappings can also be edited, deleted, or persisted for later use. After defining the mappings, you can generate an XSLT script. The generated script can then be used to combine and transform any XML files that conform to the source DTDs.
- **RDB to XML mapping editor**—The RDB to XML mapping editor is a tool for defining the mapping between one or more relational tables and an XML file. After you have created the mapping, you can generate a document access definition (DAD) script which can be run by the DB2 XML Extender to either compose XML files from existing DB2 data, or decompose XML files into DB2 data.

- **Web Service wizard**—Application Developer can create Web services from SQL statements and stored procedures. In a first step, a document access definition extended (DADX) file is created from the SQL statement or stored procedure; in a second step a Web service is created from the DADX file.

12.1.3 Preparation

To prepare for the tools walkthrough, set up a workspace, for example:

```
c:\WSAD51sg246994
```

Start Application Developer using the new workspace by typing the workspace directory name at the prompt, or by setting up a startup icon that points to the workspace:

```
<wsadhome>\wsappdev.exe -data C:\WSAD51sg246994
```

Web project

Create a Web project named `ItsoInsuraTest`:

1. Select **File -> New -> Other -> Web -> Dynamic Web Project.**
2. Enter **`ItsoInsuraTest`** as name and click **Finish**.
3. The Web project is added to a `DefaultEAR` enterprise application.

12.2 XML tools walkthrough

In this section, we discuss the various XML tools using simple examples based on the INSURA database.

You must have run the scenario to define the database and insert some test data into the tables.

12.2.1 Creating a database connection

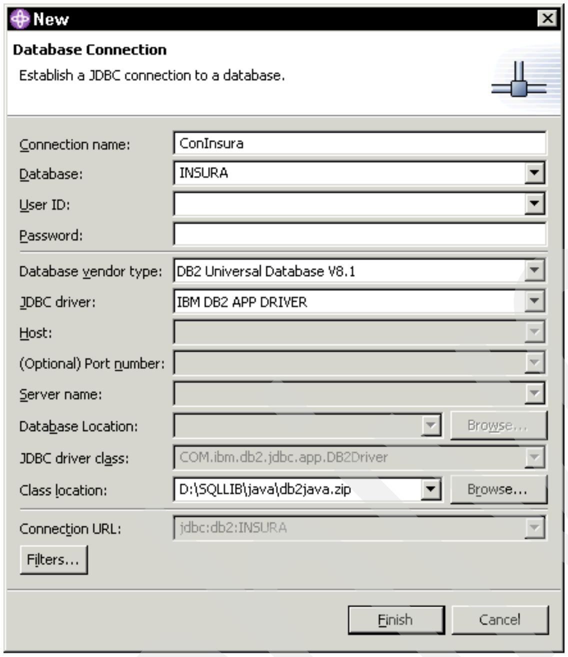
Many of the tools that will be described later depend on you having already established a connection to your database and then importing the database schema into Application Developer.

Connection dialog

To start, you must create a connection to the database containing the schema and data in question. You do this in the Data perspective. Bring up the context menu in the DB Servers view and select **New Connection**. The Database

Connection dialog is displayed (Figure 12-1). Fill in the database connection information and click **Finish**.

You could use the Filter dialog to limit the set of tables that are retrieved.



New

Database Connection

Establish a JDBC connection to a database.

Connection name: ConInsura

Database: INSURA

User ID:

Password:

Database vendor type: DB2 Universal Database V8.1

JDBC driver: IBM DB2 APP DRIVER

Host:

(Optional) Port number:

Server name:

Database Location:

JDBC driver class: COM.ibm.db2.jdbc.app.DB2Driver

Class location: D:\SQLLIB\java\db2java.zip

Connection URL: jdbc:db2:INSURA

Figure 12-1 Database connection

After that, you should see that the DB Servers view has imported the database schema from the database server (Figure 12-2 on page 342).

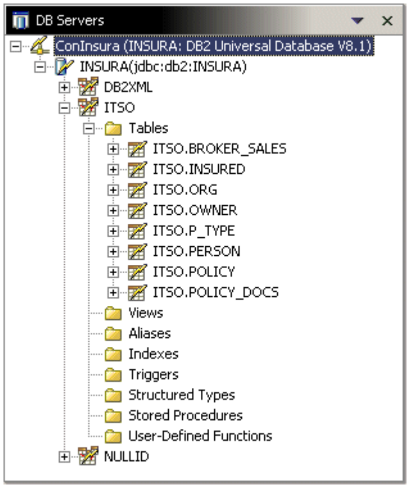


Figure 12-2 DB Servers view after connecting to the database

Importing database objects

To work with database objects in Application Developer, you have to import the objects into a project for further processing.

Often, you will import the database schema into a Simple project or a Web project, but this really depends on how you intend to use the database schema in your application. For example, if you only want to transform some data into an XML file, use a Simple project. If, on the other hand, you intend to build a Web application that extracts the data and then transform it into an HTML page to be output to a Web browser, import it into a Web project.

To import the database, schema, and tables into a project, select the database or one of the schema names (**`ITSO`**) in the DB Servers view and select **Import to Folder** (context). Next, select the project to import the database schema into and click **Finish**. Notice that depending on the project type (Simple, Web), the default final destination folder will be different.

Figure 12-3 on page 343 shows the imported files in the `ItsoInsuraTest` Web project after importing the `ITSO` tables. Notice the files that were imported into the WEB-INF subdirectory databases.

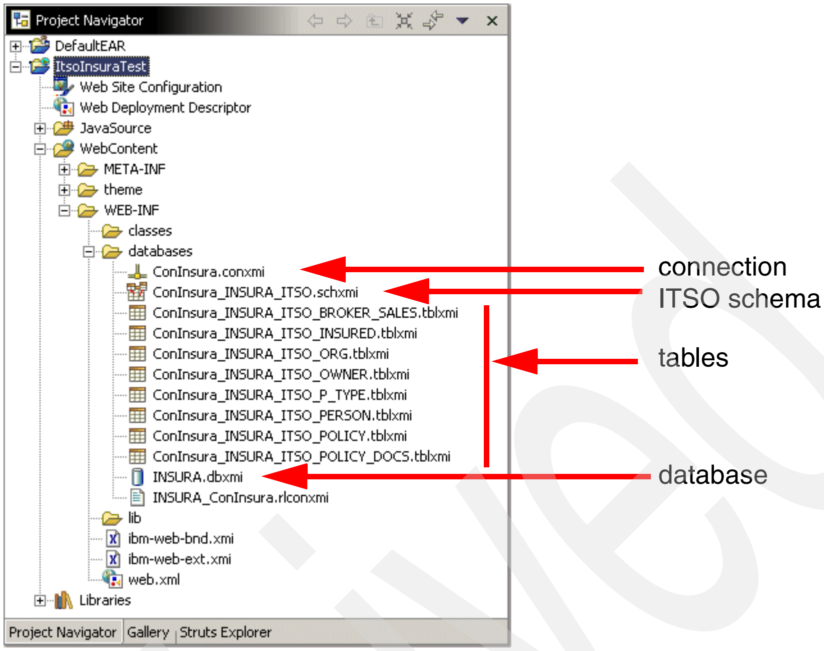

Project Navigator
DefaultEAR
ItsoInsuraTest
Web Site Configuration
Web Deployment Descriptor
JavaSource
WebContent
META-INF
theme
WEB-INF
classes
databases
ConInsura.conxmi
ConInsura_INSURA_ITSO.schxmi
ConInsura_INSURA_ITSO_BROKER_SALES.tblxmi
ConInsura_INSURA_ITSO_INSURED.tblxmi
ConInsura_INSURA_ITSO_ORG.tblxmi
ConInsura_INSURA_ITSO_OWNER.tblxmi
ConInsura_INSURA_ITSO_P_TYPE.tblxmi
ConInsura_INSURA_ITSO_PERSON.tblxmi
ConInsura_INSURA_ITSO_POLICY.tblxmi
ConInsura_INSURA_ITSO_POLICY_DOCS.tblxmi
INSURA.dbxmi
INSURA_ConInsura.rlconxmi
lib
ibm-web-bnd.xmi
ibm-web-ext.xmi
web.xml
Libraries
Project Navigator
Gallery
Struts Explorer
connection
ITSO schema
tables
database

Figure 12-3 Imported files in a Web project

Data Definition view

The Data Definition view shows the imported objects in a hierarchical list (Figure 12-4).

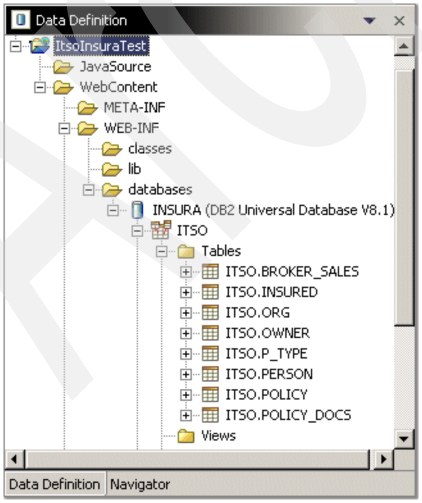
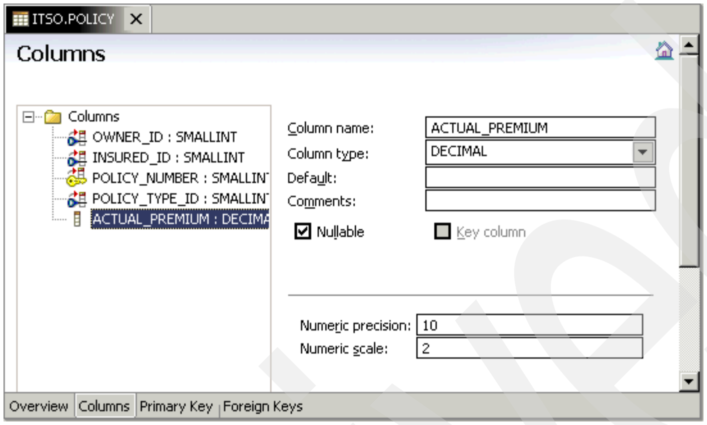


Figure 12-4 Data Definition view after import

12.2.2 Database editors

You can edit any of the imported definitions by opening (double-clicking) the files. Using these editors, you can also create new tables, schema, databases, and views. Figure 12-5 shows the table editor for the `POLICY` table.



Columns
- OWNER_ID : SMALLINT
- INSURED_ID : SMALLINT
- POLICY_NUMBER : SMALLIN
- POLICY_TYPE_ID : SMALLIN
- ACTUAL_PREMIUM : DECIMA

Column name: ACTUAL_PREMIUM
Column type: DECIMAL
Default:
Comments:
☑ Nullable □ Key column

Numeric precision: 10
Numeric scale: 2

Figure 12-5 Table editor

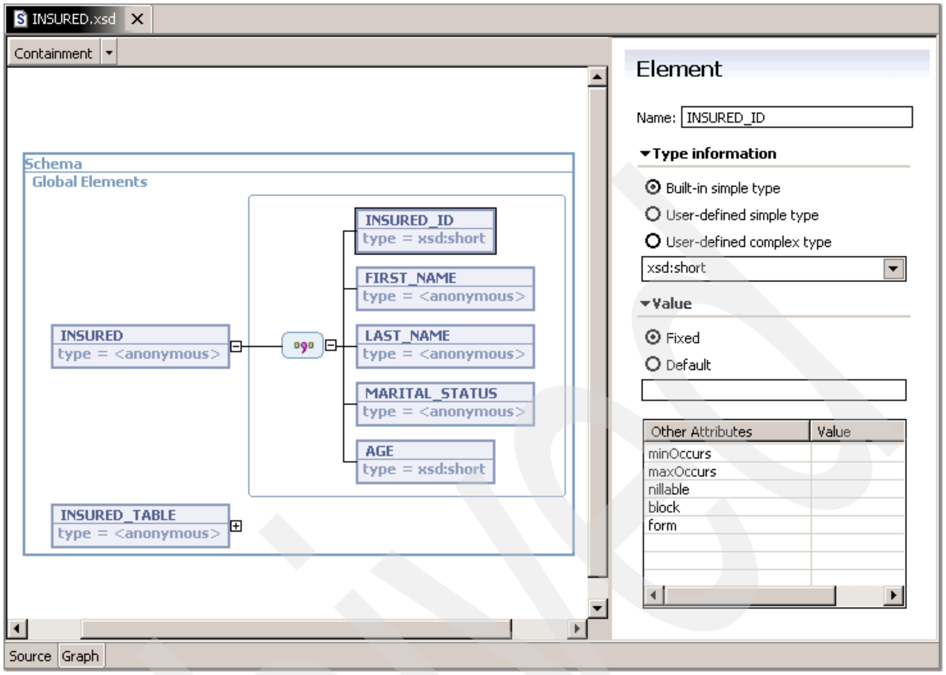
12.2.3 XML Schema, table DDL, and DDT

Application Developer provides generation utilities from table definitions to XML Schema and DDL.

You can generate an XML Schema (XSD) file from a relational table. You can then can further customize that file in the XML Schema editor.

To generate a XML Schema file from a relational table, follow these steps:

1. In the Data Definition view, select the table (**ITSO.INSURED**) and **Generate XML Schema** (context).
2. Select a project or folder to contain the XML file and type a name for it. The name of the file must end in `.xsd`. For now leave the defaults and click **Finish**.
3. The INSURED.xsd file is generated and opened in the XML Schema editor. You can either view the source or the graph of the XML Schema (Figure 12-6 on page 345).



Name: INSURED_ID

● Built-in simple type
○ User-defined simple type
○ User-defined complex type

xsd:short

● Fixed
○ Default

Figure 12-6 XML Schema editor: graph view expanded

To generate the DDL file for a table, select the table (**ITSO.INSURED**) in the Data Definition view and **Generate DDL** (context). Leave the default filename of `INSURED.sql` for the generated output. The content of the file is:

```
CREATE TABLE ITSO.INSURED
  (INSURED_ID SMALLINT NOT NULL,
   FIRST_NAME CHARACTER(20) NOT NULL,
   LAST_NAME CHARACTER(20) NOT NULL,
   MARITAL_STATUS CHARACTER(2) NOT NULL,
   AGE SMALLINT NOT NULL);

ALTER TABLE ITSO.INSURED
  ADD CONSTRAINT INSURED_PK PRIMARY KEY (INSURED_ID);
```

You can generate a relational table definition (DDL) from an XML Schema (XSD) file:

1. Select the **INSURED.xsd** file in the Navigator view and click **Generate -> DDL**.
2. Select the project or folder that will contain the relational table. Enter an output name of `INSURED1.sql`. Click **Finish**.

3. Open the generated file and compare it to the previously generated DDL file. One difference is that the file generated from the XML Schema does not show all the `NOT NULL` clauses.

From the XML Schema, you can generate a DTD, an XPath expression, or a sample XML file. For example, the generated DTD would look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT INSURED_TABLE (INSURED*)>
<!ELEMENT INSURED (INSURED_ID,FIRST_NAME,LAST_NAME,MARITAL_STATUS,AGE)>
<!ELEMENT INSURED_ID (#PCDATA)>
<!ELEMENT AGE (#PCDATA)>
<!ELEMENT FIRST_NAME (#PCDATA)>
<!ELEMENT LAST_NAME (#PCDATA)>
<!ELEMENT MARITAL_STATUS (#PCDATA)>
```

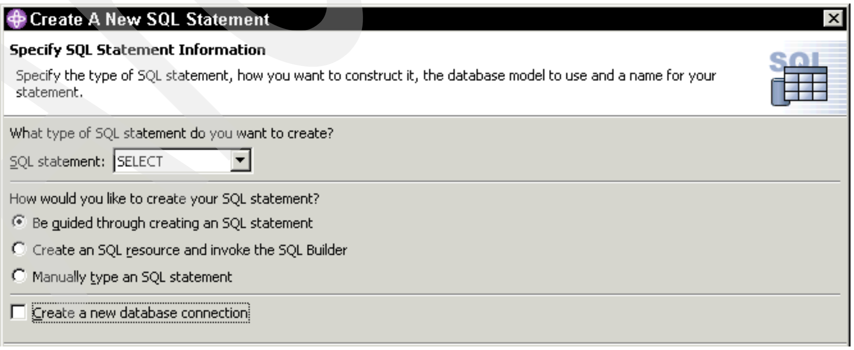
12.2.4 Creating an SQL statement

This wizard is used to write a SQL statement based on a database schema that has been imported into a WSAD project.

For example, we can create an SQL statement that lists insured people, their policy number and policy plan.

You can start the wizard by selecting **File -> New -> Other -> Data** and **SQL Statement** and clicking **Next**.

In the first wizard panel (Figure 12-7), you select either **Be guided through creating an SQL statement** or **Create an SQL resource and invoke the SQL Builder**. We will show the SQL Builder later. Deselect **Create a new database connection** (we use the existing imported database).


Create A New SQL Statement

Specify SQL Statement Information

Specify the type of SQL statement, how you want to construct it, the database model to use and a name for your statement.

What type of SQL statement do you want to create?

SQL statement: SELECT

How would you like to create your SQL statement?

● Be guided through creating an SQL statement

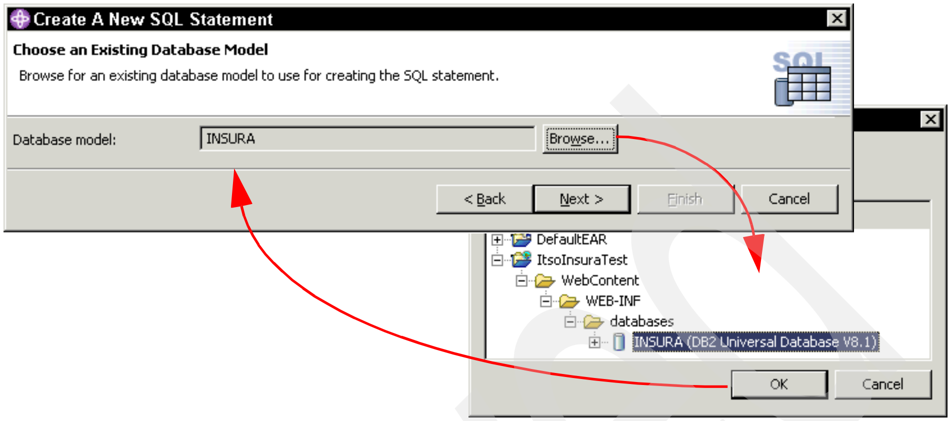
○ Create an SQL resource and invoke the SQL Builder

○ Manually type an SQL statement

□ Create a new database connection

Figure 12-7 Creating an SQL statement: guided or builder

In the second panel (Figure 12-8), click **Browse** to locate the imported database.



Create A New SQL Statement

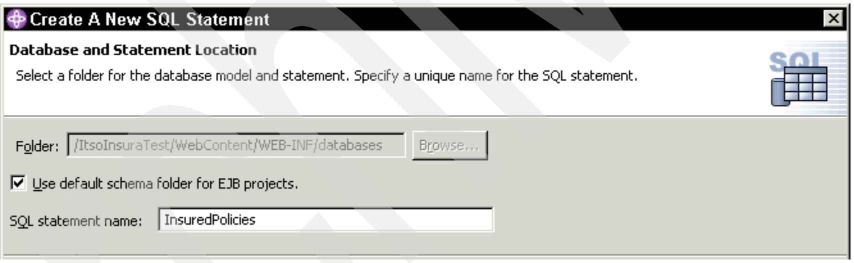
Choose an Existing Database Model

Browse for an existing database model to use for creating the SQL statement.

Database model: INSURA

Figure 12-8 Creating an SQL statement: database selection

In the third panel (Figure 12-9), enter the name of the SQL statement as `InsuredPolicies` and click **Next**.



Create A New SQL Statement

Database and Statement Location

Select a folder for the database model and statement. Specify a unique name for the SQL statement.

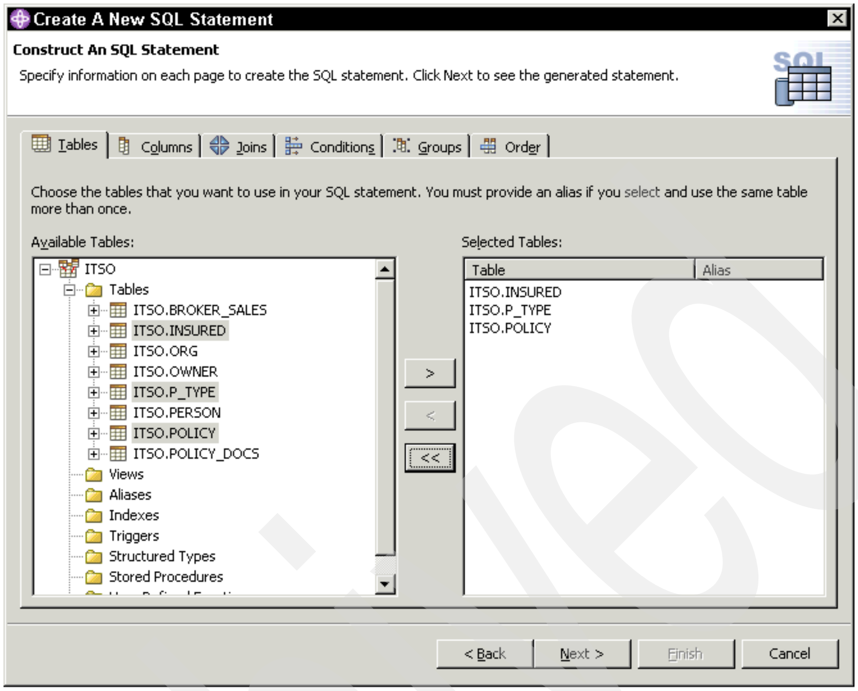
Folder: /ItsoInsuraTest/WebContent/WEB-INF/databases

☑ Use default schema folder for EJB projects.

SQL statement name: InsuredPolicies

Figure 12-9 Creating an SQL statement: name

Now we construct the actual SQL statement. First, select the three tables **INSURED**, **P_TYPE**, and **POLICY** and click **>** (Figure 12-10 on page 348).



Construct An SQL Statement

Specify information on each page to create the SQL statement. Click Next to see the generated statement.

Choose the tables that you want to use in your SQL statement. You must provide an alias if you select and use the same table more than once.

Available Tables:

```
ITSO
  Tables
    ITSO.BROKER_SALES
    ITSO.INSURED
    ITSO.ORG
    ITSO.OWNER
    ITSO.P_TYPE
    ITSO.PERSON
    ITSO.POLICY
    ITSO.POLICY_DOCS
  Views
  Aliases
  Indexes
  Triggers
  Structured Types
  Stored Procedures
```

Selected Tables:

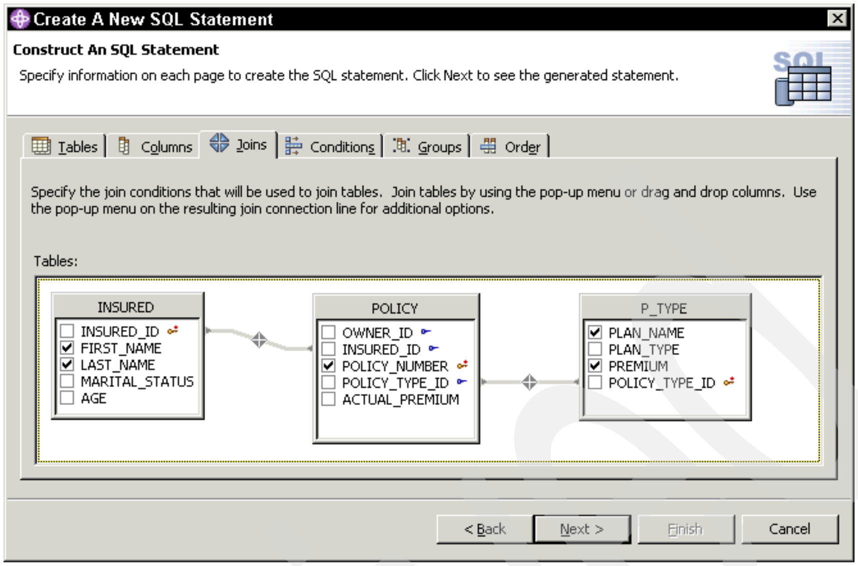
Table	Alias
ITSO.INSURED	
ITSO.P_TYPE	
ITSO.POLICY	

Figure 12-10 Creating an SQL statement: table selection

On the Columns page, expand the tables, select **FIRST_NAME** and **LAST_NAME** from `INSURED`, **POLICY_NUMBER** from `POLICY`, and **PLAN_NAME** and **PREMIUM** from `P_TYPE`. Click **>** for each selection to get the correct sequence.

On the Joins page, rearrange the tables, then join `INSURED` and `POLICY` on `INSURED_ID`, and `POLICY` and `P_TYPE` on `POLICY_TYPE_ID` (Figure 12-11 on page 349). To join two tables, drag a join column from one table to the matching column in the other table.

Note that you can select the columns also on the Join page.



Create A New SQL Statement

Construct An SQL Statement

Specify information on each page to create the SQL statement. Click Next to see the generated statement.

Specify the join conditions that will be used to join tables. Join tables by using the pop-up menu or drag and drop columns. Use the pop-up menu on the resulting join connection line for additional options.

Tables:

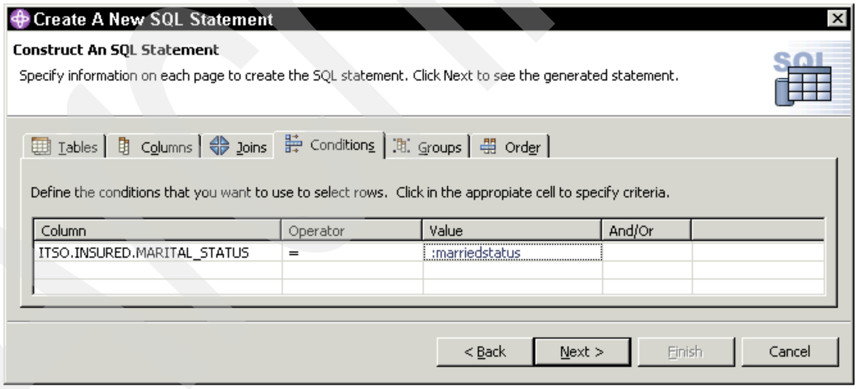
INSURED
- □ INSURED_ID
- ☑ FIRST_NAME
- ☑ LAST_NAME
- □ MARITAL_STATUS
- □ AGE

POLICY
- □ OWNER_ID
- □ INSURED_ID
- ☑ POLICY_NUMBER
- □ POLICY_TYPE_ID
- □ ACTUAL_PREMIUM

P_TYPE
- ☑ PLAN_NAME
- □ PLAN_TYPE
- ☑ PREMIUM
- □ POLICY_TYPE_ID

Figure 12-11   Creating an SQL statement: join

On the Conditions page (Figure 12-12), add a condition to select by `MARRIED_STATUS`. Select the column and the operator using the pull-down menus, and type in a variable name (`:marriedstatus`).



Create A New SQL Statement

Construct An SQL Statement

Specify information on each page to create the SQL statement. Click Next to see the generated statement.

Define the conditions that you want to use to select rows. Click in the appropiate cell to specify criteria.

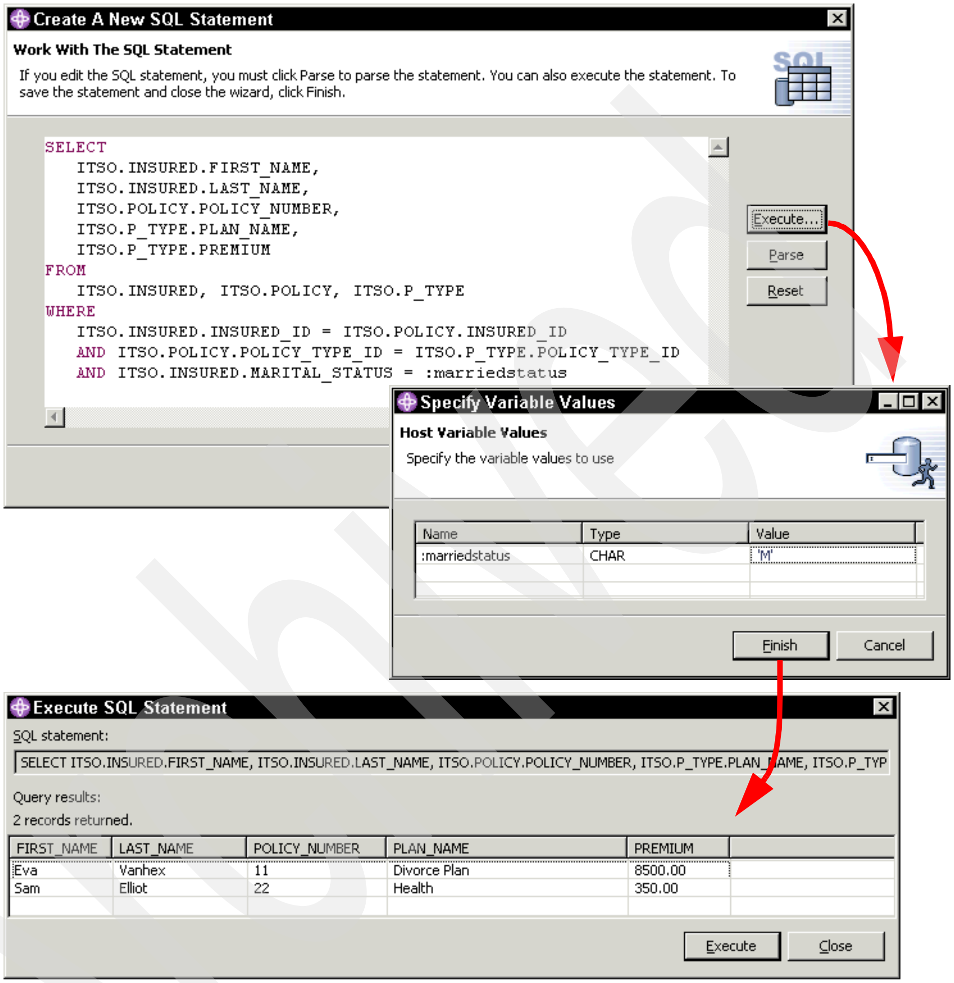
Column	Operator	Value	And/Or	
ITSO.INSURED.MARITAL_STATUS	=	:marriedstatus		

Figure 12-12   Creating an SQL statement: condition

Click **Next** and the complete SQL statement is displayed.

Click **Execute** to test the statement. You are prompted for the variable. Enter `'M'` (with quotes) for the `:marriedstatus` and click **Finish**. The results are displayed (Figure 12-13 on page 350).

Select the table(s) that your SQL statement will use and then select the **Next** button.



Create A New SQL Statement

Work With The SQL Statement

If you edit the SQL statement, you must click Parse to parse the statement. You can also execute the statement. To save the statement and close the wizard, click Finish.

```
SELECT
   ITSO.INSURED.FIRST_NAME,
   ITSO.INSURED.LAST_NAME,
   ITSO.POLICY.POLICY_NUMBER,
   ITSO.P_TYPE.PLAN_NAME,
   ITSO.P_TYPE.PREMIUM
FROM
   ITSO.INSURED, ITSO.POLICY, ITSO.P_TYPE
WHERE
   ITSO.INSURED.INSURED_ID = ITSO.POLICY.INSURED_ID
   AND ITSO.POLICY.POLICY_TYPE_ID = ITSO.P_TYPE.POLICY_TYPE_ID
   AND ITSO.INSURED.MARITAL_STATUS = :marriedstatus
```

Specify Variable Values

Host Variable Values

Specify the variable values to use

Name	Type	Value
:marriedstatus	CHAR	'M'

Execute SQL Statement

SQL statement:

```
SELECT ITSO.INSURED.FIRST_NAME, ITSO.INSURED.LAST_NAME, ITSO.POLICY.POLICY_NUMBER, ITSO.P_TYPE.PLAN_NAME, ITSO.P_TYP
```

Query results:

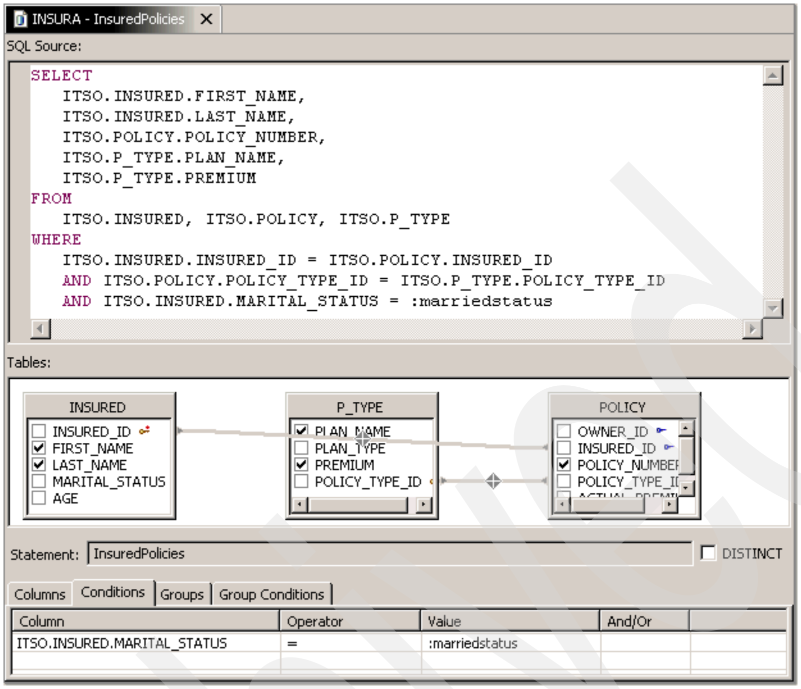
2 records returned.

FIRST_NAME	LAST_NAME	POLICY_NUMBER	PLAN_NAME	PREMIUM
Eva	Vanhex	11	Divorce Plan	8500.00
Sam	Elliot	22	Health	350.00

Figure 12-13 Creating an SQL statement: execute

Click **Finish** to save the SQL statement. The SQL Builder opens (Figure 12-14 on page 351).

You can also construct an SQL statement in the SQL Builder by dragging tables to the top or middle pane, selecting columns, making joins, and entering conditions.



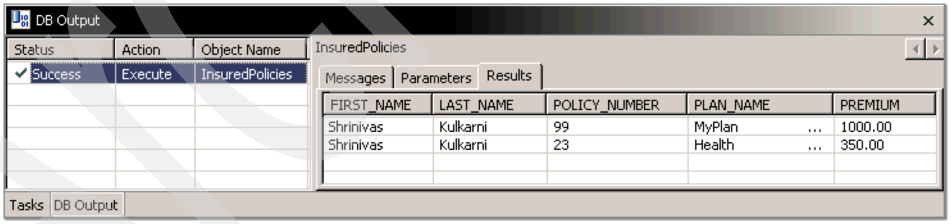
```
SELECT
   ITSO.INSURED.FIRST_NAME,
   ITSO.INSURED.LAST_NAME,
   ITSO.POLICY.POLICY_NUMBER,
   ITSO.P_TYPE.PLAN_NAME,
   ITSO.P_TYPE.PREMIUM
FROM
   ITSO.INSURED, ITSO.POLICY, ITSO.P_TYPE
WHERE
   ITSO.INSURED.INSURED_ID = ITSO.POLICY.INSURED_ID
   AND ITSO.POLICY.POLICY_TYPE_ID = ITSO.P_TYPE.POLICY_TYPE_ID
   AND ITSO.INSURED.MARITAL_STATUS = :marriedstatus
```

Statement: InsuredPolicies

Column	Operator	Value	And/Or
ITSO.INSURED.MARITAL_STATUS	=	:marriedstatus	

Figure 12-14 SQL Builder

Select **SQL -> Execute** to run the statement again. Enter `'S'` as a parameter. The output is displayed in the DB Output view (Figure 12-15).



Status	Action	Object Name
✓ Success	Execute	InsuredPolicies

FIRST_NAME	LAST_NAME	POLICY_NUMBER	PLAN_NAME	PREMIUM
Shrinivas	Kulkarni	99	MyPlan ...	1000.00
Shrinivas	Kulkarni	23	Health ...	350.00

Figure 12-15 DB Output view with SQL execution

We also create a second SQL statement named `AllInsured` that lists the `INSURED` table as:

```
SELECT ITSO.INSURED.INSURED_ID, ITSO.INSURED.FIRST_NAME,
   ITSO.INSURED.LAST_NAME, ITSO.INSURED.MARITAL_STATUS, ITSO.INSURED.AGE
FROM ITSO.INSURED
ORDER BY INSURED_ID ASC
```

12.2.5 Creating a user-defined function (UDF)

Application Developer supports the development of user-defined functions (UDFs).

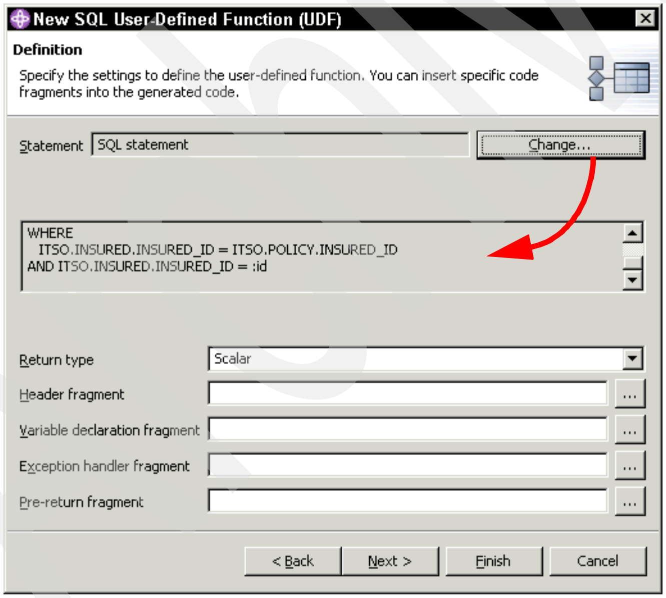
For example, let's create a UDF that returns the total premium for an insured.

To start the UDF wizard, either select **File -> New -> Other -> Data -> User-Defined Function** or expand the INSURA database and the ITSO schema in the Data Definition view, then select **User-Defined Functions** and **New -> SQL User-defined Function** (context).

If you use the first option, click **Browse** to locate the `ITSO` schema in the INSURA database. If you use the second option, this selection is bypassed.

Then, specify the name of the UDF, for example, `InsuredPremium`.

Next, you specify the UDF definition. Click **Change** to specify the SQL statement. You can use the SQL Assist wizard or just type the statement (Figure 12-16).



New SQL User-Defined Function (UDF)

Definition

Specify the settings to define the user-defined function. You can insert specific code fragments into the generated code.

Statement: SQL statement

```
WHERE
  ITSO.INSURED.INSURED_ID = ITSO.POLICY.INSURED_ID
AND ITSO.INSURED.INSURED_ID = :id
```

Return type: Scalar

Header fragment:

Variable declaration fragment:

Exception handler fragment:

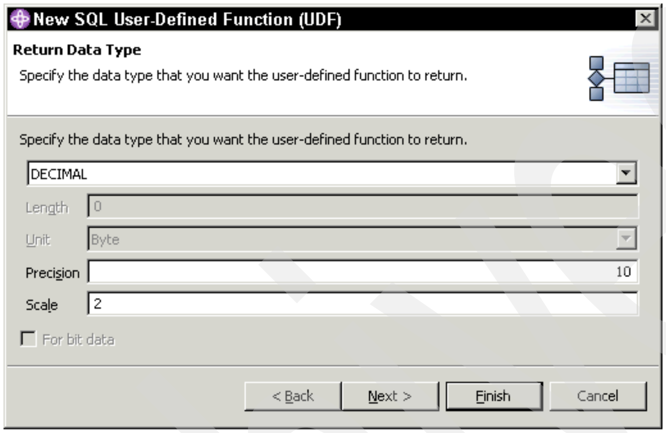
Pre-return fragment:

Figure 12-16 Creating a user-defined function: SQL statement

Enter this SQL statement:

```
SELECT SUM(ITSO.POLICY.ACTUAL_PREMIUM)
  FROM ITSO.INSURED, ITSO.POLICY
 WHERE ITSO.INSURED.INSURED_ID = ITSO.POLICY.INSURED_ID
   AND ITSO.INSURED.INSURED_ID = :id
```

The return data type from a UDF is either a scalar or a table. In the next step (Figure 12-17), set the return type to DECIMAL(10,2).


New SQL User-Defined Function (UDF)

Return Data Type

Specify the data type that you want the user-defined function to return.

Specify the data type that you want the user-defined function to return.

DECIMAL

Length: 0

Unit: Byte

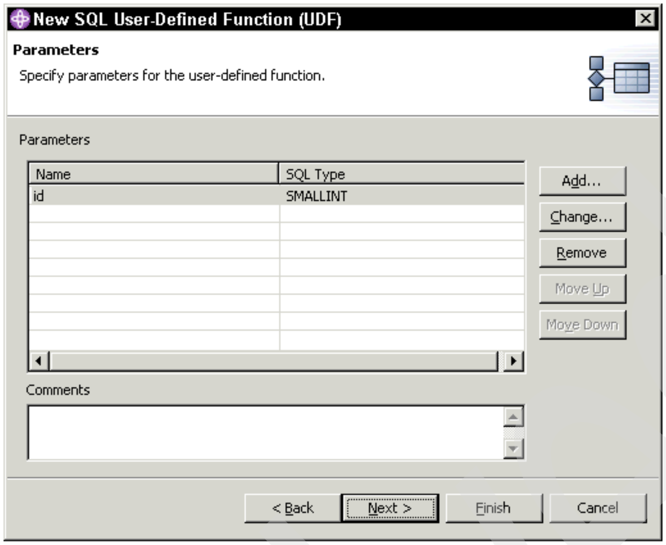
Precision: 10

Scale: 2

□ For bit data

Figure 12-17   Creating a user-defined function: return type

Next, you can define all of the parameters (and their SQL data types) for the UDF. We require one parameter, id, a SMALLINT (Figure 12-18 on page 354).



New SQL User-Defined Function (UDF)

Parameters

Specify parameters for the user-defined function.

Parameters

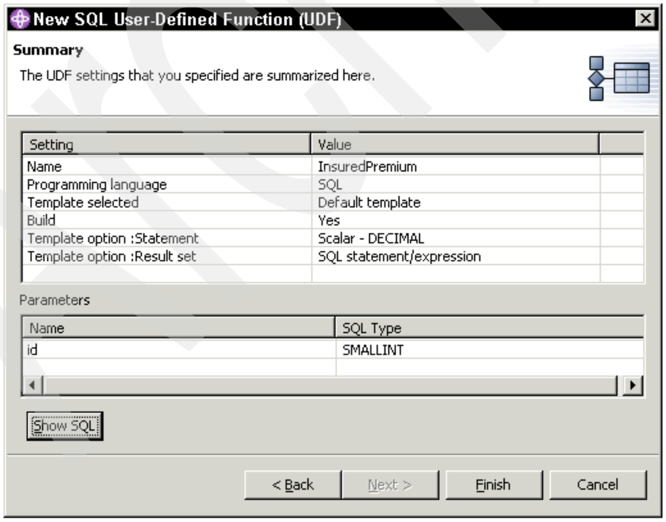
Name	SQL Type
id	SMALLINT

Comments

Figure 12-18 Creating a user-defined function: parameters

Next, you can specify a specific name (or leave it empty), and you can choose to have the UDF built immediately. The build can also be done manually later.

Finally, you are presented with a summary dialog for the creation of the UDF (Figure 12-19). Click **Finish**.



New SQL User-Defined Function (UDF)

Summary

The UDF settings that you specified are summarized here.

Setting	Value
Name	InsuredPremium
Programming language	SQL
Template selected	Default template
Build	Yes
Template option :Statement	Scalar - DECIMAL
Template option :Result set	SQL statement/expression

Parameters

Name	SQL Type
id	SMALLINT

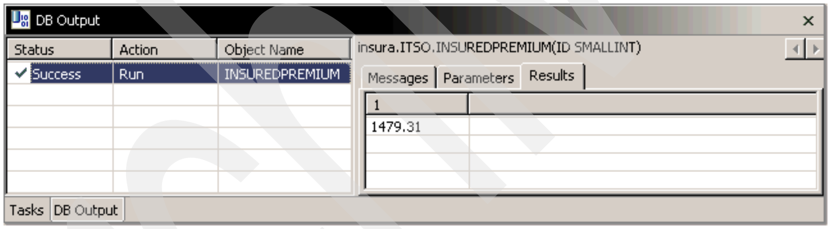
Figure 12-19 Creating a user-defined function: summary

The UDF is saved and opened in the editor:

```
CREATE FUNCTION ITSO.InsuredPremium( id SMALLINT )
    RETURNS DECIMAL(10,2)
------------------------------------------------------------------------
-- SQL UDF (Scalar)
------------------------------------------------------------------------
F1: BEGIN ATOMIC
    RETURN SELECT  SUM(ITSO.POLICY.ACTUAL_PREMIUM)
FROM
   ITSO.INSURED, ITSO.POLICY
WHERE
   ITSO.INSURED.INSURED_ID = ITSO.POLICY.INSURED_ID
   AND ITSO.INSURED.INSURED_ID = InsuredPremium.id;
END
```

If you selected **Build**, then the UDF is built and stored in the database. Otherwise select the new UDF in the Data Definition view and **Build** (context).

To test the UDF, select the **InsuredPremium** UDF in the Data Definition view and **Run** (context). Enter 987 as a parameter, click **OK**, and the result is displayed in the DB Output view ().



Status	Action	Object Name
✓ Success	Run	INSUREDPREMIUM

insura.ITSO.INSUREDPREMIUM(ID SMALLINT)

1
1479.31

Figure 12-20 Testing a UDF in Application Developer

Testing UDFs with SQL statements

You can use UDFs in SQL statements, for example in a DB2 command window:

```
db2 select insured_id, itso.insuredpremium(insured_id) as Induredpremium
       from itso.insured

INSURED_ID INDUREDPREMIUM
---------- --------------
       555        9876.54
       556         222.25
       666              -
       987        1479.31

db2 "select * from itso.insured where itso.insuredpremium(insured_id)>1000"

INSURED_ID FIRST_NAME           LAST_NAME            MARITAL_STATUS AGE
---------- -------------------- -------------------- -------------- ------
       987 Shrinivas            Kulkarni             S                  25
       555 Eva                  Vanhex               M                  35
```

12.2.6 Generating XML from an SQL statement

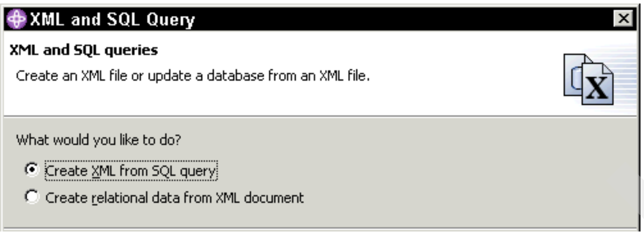
The XML and SQL Query wizard allows you to extract data from a relational database and save it in an XML file.

First, you must have already established a connection to the database (refer to 12.2.1, “Creating a database connection” on page 340). Then, you must create an SQL Query statement to extract the data desired. Finally, you can invoke this wizard and use the SQL Query statement to extract the data and then save it into a XML file.

We use the SQL statement, `InsuredPolicies`, created in 12.2.4, “Creating an SQL statement” on page 346 to extract the data from the database and write it out as an XML file.

First, invoke the XML and SQL Query wizard. You can start the wizard by selecting **File -> New -> Other -> XML -> XML and SQL Query.**

In the first panel select **Create XML from SQL query** and click **Next** (Figure 12-21 on page 357).


XML and SQL Query

XML and SQL queries

Create an XML file or update a database from an XML file.

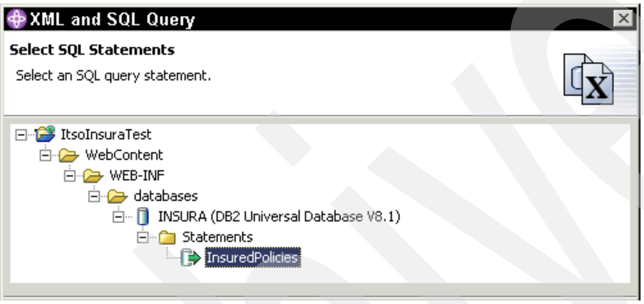
What would you like to do?

● Create XML from SQL query

○ Create relational data from XML document

Figure 12-21 Generating XML from SQL query: initial

On the next panel, expand the project, select the SQL statement to be used and click **Next** (Figure 12-22).


XML and SQL Query

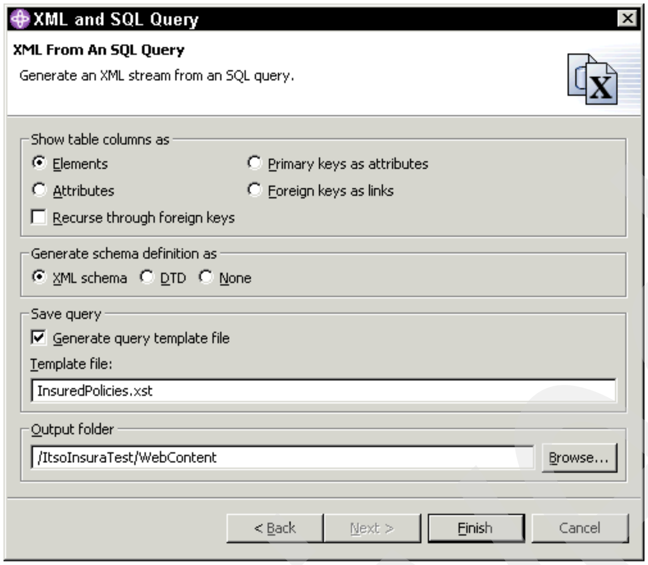
Select SQL Statements

Select an SQL query statement.

- ItsoInsuraTest
  - WebContent
    - WEB-INF
      - databases
        - INSURA (DB2 Universal Database V8.1)
          - Statements
            - InsuredPolicies

Figure 12-22 Generating XML from SQL query: select SQL statement

On the next panel, customize the transformation settings in regard to generation of elements or attributes, set the output folder, and click **Finish** (Figure 12-23 on page 358).



XML and SQL Query

XML From An SQL Query

Generate an XML stream from an SQL query.

Show table columns as

● Elements ○ Primary keys as attributes
○ Attributes ○ Foreign keys as links
□ Recurse through foreign keys

Generate schema definition as

● XML schema ○ DTD ○ None

Save query

☑ Generate query template file

Template file: InsuredPolicies.xst

Output folder: /ItsoInsuraTest/WebContent

Figure 12-23 Generating XML from SQL query: transformation settings

You are prompted for the `:marriedstatus` parameter; enter either `'M'` or `'S'`.

Output files

Table 12-1lists all of the files that are generated.

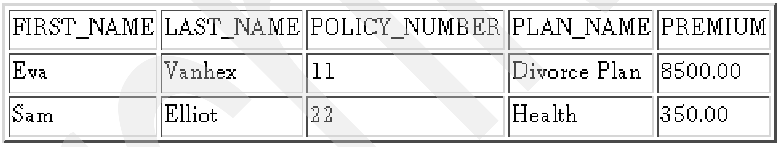
Table 12-1 XML from SQL generated files

File Name	Description
InsuredPolicies.xml	XML file with database data
InsuredPolicies.xsd InsuredPolicies.dtd	XML Schema or DTD, if selected
InsuredPolicies.xsl	XSL stylesheet to transform XML into HTML
InsuredPolicies.html	HTML file is generated from XML using XSL
InsuredPolicies.xst	Query Template with connection information and SQL statement; can be used in application programs to run SQL-to-XML transformation

Example 12-1 on page 359 shows the generated XML file and Figure 12-24 on page 359 shows the generated HTML file when opened in Page Designer.

Example 12-1 Generating XML from SQL query: XML output

```
<?xml version="1.0" encoding="UTF-8"?>
<SQLResult xmlns="http://www.ibm.com/INSURED_POLICY_P_TYPE"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.ibm.com/INSURED_POLICY_P_TYPE
      InsuredPolicies.xsd">
    <INSURED_POLICY_P_TYPE>
        <FIRST_NAME>Eva</FIRST_NAME>
        <LAST_NAME>Vanhex</LAST_NAME>
        <POLICY_NUMBER>11</POLICY_NUMBER>
        <PLAN_NAME>Divorce Plan</PLAN_NAME>
        <PREMIUM>8500.00</PREMIUM>
    </INSURED_POLICY_P_TYPE>
    <INSURED_POLICY_P_TYPE>
        <FIRST_NAME>Sam</FIRST_NAME>
        <LAST_NAME>Elliot</LAST_NAME>
        <POLICY_NUMBER>22</POLICY_NUMBER>
        <PLAN_NAME>Health</PLAN_NAME>
        <PREMIUM>350.00</PREMIUM>
    </INSURED_POLICY_P_TYPE>
</SQLResult>
```



FIRST_NAME	LAST_NAME	POLICY_NUMBER	PLAN_NAME	PREMIUM
Eva	Vanhex	11	Divorce Plan	8500.00
Sam	Elliot	22	Health	350.00

Figure 12-24 Generating XML from SQL query: HTML output

12.2.7 Updating relational tables from XML

Besides being able to extract data and save it into an XML file, the XML and SQL Query wizard can read an XML file and update, insert or delete data in a database. In the next example, we insert a new row into the `INSURED` table.

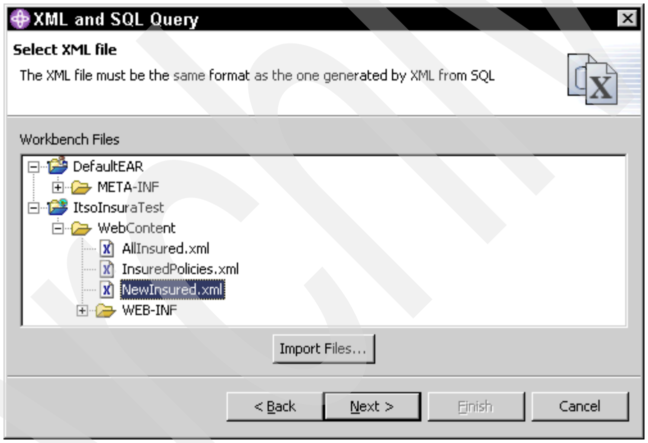
We will start by creating a file called NewInsured.xml (Example 12-2 on page 360). The format of the file can be created using the wizard from a simple SQL statement that lists the table (`AllInsured`).

Example 12-2 Table insert from XML: input file

```
<?xml version="1.0" encoding="UTF-8"?>
<SQLResult>
    <INSURED INSURED_ID="666">
        <FIRST_NAME>Unknown</FIRST_NAME>
        <LAST_NAME>Lady</LAST_NAME>
        <MARITAL_STATUS>S</MARITAL_STATUS>
        <AGE>24</AGE>
    </INSURED>
</SQLResult>
```

We start the wizard (**New -> XML -> XML and SQL Query**). This time, we select **Create relational data from XML document** and click **Next** (Figure 12-21 on page 357).

In the next panel, select the XML file to be used as input and click **Next** (Figure 12-25).



XML and SQL Query

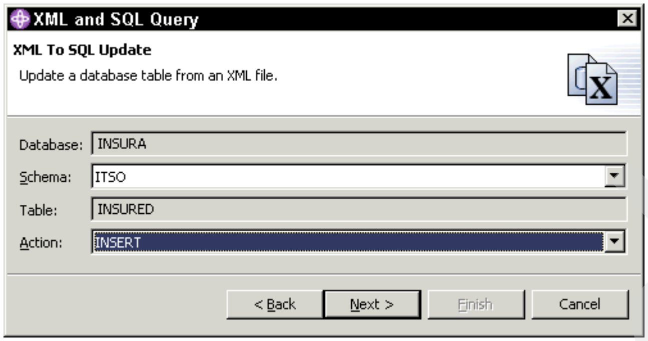
Select XML file

The XML file must be the same format as the one generated by XML from SQL

Figure 12-25 Table insert from XML: select XML file

In the next panel, we connect to the database using the existing `ConInsura` connection and click **Next**.

Next, we select the **INSERT** action (Figure 12-26 on page 361).



XML To SQL Update

Update a database table from an XML file.

Database: INSURA

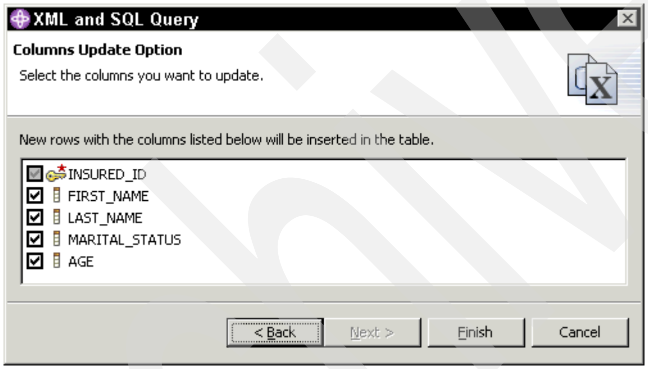
Schema: ITSO

Table: INSURED

Action: INSERT

Figure 12-26 Table insert from XML: action

Finally, we select the columns to be updated and click **Finish** (Figure 12-27).



Columns Update Option

Select the columns you want to update.

New rows with the columns listed below will be inserted in the table.

- ☑ INSURED_ID
- ☑ FIRST_NAME
- ☑ LAST_NAME
- ☑ MARITAL_STATUS
- ☑ AGE

Figure 12-27 Table insert from XML: columns

The database is updated with the content of the XML file. You can verify this using a DB2 command window:

```
INSURED_ID FIRST_NAME           LAST_NAME            MARITAL_STATUS AGE
---------- -------------------- -------------------- -------------- ------
       987 Shrinivas            Kulkarni             S                  25
       555 Eva                  Vanhex               M                  35
       556 Sam                  Elliot               M                  45
       666 Unknown              Lady                 S                  24
```

12.2.8 XSL transformations

With XSL, you can transform an XML file into many other formats. The XML and SQL Query wizard (see 12.2.6, “Generating XML from an SQL statement” on page 356) used XSL to transform the XML output of the SQL query into HTML.

You can manually perform the transformation again by selecting both the **InsuredPolicies.xml** and **InsuredPolicies.xsl** files and clicking **Transform -> XML** (Figure 12-28).

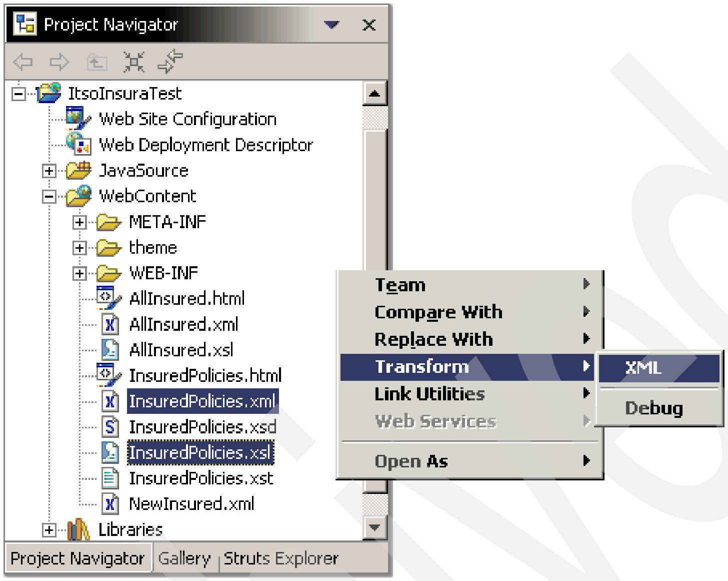


Figure 12-28 XSL transformation

An HTML output file named InsuredPolicies_InsuredPolicies_transform.html is generated and displayed in a browser window.

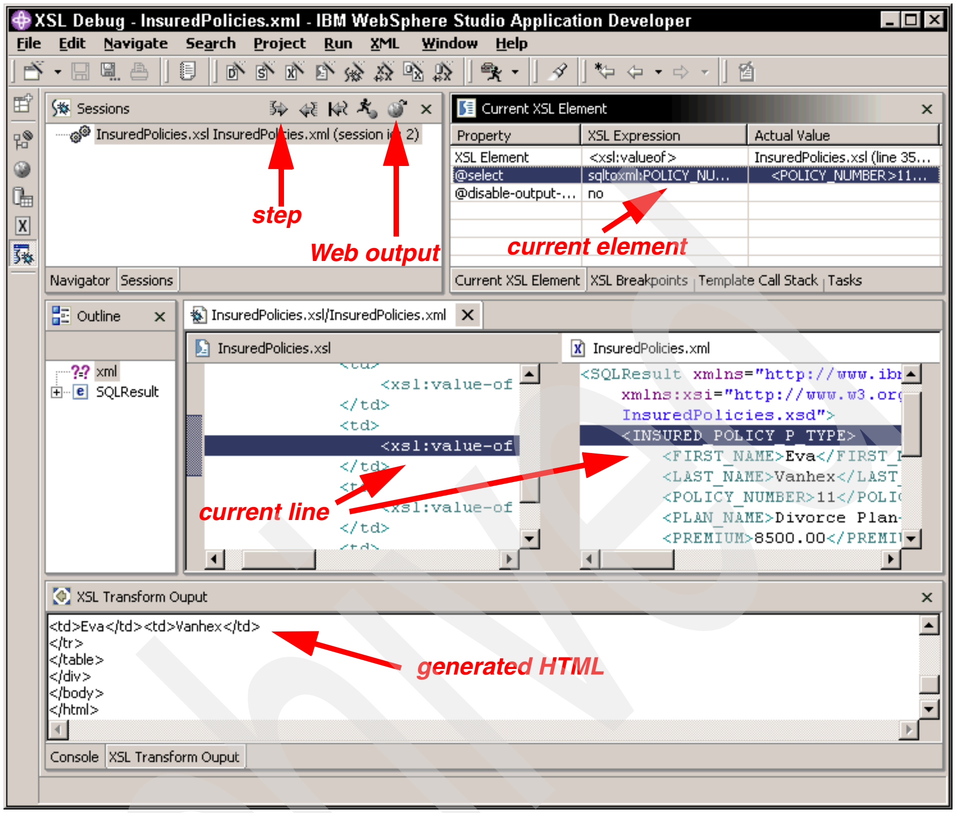
XSL debugging

The same transformation can be run using the XSL debugger. This time select the XML and XSL files and click **Transform -> Debug**.

The XSL Debug perspective opens (Figure 12-29 on page 363).

Use the step icon in the Sessions view to step through the XSL and XML code. The current XSL element is displayed (top right) and the generated HTML is displayed in the XSL Transformation Output view.

You can also set breakpoints in the XSL and run to the breakpoint.


XSL Debug - InsuredPolicies.xml - IBM WebSphere Studio Application Developer
File Edit Navigate Search Project Run XML Window Help
Sessions
InsuredPolicies.xsl InsuredPolicies.xml (session id 2)
step
Web output
Navigator Sessions
Current XSL Element
current element
Current XSL Element XSL Breakpoints Template Call Stack Tasks
Outline
xml
SQLResult
InsuredPolicies.xsl/InsuredPolicies.xml
InsuredPolicies.xsl
InsuredPolicies.xml
current line
XSL Transform Ouput
generated HTML
Console XSL Transform Ouput

Property	XSL Expression	Actual Value
XSL Element	<xsl:valueof>	InsuredPolicies.xsl (line 35…
@select	sqltoxml:POLICY_NU…	<POLICY_NUMBER>11…
@disable-output-…	no	

```
        <td>
            <xsl:value-of
        </td>
        <td>
            <xsl:value-of
        </td>
        <td>
            <xsl:value-of
        </td>
        <td>
```

```
<SQLResult xmlns="http://www.ibı
    xmlns:xsi="http://www.w3.orı
    InsuredPolicies.xsd">
    <INSURED_POLICY_P_TYPE>
        <FIRST_NAME>Eva</FIRST_I
        <LAST_NAME>Vanhex</LAST_
        <POLICY_NUMBER>11</POLIC
        <PLAN_NAME>Divorce Plan-
        <PREMIUM>8500.00</PREMII
```

```
<td>Eva</td><td>Vanhex</td>
</tr>
</table>
</div>
</body>
</html>
```

Figure 12-29 XSL Debug perspective

XML to XML transformation

XSL transformation can also transform an XML file into another XML file where the mapping is constructed from a source XML, XSD or DTD file to a target XML, XSD or DTD file.

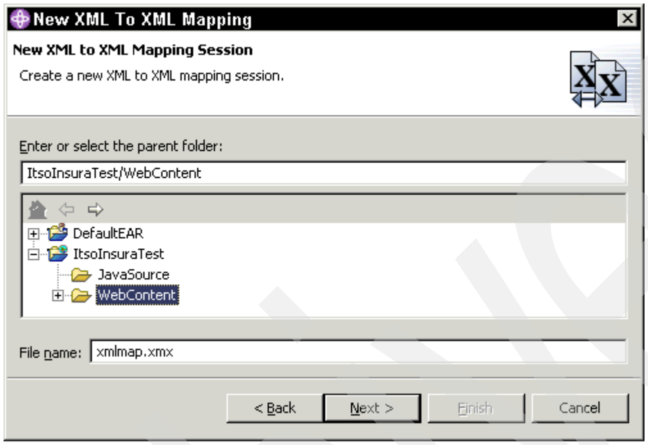
Suppose we want to transform the XML file for database insert (Example 12-2 on page 360) into a new format:

```
<SQLResult>                                     <SQLResult>
<INSURED INSURED_ID="666">                      <INSURED>
                                                   <ID>666</ID>
   <FIRST_NAME>Unknown</FIRST_NAME>                <FIRST>Unknown</FIRST>
   <LAST_NAME>Lady</LAST_NAME>                     <LAST>Lady</LAST>
   <MARITAL_STATUS>S</MARITAL_STATUS>              <MARRIED>S</MARRIED>
   <AGE>24</AGE>                                   <AGE>24</AGE>
</INSURED>                                      </INSURED>
</SQLResult>                                    </SQLResult>
```

The original file is NewInsured.xml, the target file is NewInsured2.xml.

The XML to XML Mapping wizard can generate a mapping and an XSL file for us. Start the wizard using **File -> New -> Other -> XML -> XML to XML Mapping**.

On the first panel, select the output folder and mapping file name (Figure 12-30).



New XML To XML Mapping

New XML to XML Mapping Session

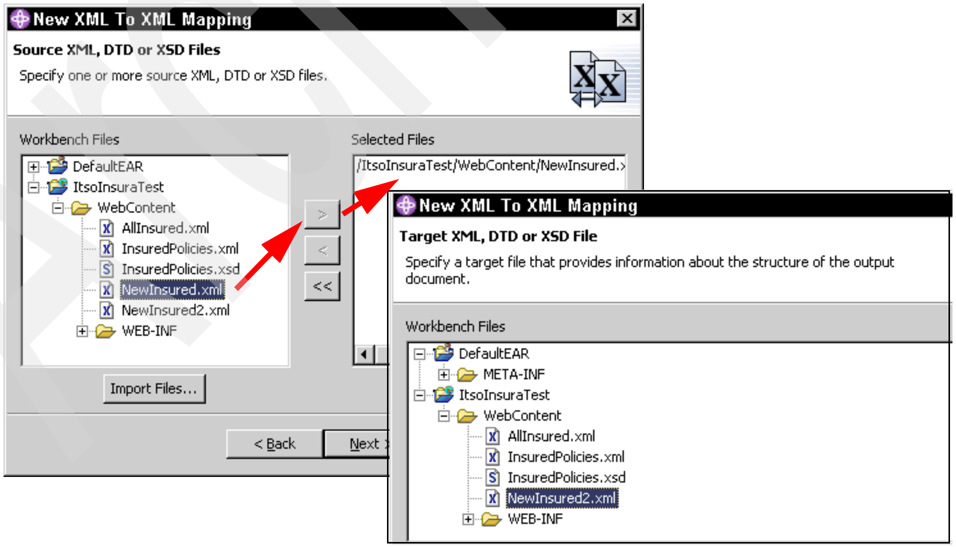
Create a new XML to XML mapping session.

Enter or select the parent folder: ItsoInsuraTest/WebContent

File name: xmlmap.xmx

Figure 12-30 XML to XML mapping: mapping file

On the next two panels, select the source and target files (Figure 12-31).



New XML To XML Mapping

Source XML, DTD or XSD Files

Specify one or more source XML, DTD or XSD files.

Selected Files: /ItsoInsuraTest/WebContent/NewInsured.x

New XML To XML Mapping

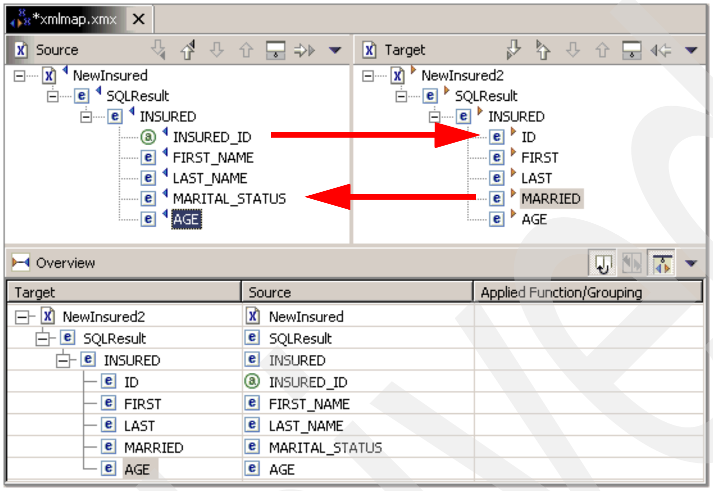
Target XML, DTD or XSD File

Specify a target file that provides information about the structure of the output document.

Figure 12-31 XML to XML mapping: source and target files

On the final panel, select the root element (**SQLResult**) and click **Finish**.

The mapping file (xmlmap.xmx) opens in the mapping editor. Complete the mapping by drag and drop of elements of the source to the target (or opposite). Figure 12-32 shows the completed mapping.



Target	Source	Applied Function/Grouping
NewInsured2	NewInsured	
SQLResult	SQLResult	
INSURED	INSURED	
ID	INSURED_ID	
FIRST	FIRST_NAME	
LAST	LAST_NAME	
MARRIED	MARITAL_STATUS	
AGE	AGE	

Figure 12-32 XML to XML mapping: completed mapping

Save the mapping file. Before closing the mapping editor, select **Mapping -> Generate XSLT Script**. In the dialog, select the target folder (**WebContent**) and target file name (**NewInsured2.xsl**).

To execute the transformation, select both the **NewInsured.xml** and **NewInsured2.xsl** files and click **Transform -> XML**. The translated file opens in a browser and looks identical to the NewInsured2.xml file.

This transformation can be executed for any XML file that matches the original layout. In many cases, the original and target layout would be given by XSD or DTD files.

12.2.9 Mapping the relational table to XML

To store XML files in relational tables using XML Extender, you require a document access definition (DAD) file. To create a DAD file, you need a mapping between XML and a relational table (or a set of tables). For simplicity's sake, we will only use one table in this example.

Generating a DTD

To create a mapping between XML and a table, we require a DTD that specifies the format of the XML files.

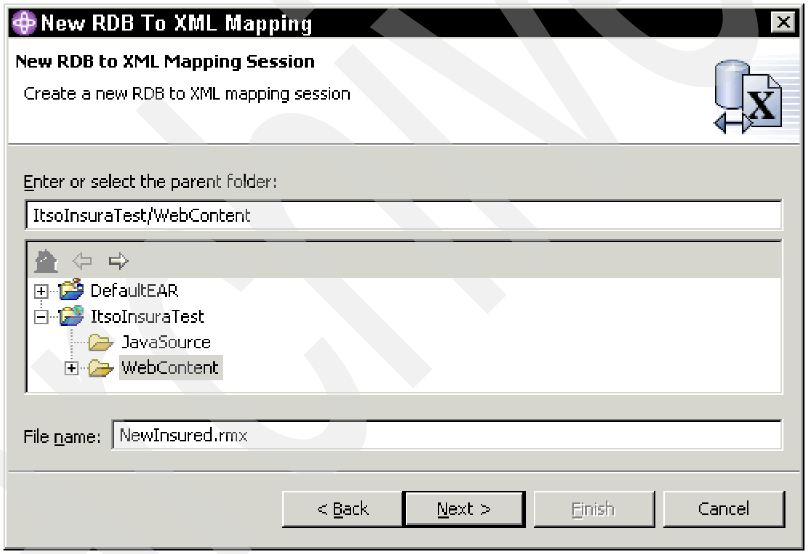
We can generate a DTD from an existing XML file:

1. Select the **NewInsured2.xml** file and click **Generate -> DTD** (context).
2. Select the target folder and set the output file name (`NewInsured2.dtd`).
3. Click **Finish** and the DTD is generated.

Creating an RDB to XML mapping

Start the RDB to XML Mapping wizard by selecting **File -> New -> Other -> XML -> RDB to XML Mapping** and clicking **Next**.

In the first panel (Figure 12-33), select the output folder (**WebContent**) and the mapping file name (**NewInsured.rmx**).



New RDB To XML Mapping

New RDB to XML Mapping Session

Create a new RDB to XML mapping session

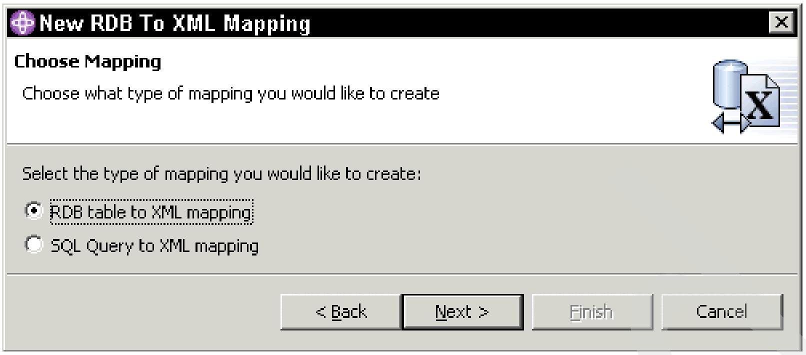
Enter or select the parent folder: ItsoInsuraTest/WebContent

File name: NewInsured.rmx

Figure 12-33 RDB to XML mapping: output file

In the next panel (Figure 12-34 on page 367), select **RDB table to XML mapping**. The other choice is **SQL Query to XML mapping**, where you would use an existing SQL statement.

This creates a mapping between a relational database table and an XML document. The DAD file you generate from this type of mapping can be used to store and retrieve data from DB2 databases.


New RDB To XML Mapping

Choose Mapping

Choose what type of mapping you would like to create

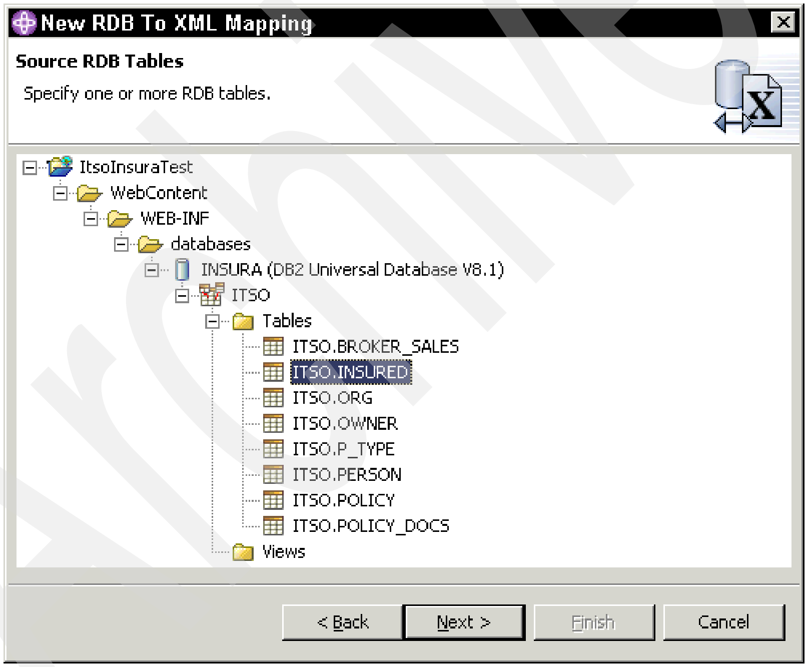
Select the type of mapping you would like to create:

● RDB table to XML mapping

○ SQL Query to XML mapping

Figure 12-34 RDB to XML mapping: mapping type

On the next panel (Figure 12-35), you select the database table to use. Notice that you must already have imported the database schema into the project.


New RDB To XML Mapping

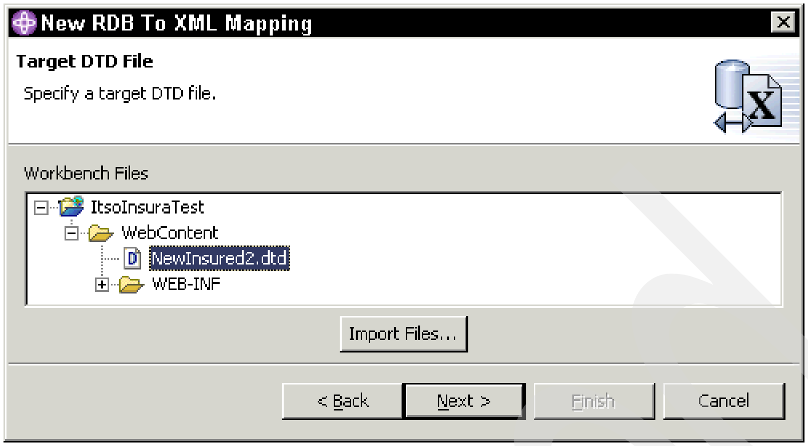
Source RDB Tables

Specify one or more RDB tables.

```
ItsoInsuraTest
  WebContent
    WEB-INF
      databases
        INSURA (DB2 Universal Database V8.1)
          ITSO
            Tables
              ITSO.BROKER_SALES
              ITSO.INSURED
              ITSO.ORG
              ITSO.OWNER
              ITSO.P_TYPE
              ITSO.PERSON
              ITSO.POLICY
              ITSO.POLICY_DOCS
            Views
```

Figure 12-35 RDB to XML mapping: table selection

On the next panel (Figure 12-36 on page 368), you select a target DTD file to map the database table. If you want to use a DTD file that is not currently in the workbench, click **Import File** and fill in the fields in the Import wizard as necessary. Click **Next**.


New RDB To XML Mapping

Target DTD File

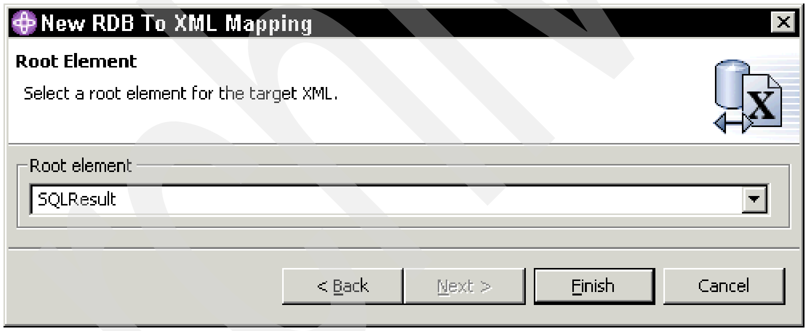
Specify a target DTD file.

Workbench Files

- ItsoInsuraTest
  - WebContent
    - NewInsured2.dtd
    - WEB-INF

Figure 12-36 RDB to XML mapping: DTD selection

Next, you select the appropriate root element (Figure 12-37). When your DTD is transformed into an XML document for mapping, the selected element will be used as the root element.


New RDB To XML Mapping

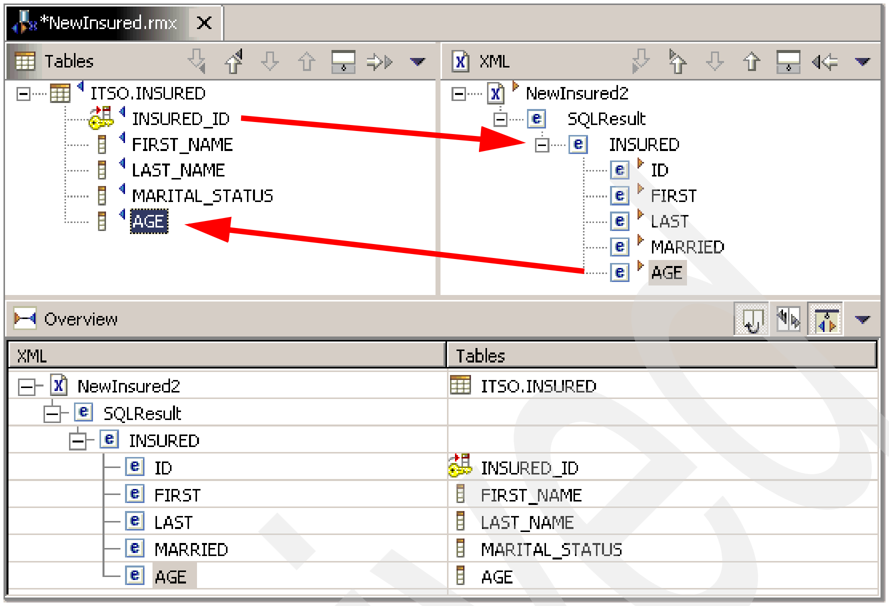
Root Element

Select a root element for the target XML.

Root element: SQLResult

Figure 12-37 RDB to XML mapping: root element

At this point, the RDB to XML mapping editor is displayed with the source and target that you specified. You drag source elements from the Tables pane and drop them on top of target elements in the XML pane to establish a mapping (Figure 12-38 on page 369).



XML	Tables
NewInsured2	ITSO.INSURED
SQLResult	
INSURED	
ID	INSURED_ID
FIRST	FIRST_NAME
LAST	LAST_NAME
MARRIED	MARITAL_STATUS
AGE	AGE

Figure 12-38   RDB to XML mapping: mapping editor

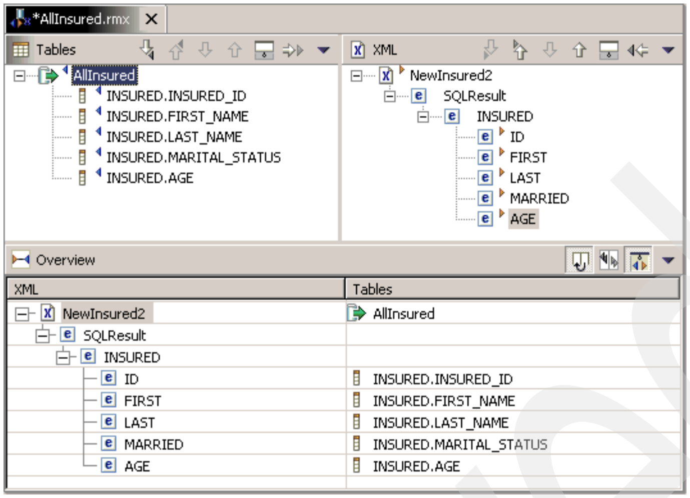
Save the mapping. We will use this mapping to create a DAD file.

12.2.10 Mapping the SQL query to XML

The mapping of an SQL `SELECT` statement to XML is almost the same as the mapping of a table.

Use the same RDB to XML Mapping wizard:

1. Set the output file as `AllInsured.rmx` (Figure 12-33 on page 366).
2. Select **SQLQuery to XML** mapping (Figure 12-34 on page 367).
3. Select the SQL statement (instead of a table). Expand the WebContent folder and the INSURA database until you can select the `AllInsured` SQL statement.
4. Select the same DTD (Figure 12-36 on page 368).
5. Select the same root element (Figure 12-37 on page 368).
6. In the mapping editor, map the columns of the SQL result to the XML elements (Figure 12-39 on page 370). Save the mapping when finished.



XML	Tables
NewInsured2	AllInsured
SQLResult	
INSURED	
ID	INSURED.INSURED_ID
FIRST	INSURED.FIRST_NAME
LAST	INSURED.LAST_NAME
MARRIED	INSURED.MARITAL_STATUS
AGE	INSURED.AGE

Figure 12-39 SQL query to XML mapping: mapping editor

12.2.11 Generating a DAD file

The relational database (RDB) to XML mapping editor generates a RMX file which can be used to generate a document access definition (DAD) file that can be used with the IBM DB2 XML Extender to generate XML documents from existing relational data, or to decompose XML documents into relational data.

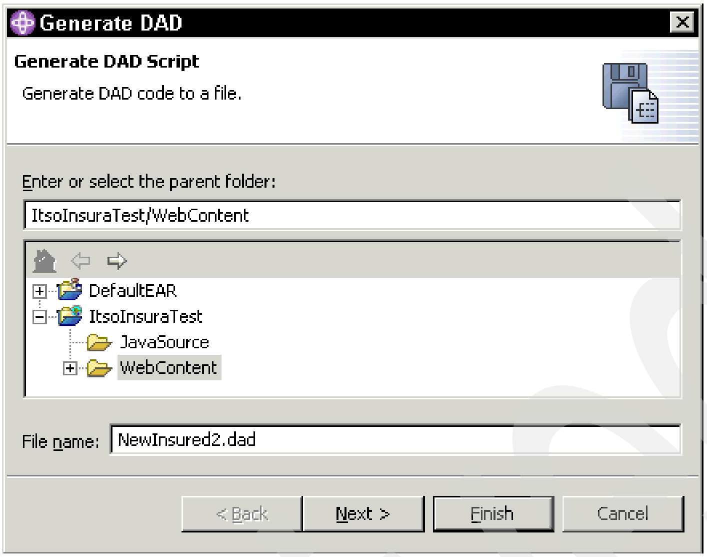
The DAD file is an XML formatted document that associates an XML document structure with tables in a DB2 database.

Once you have generated a RMX file using the RDB to XML Mapping wizard, you can generate a DAD file in two ways:

- Open the RMX file and selecting **Mapping -> Generate DAD**.
- Select the RMX file in the Project Navigator and click **Generate DAD** (context).

We will use the NewInsured.rmx file to generate a DAD file.

On the first panel, select the destination directory and the file name for the DAD file (Figure 12-40 on page 371).



Generate DAD Script

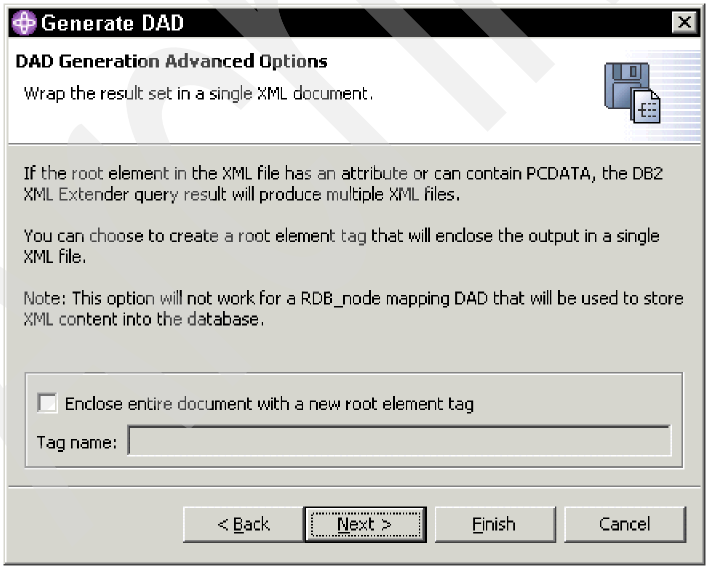
Generate DAD code to a file.

Enter or select the parent folder: ItsoInsuraTest/WebContent

File name: NewInsured2.dad

Figure 12-40 Generate DAD file: output

On the next panel, you can choose to enclose the output into a new root element (Figure 12-41).



DAD Generation Advanced Options

Wrap the result set in a single XML document.

If the root element in the XML file has an attribute or can contain PCDATA, the DB2 XML Extender query result will produce multiple XML files.

You can choose to create a root element tag that will enclose the output in a single XML file.

Note: This option will not work for a RDB_node mapping DAD that will be used to store XML content into the database.

□ Enclose entire document with a new root element tag

Tag name:

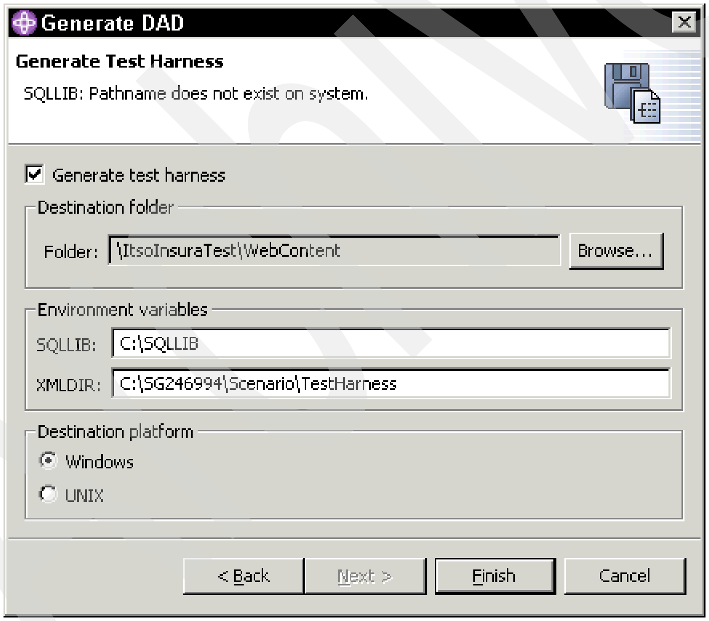
Figure 12-41 Generate DAD file: enclose document

If the root element of your target XML document has an attribute, or can contain PCDATA, the DB2 XML Extender query will return multiple XML documents. Select **Enclose entire document with a new root element tag** to add a new tag to the XML document that will enclose the contents of the file so the XML Extender query result is returned as a single XML document. Specify the tag name, then click **Next**.

Note: If you select this option for an RDB to XML mapping DAD, you will be able to retrieve XML content, but will not be able to store XML content in the database.

If you wish to create your DAD file without generating a test harness, click **Finish**.

To generate a test harness to test your script, select **Generate test harness** (Figure 12-42). A test harness is a series of script files used to enable a DB2 database for use by the DB2 XML Extender. Once enabled, it tests composing XML from data as well as decomposing XML files into relational data.



Generate DAD

Generate Test Harness

SQLLIB: Pathname does not exist on system.

☑ Generate test harness

Destination folder

Folder: \ItsoInsuraTest\WebContent

Environment variables

SQLLIB: C:\SQLLIB

XMLDIR: C:\SG246994\Scenario\TestHarness

Destination platform

● Windows

○ UNIX

Figure 12-42 Generate DAD file: test harness

1. Type the path of the DB2 SQLLIB directory, for example `C:\SQLLIB`
2. Type the path of the DB2 XML Extender directory, for example `C:\DXX`. If you are working with DB2 UDB V8.1, this field will not appear because V8.1 includes DB2 XML Extender.

3. In the XMLDIR text field, type the path of the directory in which you want the output XML file to be created. This is the same directory where the source XML files used to store information in the database are located. For example, enter `c:\SG246994\Scenario\TestHarness`.
4. Select the radio button that corresponds to the destination platform (the platform on which the test harness will be executed).
5. The default output folder is `WebContent` (you can change that by clicking **Browse**).
6. Click **Finish**.

The generated files are:

- NewInsured2.dad (Example 12-3 on page 374)
- readme.txt—instructions for test harness
- Four bat files of the test harness (setup.bat, retrieveXML.bat, storeXML.bat, and updateDTD_REF.bat)

The DAD file opens in the editor. When testing, we found out that the table element that repeats must have the attribute `multi_occurrence="YES"` added:

```
<element_node name="INSURED"  multi_occurrence="YES">
```

Test harness

To use the test harness, select the **NewInsured2.dad**, the **NewInsured2.dtd**, and the four BAT files, and click **Export**. Select **File system** and export the files to the directory specified in Figure 12-42 on page 372:

```
c:\SG246994\Scenario\TestHarness
```

In a command window, go to the TestHarness directory and execute:

```
setup.bat                      ==> start a DB2 command window
retrieveXML.bat                ==> run XML Extender with the DAD file
```

An extract of the generated XML is shown here:

```
<?xml version="1.0"?>
<!DOCTYPE SQLResult PUBLIC "NewInsured2Id" "NewInsured2.dtd">
<SQLResult>
  <INSURED>
    <ID>987</ID>
    <FIRST>Shrinivas           </FIRST>
    <LAST>Kulkarni            </LAST>
    <MARRIED>S </MARRIED>
    <AGE>25</AGE>
  </INSURED>
  <INSURED>
  ......
</SQLResult>
```

Example 12-3 Generate DAD file: generated DAD file with correction

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DAD PUBLIC "dadId" "dad.dtd">
<DAD>
  <dtdid>NewInsured2.dtd</dtdid>
  <validation>NO</validation>
  <Xcollection>
    <prolog>?xml version="1.0"?</prolog>
    <doctype>!DOCTYPE SQLResult PUBLIC "NewInsured2Id"
"NewInsured2.dtd"</doctype>
    <root_node>
      <element_node name="SQLResult">
        <RDB_node>
          <table name="ITSO.INSURED" key="INSURED_ID"/>
          <condition>
            ITSO.INSURED.INSURED_ID=ITSO.INSURED.INSURED_ID
          </condition>
        </RDB_node>
        <element_node name="INSURED" multi_occurrence="YES">   <==== modify
          <element_node name="ID">
            <text_node>
              <RDB_node>
                <table name="ITSO.INSURED"/>
                <column name="INSURED_ID" type="SmallInt"/>
              </RDB_node>
            </text_node>
          </element_node>
          <element_node name="FIRST">
            <text_node>
              <RDB_node>
                <table name="ITSO.INSURED"/>
                <column name="FIRST_NAME" type="Character(20)"/>
              </RDB_node>
            </text_node>
          </element_node>
          <element_node name="LAST">
            <text_node>
              <RDB_node>
                <table name="ITSO.INSURED"/>
                <column name="LAST_NAME" type="Character(20)"/>
              </RDB_node>
            </text_node>
          </element_node>
         ................
        </element_node>
      </element_node>
    </root_node>
  </Xcollection>
</DAD>
```

12.3 Web services tools

The creation of Web services from SQL statements or stored procedures is a multi-step process:

1. Create a DADX group configuration that specifies how the database is accessed.
2. Create a DADX file from an SQL statement or stored procedure.
3. Create a Web service from the DADX file.

12.3.1 Creating a DADX group configuration

Before you can create a DADX file from a SQL statement and then create a Web service from the DADX file, you have to create a DADX group configuration.

To get started, select **File -> New -> Other -> Web Services -> Web Service DADX Group Configuration** (Figure 12-43 on page 376).

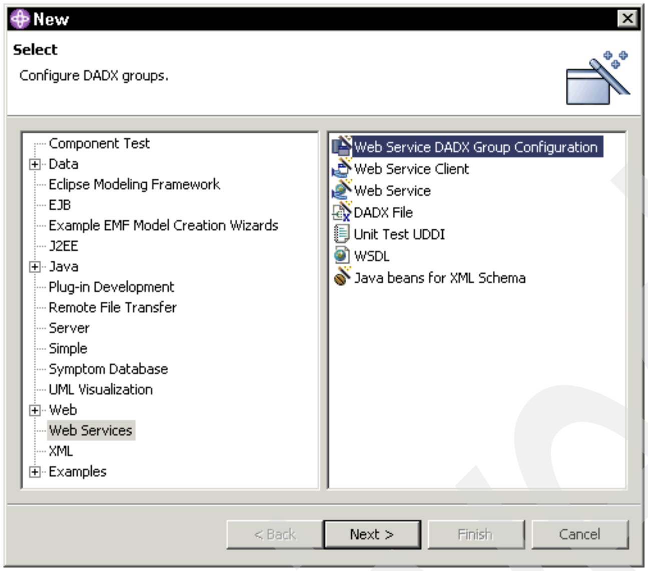


Figure 12-43 Web services tool selection

You will be presented with a list of the existing Web projects to add your group to. Highlight the Web project to add your group to and then select the **Add Group** button. You should then enter the name of the group.

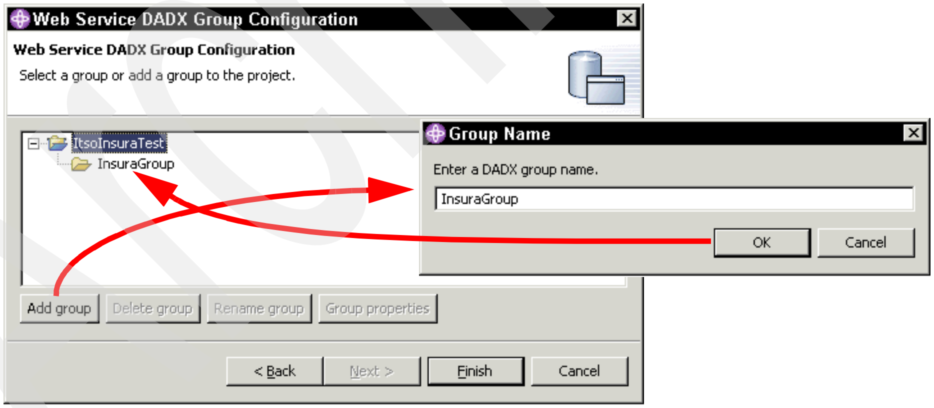
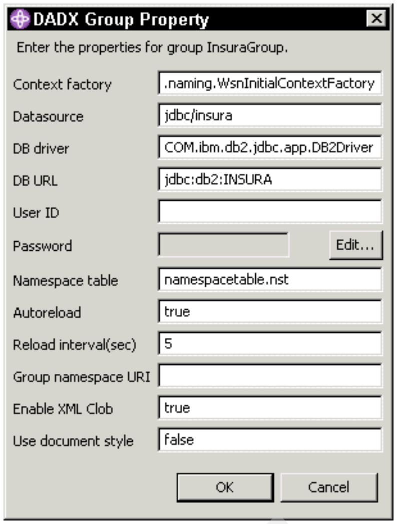


Figure 12-44 Adding a DADX group

Next, select the group and click **Group properties** to specify the database connection parameters. Figure 12-45 on page 377 shows the group properties dialog.



DADX Group Property

Enter the properties for group InsuraGroup.

Context factory	.naming.WsnInitialContextFactory
Datasource	jdbc/insura
DB driver	COM.ibm.db2.jdbc.app.DB2Driver
DB URL	jdbc:db2:INSURA
User ID	
Password	
Namespace table	namespacetable.nst
Autoreload	true
Reload interval(sec)	5
Group namespace URI	
Enable XML Clob	true
Use document style	false

Figure 12-45 DADX group properties

Enter the correct DB URL and optional user ID and password. To use a data source instead of a JDBC driver, enter a data source name (`jdbc/insura`) and the WebSphere context factory:

```
com.ibm.websphere.naming.WsnInitialContextFactory
```

> **Note:** You should set the value of *Use document style* to `true` to generate Web Service Interoperability (WS-I) compliant Web Services. The value `false` generates rpc-type Web services that are not WS-I compliant.

The properties are stored in the group.properties file under JavaSource/groups.InsuraGroup (Example 12-4 on page 378).

Example 12-4 DADX group.properties file

```
#Wed Aug 06 16:30:31 PDT 2003
namespaceTable=namespacetable.nst
groupNamespaceUri=
reloadIntervalSeconds=5
dbDriver=COM.ibm.db2.jdbc.app.DB2Driver
dbURL=jdbc\:db2\:INSURA
enableXmlClob=true
useDocumentStyle=false
password=
datasourceJNDI=jdbc/insura
initialContextFactory=com.ibm.websphere.naming.WsnInitialContextFactory
userID=
autoReload=true
```

12.3.2 Creating a DADX file from an SQL statement

The DADX wizard supports the creation of a DADX file from a combination of one or more SQL statements, stored procedures, and DAD files. A DADX file is an XML file used to create a Web service that accesses a relational database.

A document access definition (DAD) file is a user-specified file that provides control over the mapping of XML document elements to DB2 database columns for storage and retrieval using DB2 XML Extender. The DADX wizard supports only Xcollection DAD. The types of mappings and operations supported by the DADX wizard are as follows:

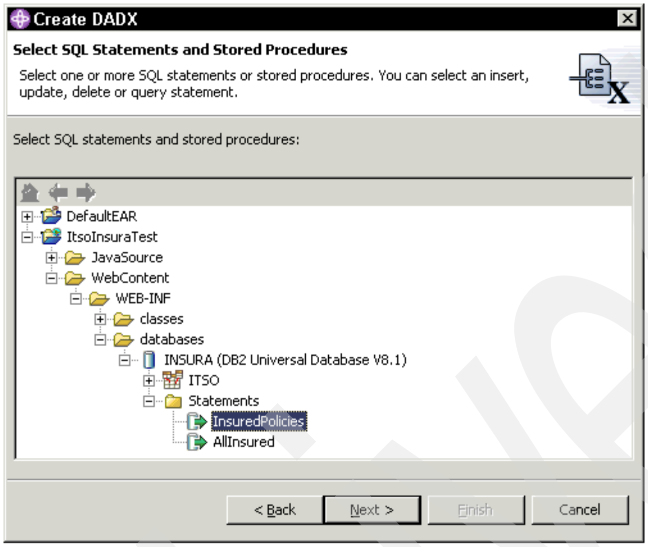
- SQL statement operations (query and update operations based on SQL statements are supported)
- RDB_node mapping (`storeXML` and `retrieveXML` operations based on DAD file mappings are supported).

  If the DAD file defines an SQL statement mapping, you may have to customize the SQL_OVERRIDE information to specify the parameters you want to override from the SQL host variables.
- Stored procedure calls

The SQL statement or stored procedure must exist in the workspace to complete the DADX wizard.

To start the wizard, select **File -> New -> Other -> Web Services -> DADX File**.

In the first panel, select the SQL statements or stored procedures involved. Expand the project folders until you can select the statements and/or stored procedures (Figure 12-46).


Create DADX

Select SQL Statements and Stored Procedures

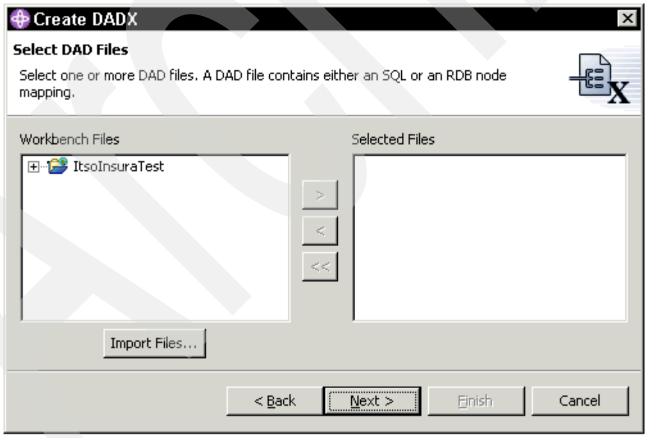
Select one or more SQL statements or stored procedures. You can select an insert, update, delete or query statement.

Select SQL statements and stored procedures:

```
DefaultEAR
ItsoInsuraTest
  JavaSource
  WebContent
    WEB-INF
      classes
      databases
        INSURA (DB2 Universal Database V8.1)
          ITSO
          Statements
            InsuredPolicies
            AllInsured
```

Figure 12-46 DADX generation: select SQL statement

Next, optionally select the DAD files involved (Figure 12-47).


Create DADX

Select DAD Files

Select one or more DAD files. A DAD file contains either an SQL or an RDB node mapping.

Workbench Files

ItsoInsuraTest

Selected Files

Figure 12-47 DADX generation: select DAD file

You can use this page to select one or more DAD files that you would like to include in your DADX file. If needed, click **Import Files** to import a DAD file from your file system.

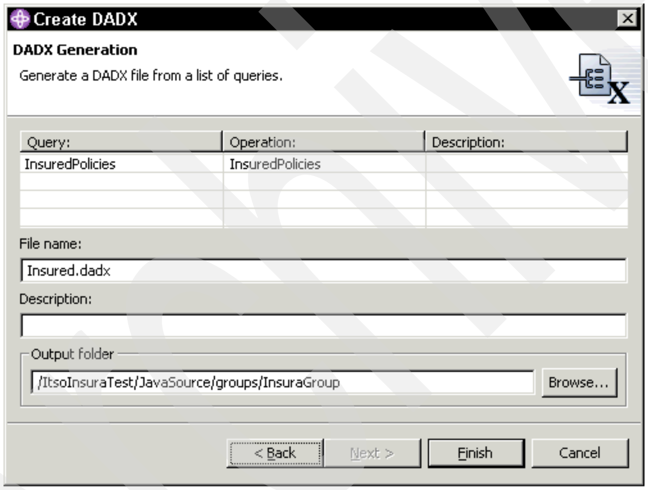
In the next panel (Figure 12-48), you specify the output location and DADX file name. Enter a file name for the DADX file, for example, `Insured.dadx`. It must have the extension of .dadx.

For execution, DADX files must be in the DADX group folder:

```
JavaSource/groups.InsuraGroup
```

Any relevant DAD files also have to be in this folder. At runtime, this is where these files will be expected to be located. If this is not true, you may experience database or server errors.

You can optionally modify or add text to the Operation or Description fields by selecting the row, then selecting the cell that you wish to modify. The Operation field provides a name for the name attribute of the operation tag in the generated DADX file and the Description field provides a description text for the document tag. Type your changes, then press **Enter**.



DADX Generation

Generate a DADX file from a list of queries.

Query:	Operation:	Description:
InsuredPolicies	InsuredPolicies	

File name: Insured.dadx

Description:

Output folder: /ItsoInsuraTest/JavaSource/groups/InsuraGroup

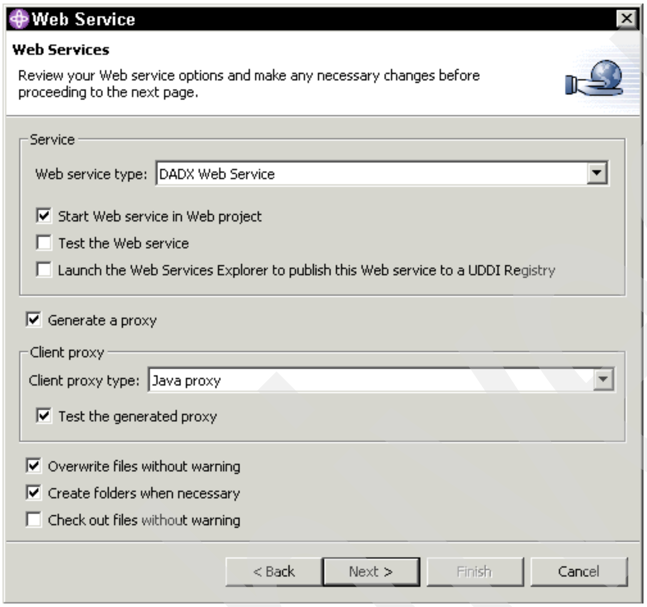
Figure 12-48 DADX generation: output file

Click **Finish** to generate the Insured.dadx file. The file opens in the editor.

12.3.3 Creating a Web service from DADX

You can easily generate a Web service from a DADX file using the Web Service wizard. To start the wizard, select the generated **Insured.dadx** file and click **File -> New -> Other -> Web Services -> Web Service**.

In the first panel of the Web Service dialog (Figure 12-49 on page 381), make sure that the Web service type is set to `DADX Web Service`. Then, for testing purposes, you want to create a test client. Select **Generate a proxy** and **Test the generated proxy**. A second Web project will be automatically created to hold the test client. If the primary project's name is *MyProject*, the test client project name will be *MyProjectClient*.



Web Service

Web Services

Review your Web service options and make any necessary changes before proceeding to the next page.

Service

Web service type: DADX Web Service

☑ Start Web service in Web project

☐ Test the Web service

☐ Launch the Web Services Explorer to publish this Web service to a UDDI Registry

☑ Generate a proxy

Client proxy

Client proxy type: Java proxy

☑ Test the generated proxy

☑ Overwrite files without warning

☑ Create folders when necessary

☐ Check out files without warning

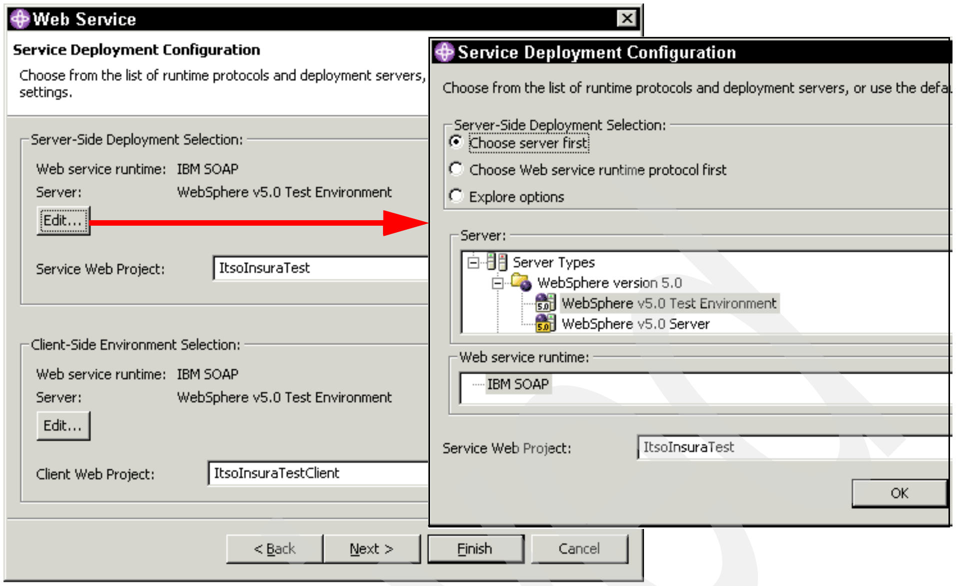
Figure 12-49 Web Service wizard: initial

In the next panel (Figure 12-50 on page 382), we set up the deployment configuration. We can select an existing test server, or we can have a server created for us (we have not defined a server yet). Make sure that the project names are correct:

- `ItsoInsuraTest`—our Web project
- `ItsoInsuraTestClient`—a new Web project for the proxy and the test client

Click **Edit** for the server-side deployment selection. A WebSphere v5.0 Test Environment server will be created.

Click **Next**. You will get a warning stating that the Web service is not WS-I compliant. DB2 Web services are not created using the latest Web services standards. Click **Ignore**.



Web Service

Service Deployment Configuration

Choose from the list of runtime protocols and deployment servers, settings.

Server-Side Deployment Selection:

Web service runtime: IBM SOAP

Server: WebSphere v5.0 Test Environment

Service Web Project: ItsoInsuraTest

Client-Side Environment Selection:

Web service runtime: IBM SOAP

Server: WebSphere v5.0 Test Environment

Client Web Project: ItsoInsuraTestClient

Service Deployment Configuration

Choose from the list of runtime protocols and deployment servers, or use the defa

Server-Side Deployment Selection:

- ● Choose server first
- ○ Choose Web service runtime protocol first
- ○ Explore options

Server:

- Server Types
  - WebSphere version 5.0
    - WebSphere v5.0 Test Environment
    - WebSphere v5.0 Server

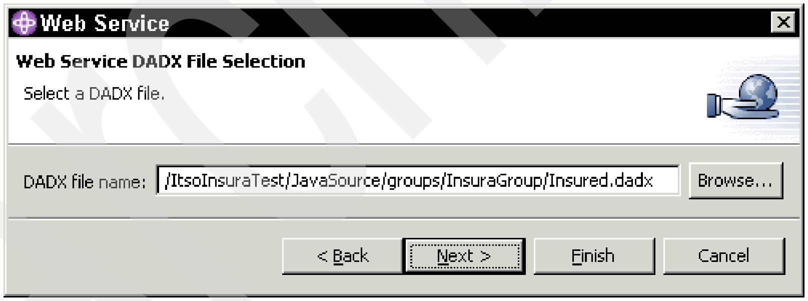
Web service runtime:

- IBM SOAP

Service Web Project: ItsoInsuraTest

Figure 12-50 Web Service wizard: deployment environment

In the next panel (Figure 12-51), select the DADX file. If you did not select the file before starting the wizard, click **Browse** to locate the DADX file. You will find the DADX file usually in the *groups* folder under JavaSource.



Web Service

Web Service DADX File Selection

Select a DADX file.

DADX file name: /ItsoInsuraTest/JavaSource/groups/InsuraGroup/Insured.dadx

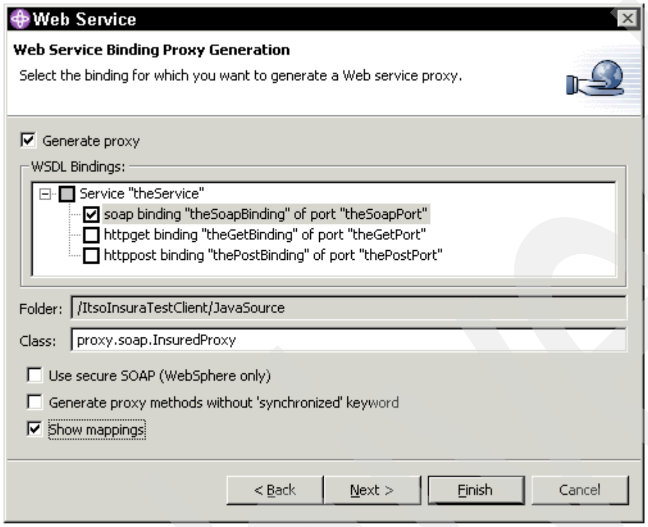
Figure 12-51 Web Service wizard: DADX selection

At this point, a default server is created for testing.

The next panel displays the group properties (see Figure 12-45 on page 377). You can make modifications to the properties or accept the defaults that you set up earlier.

The server is now started (be patient). In the next panel (Figure 12-52), you specify how the Web service bindings are generated. Select **SOAP binding** and **Show mappings**.

The proxy is generated into the `ItsoInsuraTestClient` project.



Web Service

Web Service Binding Proxy Generation

Select the binding for which you want to generate a Web service proxy.

☑ Generate proxy

WSDL Bindings:

- □ Service "theService"
  - ☑ soap binding "theSoapBinding" of port "theSoapPort"
  - □ httpget binding "theGetBinding" of port "theGetPort"
  - □ httppost binding "thePostBinding" of port "thePostPort"

Folder: /ItsoInsuraTestClient/JavaSource

Class: proxy.soap.InsuredProxy

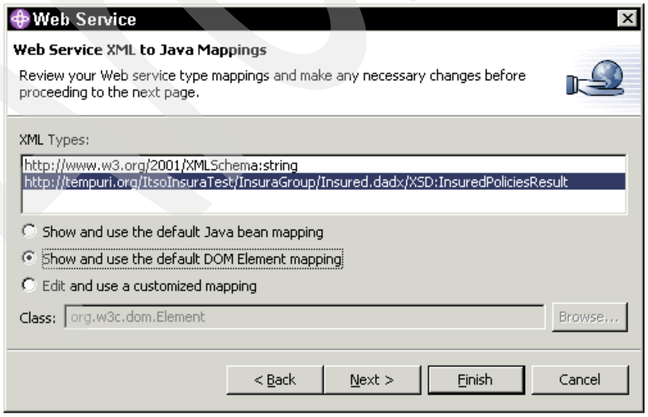
□ Use secure SOAP (WebSphere only)

□ Generate proxy methods without 'synchronized' keyword

☑ Show mappings

Figure 12-52   Web Service wizard: proxy

In the next panel (Figure 12-53), you select how the Web service XML stream is mapped to Java. If you do not want to further transform the XML with XSL, select **Show and use the default Java bean mapping**. Otherwise, select **Show and use the default DOM Element mapping**.



Web Service

Web Service XML to Java Mappings

Review your Web service type mappings and make any necessary changes before proceeding to the next page.

XML Types:

http://www.w3.org/2001/XMLSchema:string
http://tempuri.org/ItsoInsuraTest/InsuraGroup/Insured.dadx/XSD:InsuredPoliciesResult

○ Show and use the default Java bean mapping

● Show and use the default DOM Element mapping

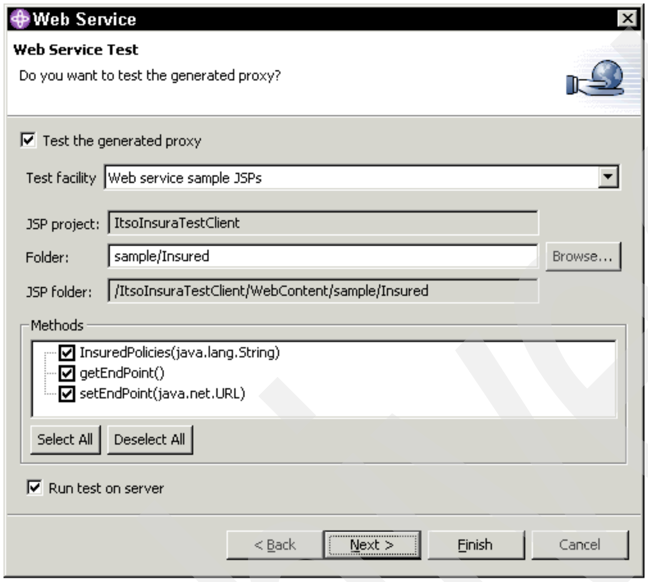
○ Edit and use a customized mapping

Class: org.w3c.dom.Element

Figure 12-53   Web Service wizard: XML to Java mappings

Skip the next panel; it only confirms the mapping you have selected.

On the next panel (Figure 12-54), the options for the test client are displayed. Accept the defaults.



Web Service

Web Service Test

Do you want to test the generated proxy?

☑ Test the generated proxy

Test facility: Web service sample JSPs

JSP project: ItsoInsuraTestClient

Folder: sample/Insured

JSP folder: /ItsoInsuraTestClient/WebContent/sample/Insured

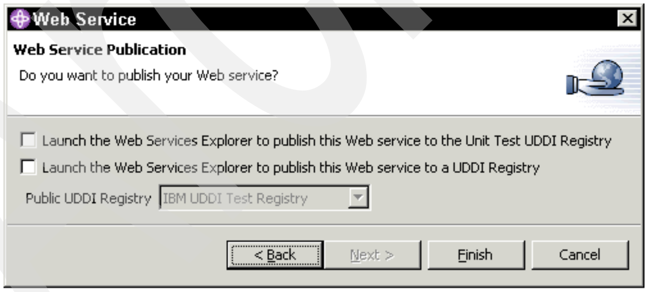
Methods

- ☑ InsuredPolicies(java.lang.String)
- ☑ getEndPoint()
- ☑ setEndPoint(java.net.URL)

☑ Run test on server

Figure 12-54 Web Service wizard: test client

On the final panel (Figure 12-55), you can choose to start the Web Services Explorer to publish the Web service to a UDDI registry.



Web Service

Web Service Publication

Do you want to publish your Web service?

☐ Launch the Web Services Explorer to publish this Web service to the Unit Test UDDI Registry

☐ Launch the Web Services Explorer to publish this Web service to a UDDI Registry

Public UDDI Registry: IBM UDDI Test Registry

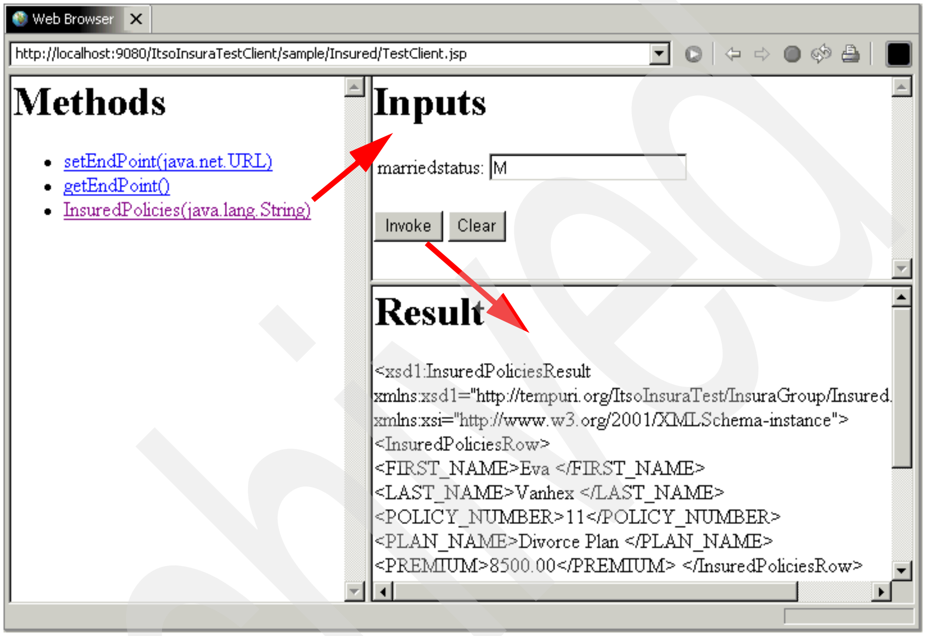
Figure 12-55 Web Service wizard: publish to UDDI

Click **Finish** and the Web service is started in the server. Additionally, a Web browser is opened with the test client.

Testing with the test client

In the test client, select the `InsuredPolicies` method, enter the `marriedstatus` value (`M` or `S`), click **Invoke** and the result XML is displayed.

You should see the XML stream returned by the Web service containing the data queried from the database (Figure 12-56).



http://localhost:9080/ItsoInsuraTestClient/sample/Insured/TestClient.jsp

Methods

- setEndPoint(java.net.URL)
- getEndPoint()
- InsuredPolicies(java.lang.String)

Inputs

marriedstatus: M

Invoke Clear

Result

```
<xsd1:InsuredPoliciesResult
xmlns:xsd1="http://tempuri.org/ItsoInsuraTest/InsuraGroup/Insured.
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<InsuredPoliciesRow>
<FIRST_NAME>Eva </FIRST_NAME>
<LAST_NAME>Vanhex </LAST_NAME>
<POLICY_NUMBER>11</POLICY_NUMBER>
<PLAN_NAME>Divorce Plan </PLAN_NAME>
<PREMIUM>8500.00</PREMIUM> </InsuredPoliciesRow>
```

Figure 12-56 Web service test client

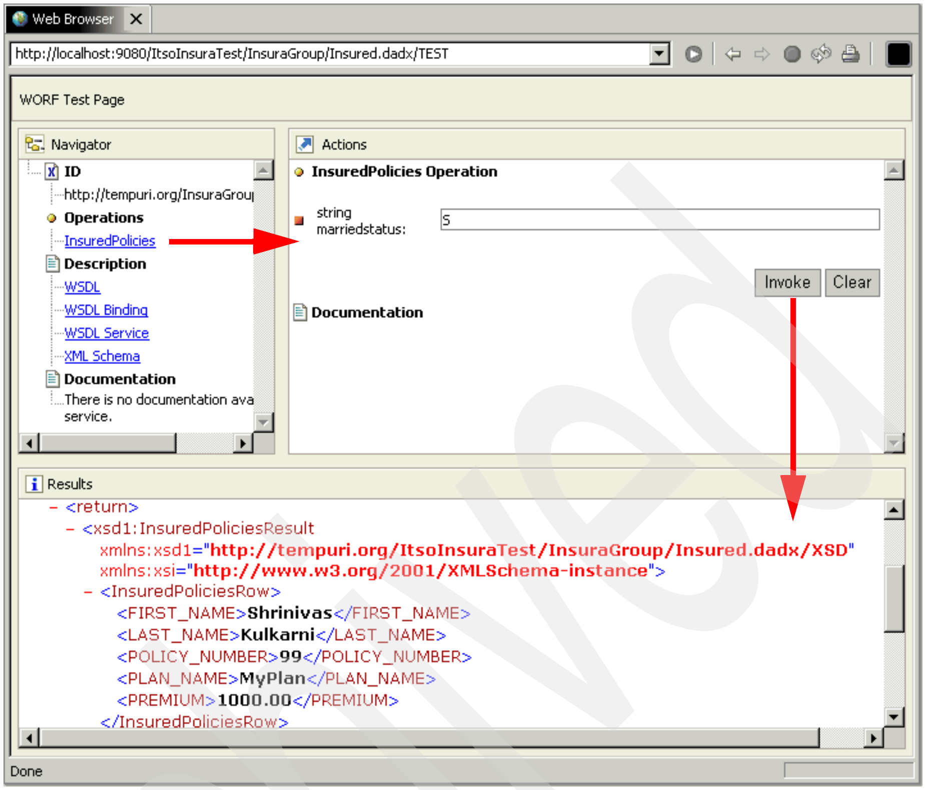
Testing with WORF

DB2 provides a Web Service Object Runtime Facility (WORF) for testing of DB2 Web services. To use WORF, enter this URL in the browser:

```
http://localhost:9080/ItsoInsuraTest/InsuraGroup/Insured.dadx/TEST
```

In the WORF test facility (Figure 12-57 on page 386), select the **InsuredPolicies** method, enter the `marriedstatus` value (`M` or `S`), click **Invoke** and the result XML is displayed.

Also click **WSDL** and **XML Schema** to see the generated WSDL and XML Schema files.



string marriedstatus: S

```
- <return>
  - <xsd1:InsuredPoliciesResult
      xmlns:xsd1="http://tempuri.org/ItsoInsuraTest/InsuraGroup/Insured.dadx/XSD"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    - <InsuredPoliciesRow>
        <FIRST_NAME>Shrinivas</FIRST_NAME>
        <LAST_NAME>Kulkarni</LAST_NAME>
        <POLICY_NUMBER>99</POLICY_NUMBER>
        <PLAN_NAME>MyPlan</PLAN_NAME>
        <PREMIUM>1000.00</PREMIUM>
      </InsuredPoliciesRow>
```

Figure 12-57 Web service testing with WORF

Generated files

Let us review all the files that are generated in the two projects.

Web service project: ItsoInsuraTest

In the project where the Web service is installed, we find:

- **WebContent/admin**—folder with a small administrative Web application to display the installed Web services. Select the index.html file and Run on Server to see the Web services.
- **WebContent/WEB-INF/isd**—folder with the deployment descriptor (Insured.isd) for this Web service. This file is then added to the dds.xml file (below).
- **WebContent/WEB-INF/lib**—folder with runtime JAR files.
- **WebContent/WEB-INF/web.xml**—Web deployment descriptor. If you open it, you will find that a servlet named InsureGroup has been added, pointing to com.ibm.etools.webservice.rt.dxx.servlet.DxxInvoker.

- **WebContent/worf**—folder with the WORF test facility (HTML and JSP).
- **WebContent/wsdl**—folder with the WSDL file InsuredService.wsdl.

Web service client project: ItsoInsuraTestClient

In the client project, we find:

- **JavaSource/org.tempuri.itsoinsuratest.insuragroup.insured.dadx.xsd**—a JavaBean that describes the result row.
- **JavaSource/proxy.soap**—Java package with the proxy class (`InsuredProxy`).
- **WebContent/sample/Insured**—folder with the test client JSPs (select the `TestClient.jsp` and **Run on Server** to launch the test client).
- **WebContent/WEB-INF/lib**—folder with client runtime JAR files.

WSDL file

Open the InsuredService.wsdl file and expand the service and port types to get a better understanding of the Web service operations (Figure 12-58 on page 388).

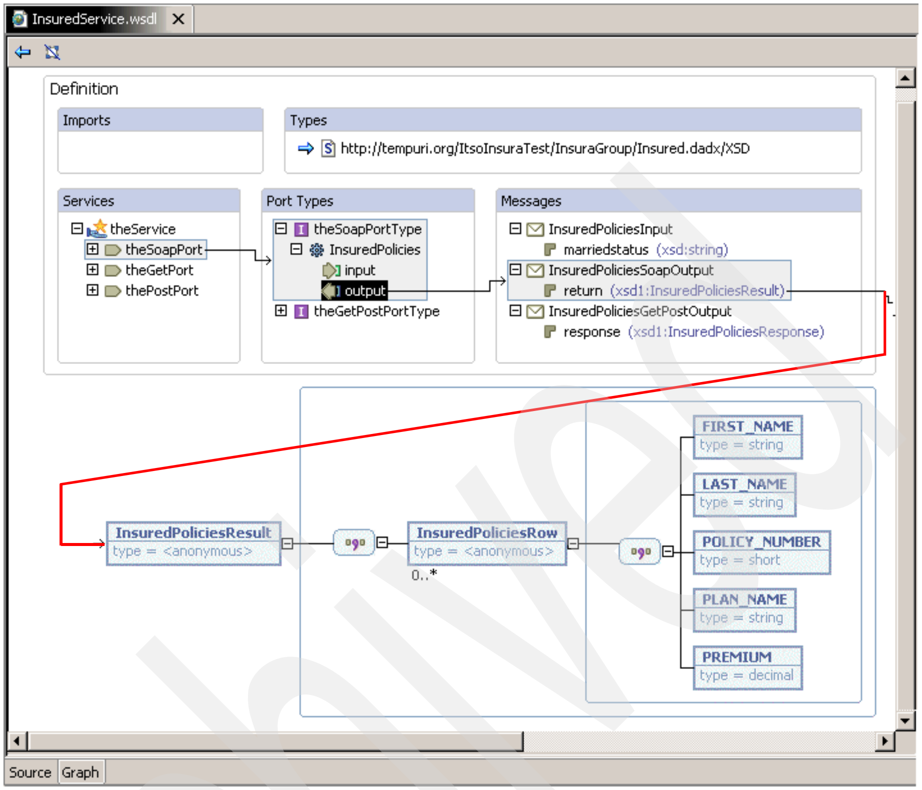


Figure 12-58 WSDL editor

Server

Open the Server perspective and explore the views:

- In the Navigator view, you can see a Servers project, where server definitions are stored.
- In the Server Configuration view, you find the WebSphere v5.0 Test Environment server that was defined automatically.
- In the Servers view, you can also see the server, with an indication that it is started.
- In the Console view, you can see the messages from starting the server, loading the `DefaultEAR` enterprise application, and executing the tests.

Remember that we specified a data source in the group configuration (Figure 12-45 on page 377). This data source was not used because a data source must be defined in the server.

Data sources are defined by opening the server configuration of the server, selecting a JDBC driver on the Data sources page, and then defining a data source.

Part 4

Worked examples

13

Worked scenario

This chapter describes a scenario featuring an hypothetical insurance company that is starting to use XML functionality in their day-to-day business. The following simple scenarios will be discussed:

- Providing stock quotes through a Web Service provider, and XML wrapper functionality.
- Storing XML documents, both intact and by shredding them into relational tables, using DB2 XML Extender features.
- Exploiting the information from the stored XML documents, intact or shredded, using XML Extender functionality (RDB_node and SQL statement DAD mappings) and via SQL/XML.
- Obtaining stock news information through Web Service requestor (UDF) functionality to assist in stock purchases.

13.1 Scenario introduction

In this chapter, we describe a hypothetical insurance company called ITSO Insurance, Inc. We do not attempt to describe the full business processes used by a real insurance company. All our scenarios are very simple, and their only purpose is to illustrate some of the ways in which you may incorporate XML documents in a database-oriented environment, using DB2, DB2 XML Extender and DB2 Information Integrator. For details on what versions of these products are required, and how to install them, see Appendix A, “Installation” on page 545.

ITSO Insurance, Inc. is a well-established insurance company that mainly works with their self-employed agents who work only for ITSO Insurance, Inc. A while ago, it was decided that ITSO Insurance, Inc. needed to expand its horizons, and should also try to sell insurance policies through insurance brokers.

As it turns out, insurance brokers are very much into XML. To standardize requests for quotes and providing policy information to insurance companies, they have agreed on a set of XML DTDs that all brokers use.

Since we already have an existing infrastructure to handle new insurance policies (remember that we normally use self-employed agents to sell our insurance policies), we want to keep that existing infrastructure in place, and reuse it for policies coming in through our new broker channel.

ITSO Insurance, Inc.’s existing infrastructure is based on a relational database model using DB2 to store and process all their data. Because the policies coming in through the broker channel are XML documents, we need to build some XML technology into our existing infrastructure to handle these new incoming policies, and build new means of communication with the broker channel to exchange requests (for example, stock quotes and broker reports) as XML documents.

In the scenario, we have the following players:

- **ITSO Insurance, Inc.**

  This is the company we just described and it has to implement the additional processes described hereafter to be able to handle the new broker channel.

- **Insurance agents**

  These are self-employed people who work exclusively for ITSO Insurance, Inc. So far, ITSO Insurance, Inc. has only worked with these agents. A wide range of applications have been developed to provide these insurance agents with the necessary tools to do a good job of selling insurance policies, as well as handling claims for their customers. The insurance agents do not play a significant role in these scenarios, although they have been the main driver behind the existing application and data infrastructure that we want to reuse as much as possible.

- **Insurance brokers**

  Customers go to insurance brokers to ask for quotes for certain types of policies and conditions. Unlike insurance agents, insurance brokers are not tied to any specific company. Insurance brokers can go and find the best deal of any company. We want insurance brokers to find their way to ITSO Insurance, Inc. by providing them with state of the art XML functionality to handle their requests.

- **Stock brokers**

  The insurance company re-invests the premiums into long-term investments. To get the best deal, the insurance company gathers information about its options from different stock brokers in order to select the best solution. To be able to make wise investment decisions, ITSO Insurance, Inc. needs to be able to obtain up-to-date information about companies from a reliable source.

The players in the scenario are shown in Figure 13-1.

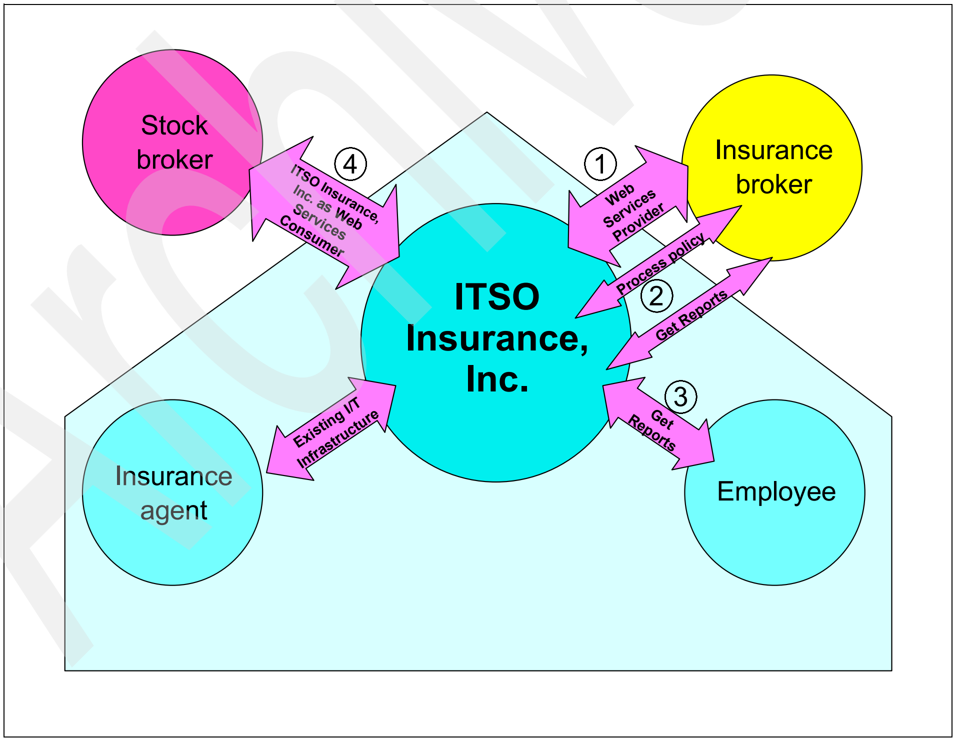


Figure 13-1 Overview of the players and processes

Next, we briefly describe the scenarios that we implemented. We discuss them in much more detail later on.

13.2 Scenario 1: Insurance quotes

The insurance broker is one of many brokers who go to the insurance company for price quotes. Price quotes used to be formerly provided by a person over the phone, but now the information is provided through a Web Service. In this case, our insurance company ITSO Insurance, Inc. is the Web service provider. They use the Web Object Runtime Framework (WORF) to provide the service. The request of the broker specifies such things as the name of the potential customer, insurance period, or other information that they are asked to provide as input to the Web service. To calculate a premium, we use the customer's insurance history information (the number and amount of previous claims). After the premium has been calculated, we return the information in the reply of the Web service to the broker.

This scenario is made up of two parts:

- Using the WORF to provide a Web service. For more details on the actual implementation of this part of the scenario, see 16.2, "Insurance application as Web service provider" on page 479.
- Using an XML wrapper to process the customer's insurance history information. For more details on the actual implementation of this part of the scenario, see 14.2.2, "The XML wrapper" on page 403.

This scenario is also shown in Figure 13-2 on page 397.

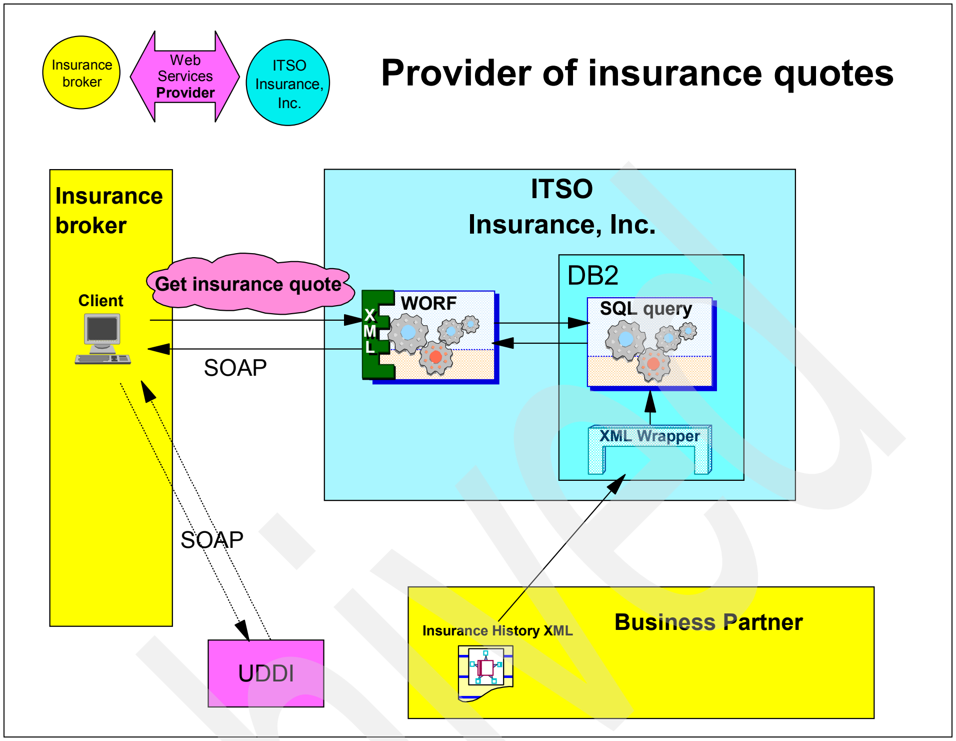


Figure 13-2 Getting the insurance quote

13.3 Scenario 2: Processing the insurance policy

When the insurance broker accepts our quote, we (ITSO Insurance, Inc.) need to start processing the information. The broker uses an industry-wide standard file format for insurance contracts based on XML to submit the accepted deal.

When the insurance broker submits the accepted deal, the XML document is shredded into relational tables (in an XML collection), so that existing (legacy) processes and applications, which are currently used to process policies coming in from agents, can pick up the new policies coming through the broker channel.

However, ITSO Insurance, Inc. has not fully implemented the data model that is used to exchange the data (XML schema) in their legacy applications. Since ITSO Insurance, Inc. does not use the full model yet, some information that is in the XML document cannot be represented in ITSO Insurance, Inc.'s existing relational model. Therefore, we also store the entire document as an XML column.

This is shown in Figure 13-3.

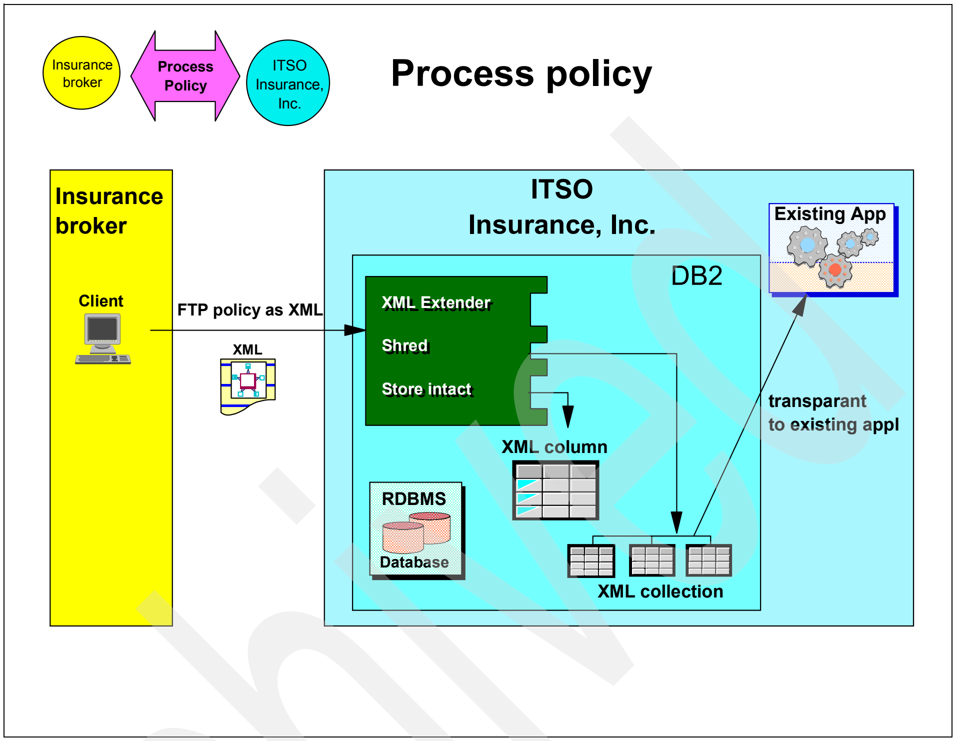


Figure 13-3 Processing incoming policies

A detailed description of the implementation can be found in 14.3, “Scenario 2: storing insurance policies” on page 416.

13.4 Scenario 3: Generating XML documents

The information that has been stored in XML columns (stored intact), and/or shredded into a set of relational tables (XML collection), can also be used as a source to generate XML document from.
As mentioned in the introduction, the brokers are rather fond of XML and like to get reports on their sales (broker sales report) in XML format.
In addition, many of our corporate customers like to receive reports on the policies they have with us (ITSO Insurance, Inc.), for example for their employees who have a health plan with us, which the employees pay for. These reports are produced on demand in XML format and forwarded to the corporate customers.

In this scenario, we demonstrate how to generate XML documents from:

- Information stored in the XML column's side tables using SQL/XML
- Information stored in the XML collection using:
  - XML Extender's RDB_node and SQL statement mapping
  - SQL/XML

This scenario is shown graphically in Figure 13-4, and described in more detail in 14.4, "Scenario 3: composing XML documents" on page 432.

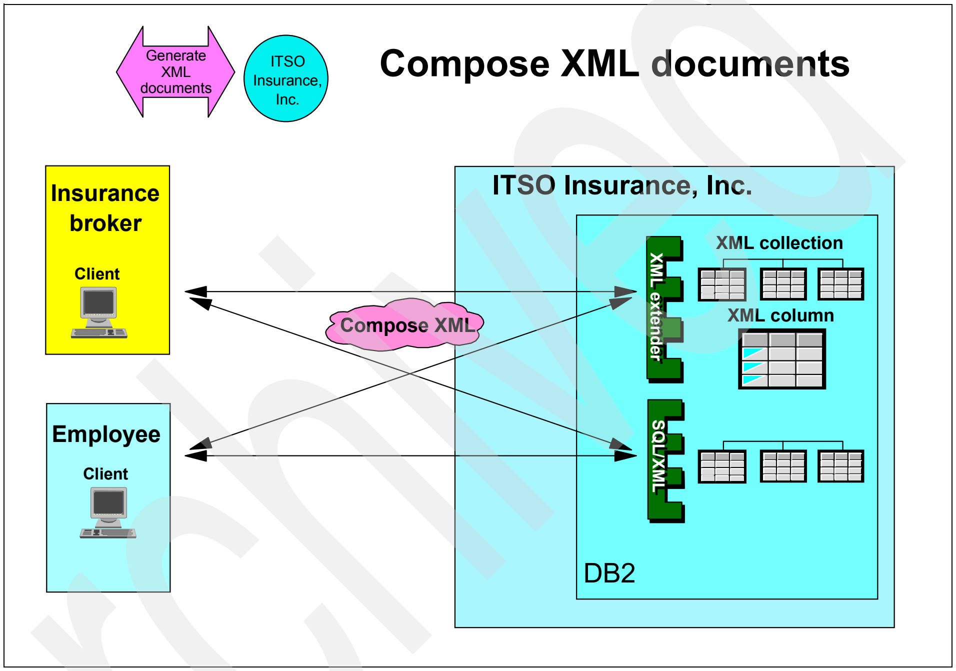


Figure 13-4 Generating XML documents

13.5 Scenario 4: Gathering information for investments

An insurance company has to invest the premiums that it receives and put them into long-term investments. ITSO Insurance, Inc. uses the stock market to do this (probably bonds). To get the best deal, the insurance company asks for bids.

To ask for bids, the insurance company uses Web services again. However, in this case, the insurance company is the Web service consumer. To implement this, we use the DB2 Web service consumer UDFs, as shown in Figure 13-5 on page 400.

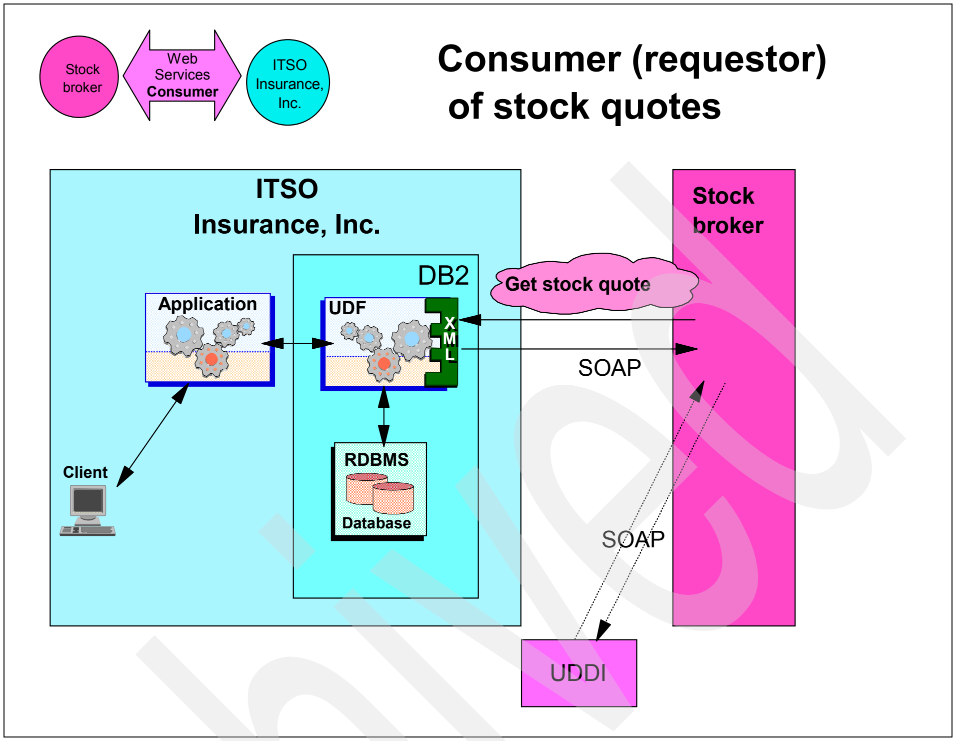


Figure 13-5 ITSO Insurance, Inc. as a Web service consumer

A detailed description of how to implement this scenario using "native" DB2 services can be found in 14.5, "Scenario 4: Web service requestor" on page 449. A complete implementation using WebSphere Studio Application Developer functionality can be found in 16.5, "Insurance application as a Web service requestor" on page 518.

14

Scenario implementation using DB2 functionality

This chapter shows you how to implement the scenarios outlined in the previous chapter using standard DB2 SQL queries and tools like the DB2 Control Center.

First, we describe how to install and set up the scenario database in order for the scenario to be implemented.

Topics in this chapter include:

14.1 Setting up the system for the scenario

The files that make up the scenario are made available for download from the ITSO Web site. For instructions, see Appendix F, “Additional material” on page 655. The scenario files should be unzipped to a directory called:

```
C:\SG246994\Scenario
```

Most .bat files take a database name (and sometimes additional parameters) as input. The database name defaults to `INSURA`.

Note: All *.DAD files reference DAD.DTD as follows:

```
<!DOCTYPE DAD SYSTEM "C:\sg246994\Scenario\dad.dtd">
```

That is, we copied the dad.dtd file that comes with XML Extender to our `\Scenario` directory to make the scenario self-contained. You can change the reference to the actual location where you installed the DB2 XML Extender code, normally:

```
[DB2Installdir]\samples\db2xml\dtd\dad.dtd
```

Setting up the INSURA database

By default all the scenario objects are created in a DB2 database named INSURA. Most .bat files accept other database names as input parameters in case you want to use a different name or create an additional database.

- Start a DB2 command window.
- Change the directory to the one where the scenario samples are located:

  ```
  CD C:\SG246994\Scenario
  ```

- Run **`SetupDB.bat (or SetupDB databasename)`**. This command creates the database, enables it for XML Extender, sets up the tables, and loads sample data into some of the tables.
- Run **`SetupXMLColumn.bat`**. This command file enables a column as an XML column (Policy_Doc in the Policy_Docs table).
- Run **`SetupXMLWrapper.bat`**. This command file creates XML wrapper nicknames for Customers, Insurances, and Claims. (Note that in order to be able to use the XML Wrapper, you must have the DB2 Information Integrator product installed. See, “DB2 Information Integrator” on page 547 for details on how to install DB2 Information Integrator.)
- Run **`SetupUDF.bat`** to set up the soap UDFs for the Web Service requestor functions and Headlines table UDF.
- The directory also contains a file called scenario-readme.txt. It contains the latest information on how to successfully set up, and run the scenario.

14.2 Scenario 1: Web service provider

In this first scenario, we provide a simple business case in which it makes sense to use Web services. Then, we provide details showing how to implement such a scenario.

14.2.1 Overview

When a customer goes to an insurance broker, the broker company will go out and look for the best fit for the customer's need that exists in the market. To facilitate the information gathering process, insurance brokers make use of Web services. They invoke Web services provided by different insurance companies.

For this scenario we use a fictitious insurance company named ITSO Insurance, Inc. The first thing ITSO Insurance, Inc. must do to be able to convince insurance brokers to recommend its insurance policies to customers is to provide them with an insurance policy quote system for certain types of policies. This could be delivered using a Web service. We do this by using the Web Services Object Runtime Framework (WORF).

To determine the premium to quote for the policy, we look up the potential customer's insurance history and analyze his previous claims. The more claims, the higher the proposed premium will be. To analyze the insurance history, we use the XML wrapper (as the insurance history is provided as an XML file).

Therefore this scenario is composed of two parts:

- Setting up a Web service using WORF. More details on how to set up and use WORF is discussed in 10.2.2, "Installing WORF on IBM WebSphere Application Server" on page 279.
- Analyzing the insurance history file using the XML wrapper

14.2.2 The XML wrapper

In this section we focus on the XML wrapper aspects of the scenario. A description of a sample implementation for our scenario using WebSphere Studio Application Developer can be found in 16.2, "Insurance application as Web service provider" on page 479.

As mentioned previously, using XML wrapper requires that you have DB2 Information Integrator installed. For details, see Appendix A, "Installation" on page 545.

14.2.3 Context

ITSO Insurance, Inc. uses the services of a business partner that keeps track of every person's insurance history across all the insurance companies. This history includes details such as the number of claims the person has had so far and the amount of each claim. Based on the customer's identity, ITSO Insurance, Inc. queries the business partner for the insurance history of this potential customer. The insurance history is provided as an XML file available through the Web as a URI; however, to keep it simple, we use a file in the local file system from here on. The algorithm used to calculate the quote for a given customer takes into account the total amount of money claimed so far using this simple formula:

```
regular premium + 4% of claims
```

For simplicity, we assume that all insurance requests are initiated by brokers. The IT system for ITSO Insurance, Inc. does not keep track of brokers. The broker details are sent to Web services as part of the XML request document.

14.2.4 XML wrapper implementation

To enable brokers to request motor vehicle insurance quotes from us, we use a Web service. The Web service takes the request, looks up the insurance history of the person involved, calculates the quote, and returns a reply. The Web services part of the story is described in more detail in Chapter 16, “DB2 Web services and XML with Application Developer” on page 475. In this section, we focus on how to retrieve the information from the insurance history (XML file) provided by our business partner by using the XML wrapper.

InsuranceHistory.xml document layout

Example 14-1 shows the layout of the insurance history document.

Example 14-1 Layout of the insurance history XML document

```
<InsuranceHistory>
   <Customer>
      <MotorVehicleInsurance>
            <claim>
            </claim>
            ...
      </MotorVehicleInsurance>
      ...
   </Customer>
   ...
</InsuranceHistory>
```

Creating nicknames for the XML document

In the \Scenario directory, there are two sets of definitions for the XML wrapper. They provide equal functionality as far as generating the policy quote is concerned. The latter maps all elements and attributes of the XML insurance history file, whereas the former only maps those elements that are required to be able to return the insurance quote.

- The SetupXMLWrapper.bat file creates:
  - Wrapper definition for the XML wrapper
  - Server definition for the XML wrapper
  - Three nicknames to map our insurance history document:
    - Customers
    - Insurances
    - Claims
- The SetupXMLWrappercs.bat file creates:
  - Wrapper definition for the XML wrapper
  - Server definition for the XML wrapper
  - Three nicknames to map our insurance history document:
    - Customers
    - Insurances
    - Claims
  - Three views of these nicknames:
    - Customers_V
    - Insurances_V
    - Claims_V

These definitions were generated by the DB2 Control Center. In the proceeding pages, we show how to generate the nicknames using the DB2 Control Center. (For more information on XML wrapper, see Chapter 11, “XML wrapper” on page 305.)

Automatic nickname discovery by the DB2 Control Center

After creating a wrapper for the XML wrapper, as well as a server definition, you can use the DB2 Control Center to discover the structure of the XML file that you are trying to map. This is shown on the next set of figures and the procedure below.

1. To start the discovery process, right-click the server name (**XML_SERVER**) that you defined for the XML wrapper (MY_XML), as shown in Figure 14-1.

   This takes you to Figure 14-2 on page 407.

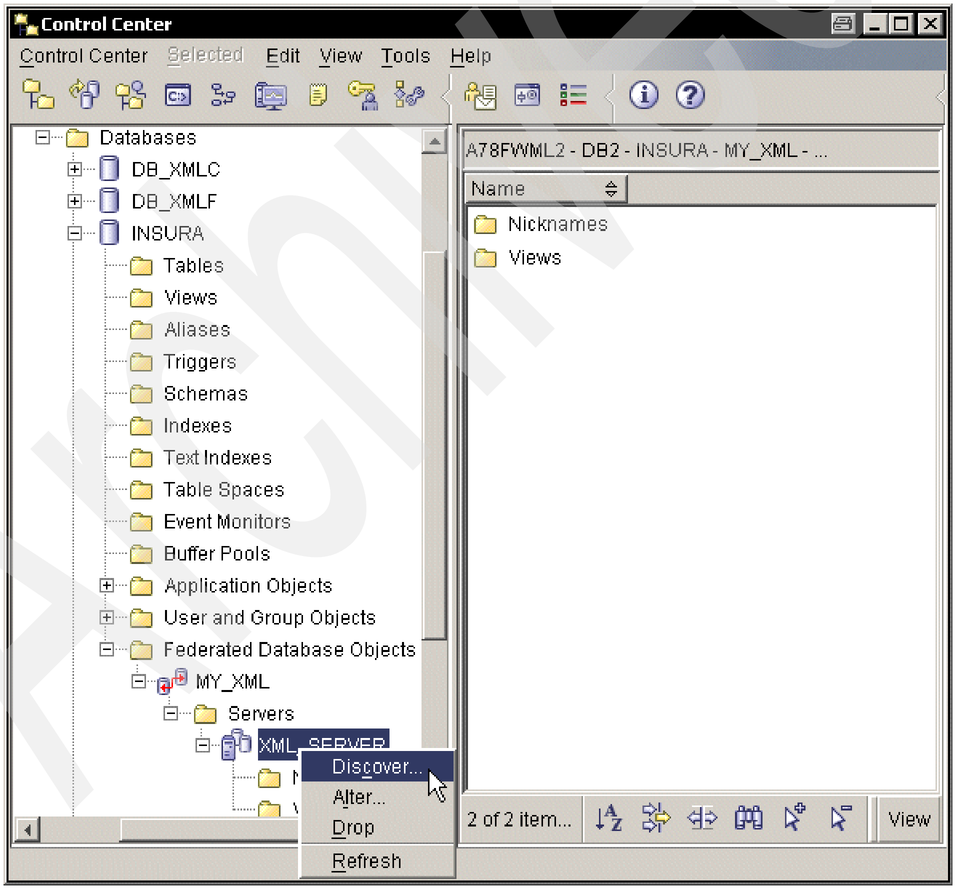
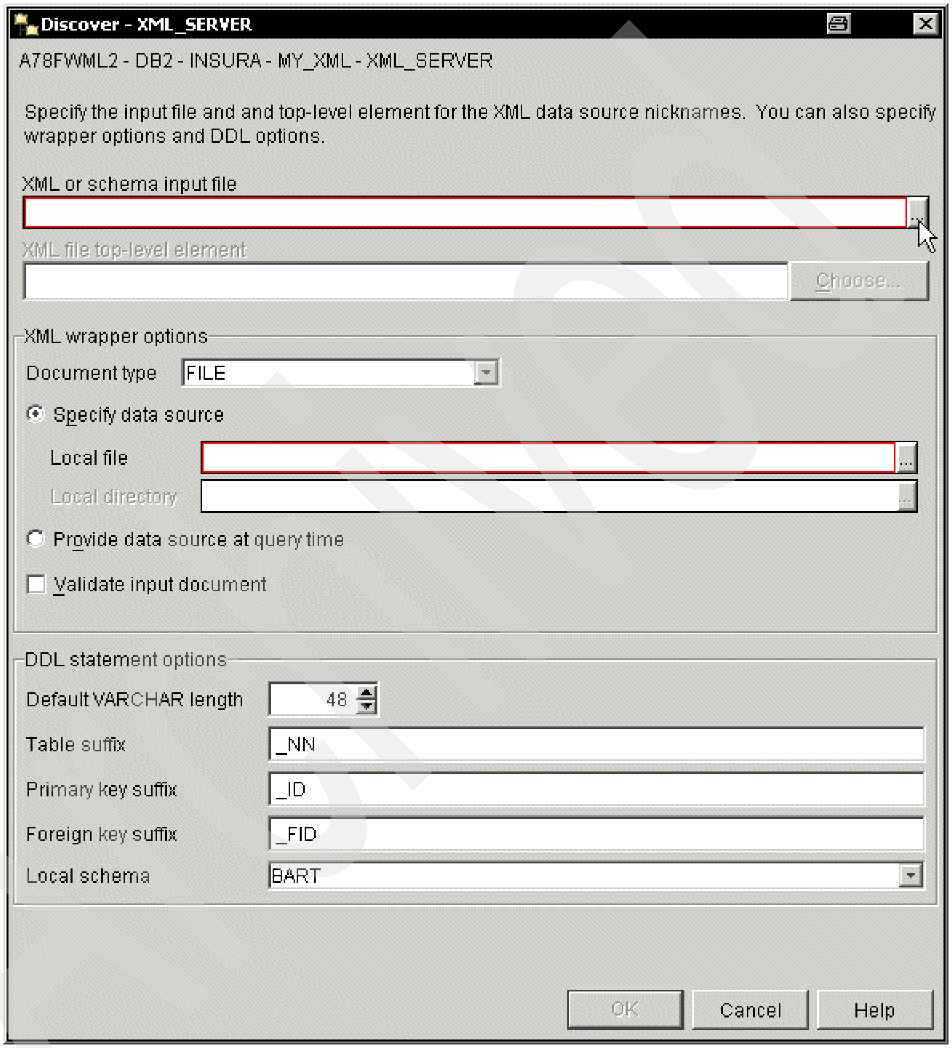


Figure 14-1 Starting the discovery process

2. In this window, you can indicate where to look for the mapping of the XML file. It can either be an XML Schema or an XML file. Either fill in the full file path and file name, or click the dots (**...**) to the right of the field called XML or schema input file as shown in Figure 14-2.



Discover - XML_SERVER

A78FWML2 - DB2 - INSURA - MY_XML - XML_SERVER

Specify the input file and and top-level element for the XML data source nicknames. You can also specify wrapper options and DDL options.

XML or schema input file: ...

XML file top-level element: Choose...

XML wrapper options

Document type: FILE

● Specify data source

Local file: ...

Local directory: ...

○ Provide data source at query time

□ Validate input document

DDL statement options

Default VARCHAR length: 48

Table suffix: _NN

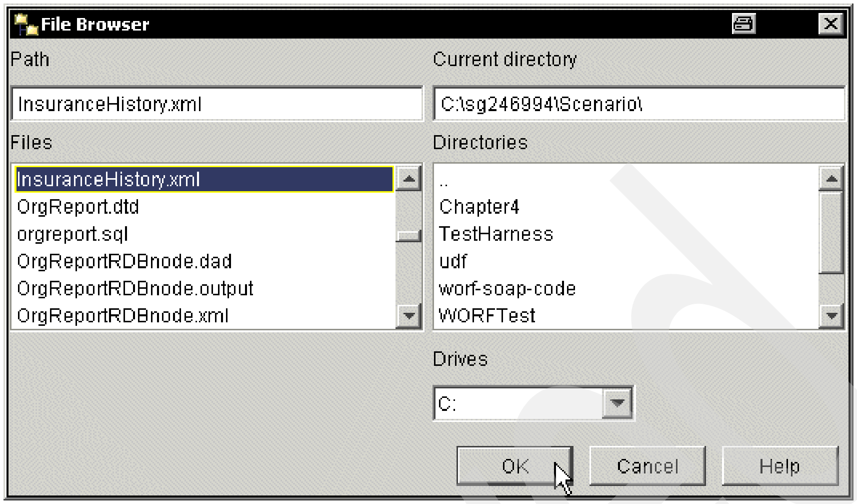
Primary key suffix: _ID

Foreign key suffix: _FID

Local schema: BART

Figure 14-2 XML nickname discovery

This takes you to Figure 14-3 on page 408, where we select the InsuranceHistory.xml file.



Path: InsuranceHistory.xml

Current directory: C:\sg246994\Scenario\

Files:
InsuranceHistory.xml
OrgReport.dtd
orgreport.sql
OrgReportRDBnode.dad
OrgReportRDBnode.output
OrgReportRDBnode.xml

Directories:
..
Chapter4
TestHarness
udf
worf-soap-code
WORFTest

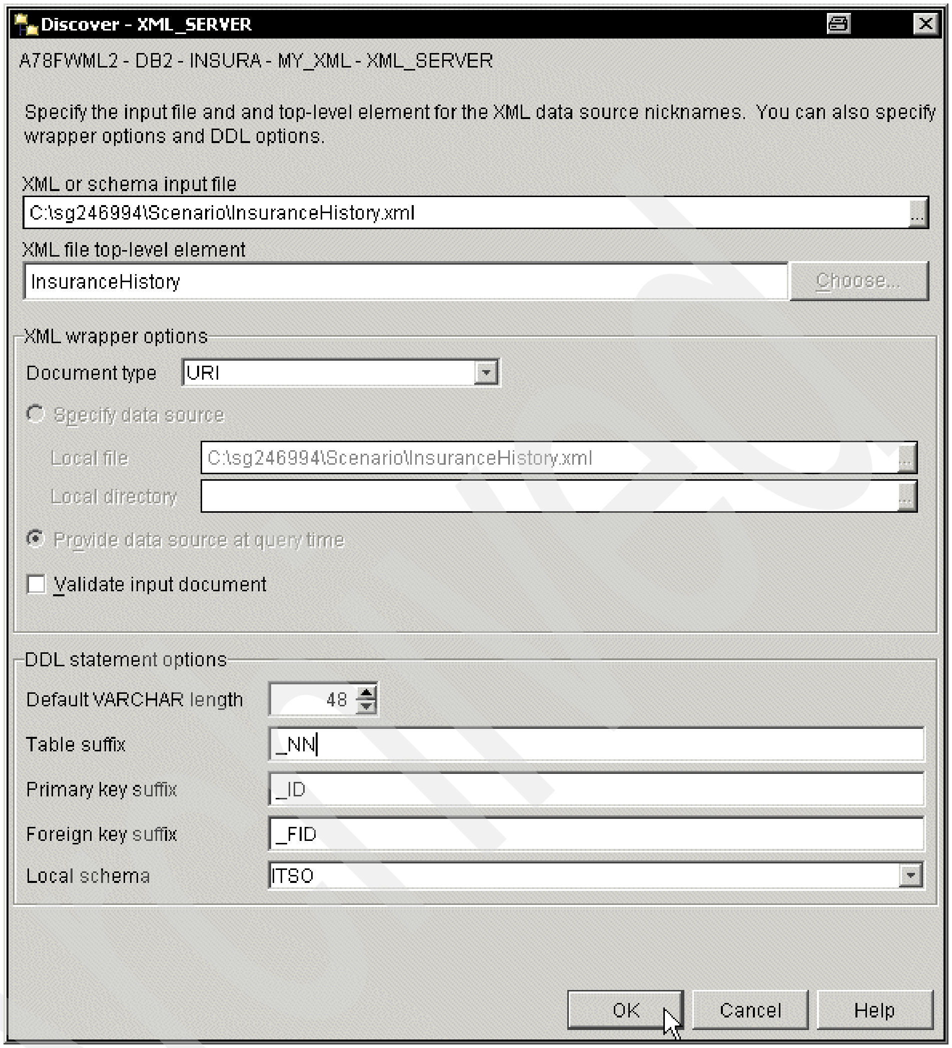
Drives: C:

Figure 14-3 Input file selection

3. Click **OK** when done. This takes us back to the Discover XML_Server window. In this window, the file name and the top-level element name (root) are already filled in by the Control Center, as `InsuranceHistory`, as shown in Figure 14-4 on page 409.

> **Note:** Note that we select the “URI” wrapper option. This allows us to specify a URI at execution time of the queries against the XML wrapper nicknames. We use this option as it provides us with the flexibility of changing the location of the InsuranceHistory.xml file at execution time. In our case the file is a local file in the file system, but in real life this file is likely to be maintained by an external company, and we can reach the file through a Web page that we specify in the URI.

4. We also use ITSO as the schema name to make sure the nicknames are created in the same schema as the other tables in our scenario. We do not change the other DDL options at this time. We will change them later.


Discover - XML_SERVER

A78FWML2 - DB2 - INSURA - MY_XML - XML_SERVER

Specify the input file and and top-level element for the XML data source nicknames. You can also specify wrapper options and DDL options.

XML or schema input file: C:\sg246994\Scenario\InsuranceHistory.xml

XML file top-level element: InsuranceHistory

XML wrapper options

Document type: URI

○ Specify data source

Local file: C:\sg246994\Scenario\InsuranceHistory.xml

Local directory:

● Provide data source at query time

□ Validate input document

DDL statement options

Default VARCHAR length: 48

Table suffix: _NN

Primary key suffix: _ID

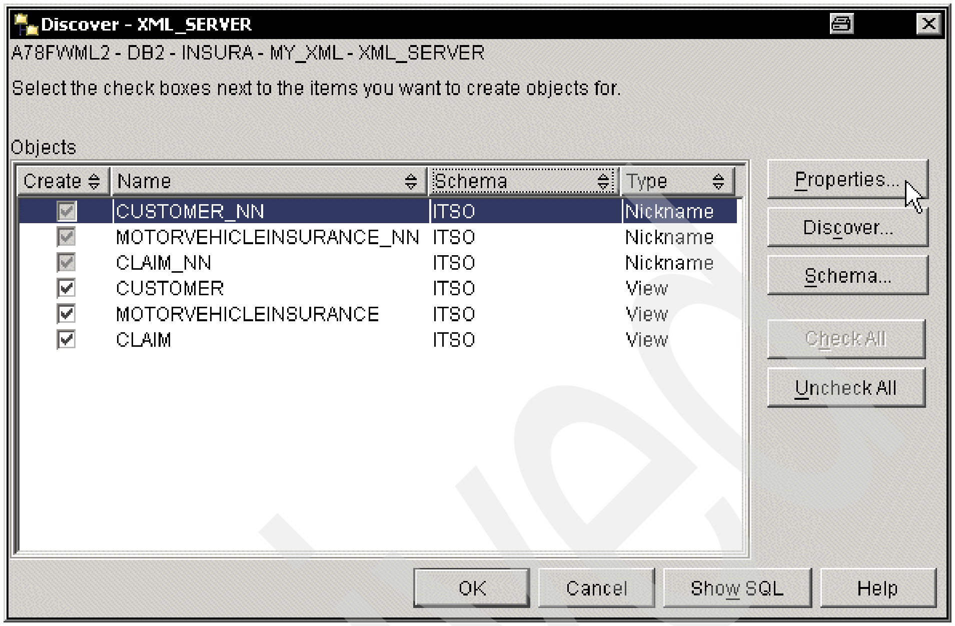
Foreign key suffix: _FID

Local schema: ITSO

Figure 14-4 Discover XML_Server window

5. Click **OK** when ready.

   This takes us to the window in Figure 14-5 on page 410. The DB2 Control Center discovered three hierarchy levels and therefore proposes to create three nicknames, CUSTOMER_NN, MOTORVEHICLEINSURANCE_NN and CLAIM_NN. The tool also suggests that we create three views to access the information in each hierarchical level.



A78FWML2 - DB2 - INSURA - MY_XML - XML_SERVER

Select the check boxes next to the items you want to create objects for.

Objects

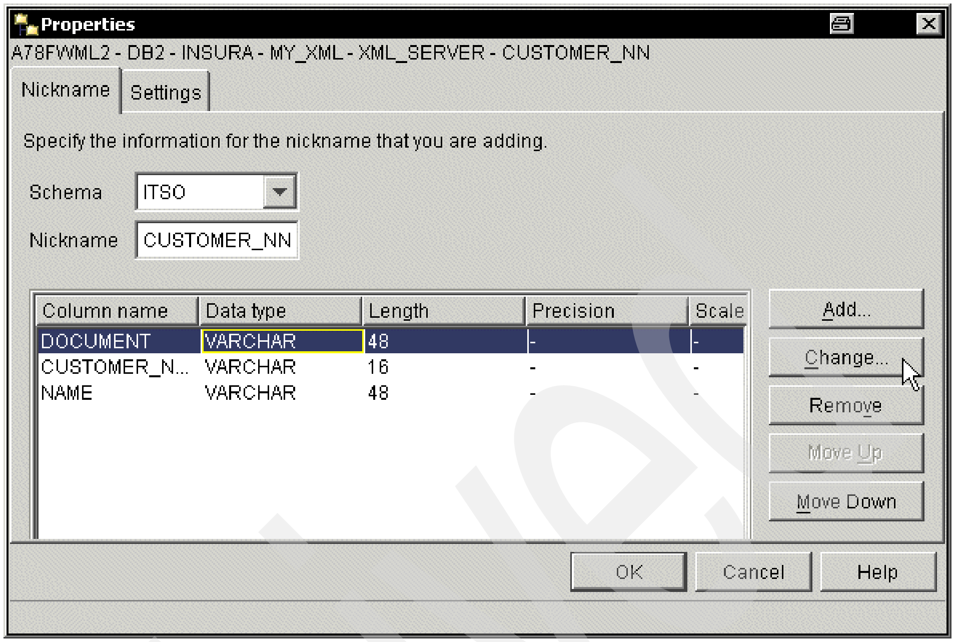
Create	Name	Schema	Type
☑	CUSTOMER_NN	ITSO	Nickname
☑	MOTORVEHICLEINSURANCE_NN	ITSO	Nickname
☑	CLAIM_NN	ITSO	Nickname
☑	CUSTOMER	ITSO	View
☑	MOTORVEHICLEINSURANCE	ITSO	View
☑	CLAIM	ITSO	View

Figure 14-5 Generated nicknames and views

6. Because we want to change some of the attributes of the nickname, like the column definition, we click the **Properties** button.

   This takes us to Figure 14-6 on page 411.

7. Select the **DOCUMENT** column (this is the default column name generated to specify the URI at execution time), and click the **Change** button.



A78FWML2 - DB2 - INSURA - MY_XML - XML_SERVER - CUSTOMER_NN

Specify the information for the nickname that you are adding.

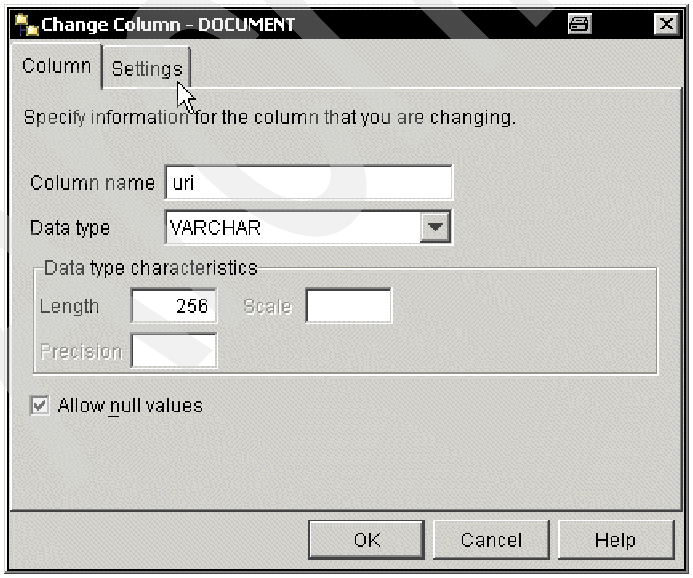
Schema: ITSO

Nickname: CUSTOMER_NN

Column name	Data type	Length	Precision	Scale
DOCUMENT	VARCHAR	48	-	-
CUSTOMER_N...	VARCHAR	16	-	-
NAME	VARCHAR	48	-	-

Figure 14-6 Change properties

This takes you to a window shown in Figure 14-7 below.



Specify information for the column that you are changing.

Column name: uri

Data type: VARCHAR

Data type characteristics

Length: 256

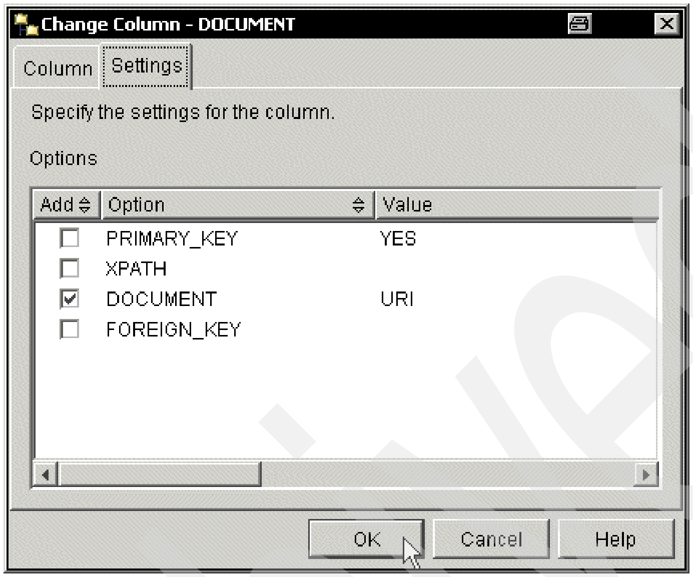
Scale:

Precision:

☑ Allow null values

Figure 14-7 Change column settings

8. Change the column name to URI and the length to 256. Click the **Settings** tab to verify that the field actually is the URI field we want as shown in Figure 14-8.



Specify the settings for the column.

Options

Add	Option	Value
□	PRIMARY_KEY	YES
□	XPATH	
☑	DOCUMENT	URI
□	FOREIGN_KEY	

Figure 14-8 Change column settings -2

9. Click **OK** when you are done.

   This takes you back to the window shown in Figure 14-6 on page 411,

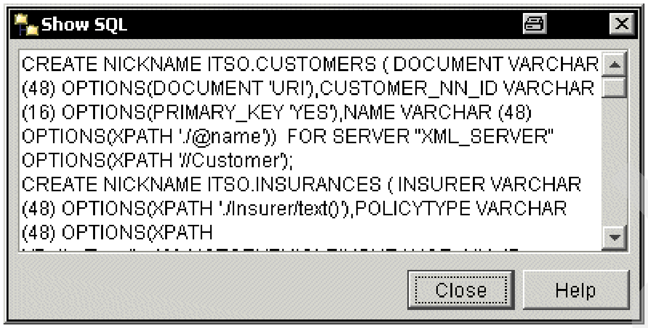
10. Repeat the previous steps for any other columns in the nickname that need to be changed.

11. After you set up your nicknames, click **OK**.

    This returns you to the window shown in Figure 14-5 on page 410.

12. Then make changes to all other nicknames and views. When done, click the **SHOW SQL** button. Or you can click the **OK** button to generate the SQL statements.

    This takes you to the window shown in Figure 14-9 on page 413.



```
CREATE NICKNAME ITSO.CUSTOMERS ( DOCUMENT VARCHAR
(48) OPTIONS(DOCUMENT 'URI'),CUSTOMER_NN_ID VARCHAR
(16) OPTIONS(PRIMARY_KEY 'YES'),NAME VARCHAR (48)
OPTIONS(XPATH './@name'))  FOR SERVER "XML_SERVER"
OPTIONS(XPATH '//Customer');
CREATE NICKNAME ITSO.INSURANCES ( INSURER VARCHAR
(48) OPTIONS(XPATH './Insurer/text()'),POLICYTYPE VARCHAR
(48) OPTIONS(XPATH
```

Figure 14-9 Show SQL window

13. You can select all the generated SQL by clicking inside the Show SQL window, pressing **Ctrl+A**. To select the entire content, press **Ctrl+C** to copy the SQL statements and then paste them into an editor, Notepad, for example.

14. Edit the SQL statements as needed. You probably will want to make some additional changes if you want to generate the views as well. Changes you make to column names in the nickname are not reflected in the CREATE VIEW statement. In Example 14-2, we added the URI column to the INSURANCES_V and CLAIMS_V view, as this is not automatically generated by the Control Center, and you cannot use the views without the URI column reference. Lastly, the Customer.Name column was added to the default CLAIMS_V view, to make the execution of our proposed premium SQL statement easier to code. After the necessary editing, the SQL statements resemble the code in Example 14-2.

Example 14-2 CreateWrapperDefsCS.sql

```
CREATE NICKNAME ITSO.CUSTOMERS
  ( URI VARCHAR (256) OPTIONS(DOCUMENT 'URI'),
    CUSTOMER_ID VARCHAR (16) OPTIONS(PRIMARY_KEY 'YES'),
    NAME VARCHAR (50) OPTIONS(XPATH './@name'))
  FOR SERVER "XML_SERVER"  OPTIONS(XPATH '//Customer');

CREATE NICKNAME ITSO.INSURANCES
 ( INSURER VARCHAR (50) OPTIONS(XPATH './Insurer/text()'),
   POLICYTYPE VARCHAR (48) OPTIONS(XPATH './PolicyType/text()'),
   INSURANCE_ID VARCHAR (16) OPTIONS(PRIMARY_KEY 'YES'),
   PREMIUMPAYMENTLAPSES VARCHAR (48)
                     OPTIONS(XPATH './PremiumPaymentLapses/text()'),
   START_DATE VARCHAR (10) OPTIONS(XPATH './@startDate'),
   END_DATE VARCHAR (10) OPTIONS(XPATH './@endDate'),
   POLICYCLOSURE_REASON VARCHAR (100)
```

```
                        OPTIONS(XPATH './PolicyClosure/Reason/text()'),
   POLICYCLOSURE_INITIATEDBY VARCHAR (48)
                             OPTIONS(XPATH './PolicyClosure/@initiatedBy'),
   CUSTOMER_FID VARCHAR (16) OPTIONS(FOREIGN_KEY 'CUSTOMERS'))
  FOR SERVER "XML_SERVER"  OPTIONS(XPATH './MotorVehicleInsurance');

CREATE NICKNAME ITSO.CLAIMS
 ( DESCRIPTION VARCHAR (256) OPTIONS(XPATH './Description/text()'),
   AMOUNT VARCHAR (10) OPTIONS(XPATH './AmountClaimed/text()'),
   DATE VARCHAR (10) OPTIONS(XPATH './@claimDate'),
   INSURANCE_FID VARCHAR (16) OPTIONS(FOREIGN_KEY 'INSURANCES'))
 FOR SERVER "XML_SERVER"  OPTIONS(XPATH './Claim');

CREATE VIEW ITSO.CUSTOMERS_V
   AS SELECT Customer.Uri, Customer.Customer_ID, Customer.name
   FROM   ITSO.Customers Customer;

CREATE VIEW ITSO.INSURANCES_V
   AS SELECT Insurance.Insurer, Insurance.PolicyType,
               Insurance.Insurance_ID, Insurance.PremiumPaymentLapses,
               Insurance.start_Date, Insurance.end_Date,
               Insurance.PolicyClosure_Reason,
               Insurance.PolicyClosure_initiatedBy,
               Customer.Customer_ID , Customer.URI
   FROM   ITSO.Insurances Insurance, ITSO.Customers Customer
   WHERE  Customer.Customer_ID = Insurance.Customer_FID;

CREATE VIEW ITSO.CLAIMS_V
   AS SELECT Claim.Description, Claim.Amount, Claim.Date,
             Insurance.Insurance_ID,Customer.URI,Customer.Name
   FROM   ITSO.Claims Claim, ITSO.Insurances Insurance,
          ITSO.Customers Customer
   WHERE  Insurance.Insurance_ID = Claim.Insurance_FID
     AND  Customer.Customer_ID = Insurance.Customer_FID;
```

Using the XML nicknames

As mentioned before, we use the insurance history information to determine a proposed insurance premium for motor vehicle insurance policies. The algorithm to calculate the quote for a given customer takes into account the total amount of money claimed so far, using this simple formula:

```
regular premium + 4% of claims
```

The regular (base) premium is stored in the Premium column of the P_TYPE table for Motor insurance. To determine the additional premium, we look up the person in the insurance history XML file, sum all this person’s previous claims, and take 4% to be added to his base premium.

To get to the claims information in the insurance history XML file, we need to join the claims nickname with the higher levels in the hierarchy (Insurances and Customers) using their PK-FK relationships. We SUM all the amount(s) for the different claims we find. Note that we use the COALESCE built-in function. This is to make sure the query adds 0 additional premium in case the person does not have an insurance history. This query is shown in Example 14-3.

Example 14-3 Sample query using XML wrapper

```
-- Do a premium calculation for a Motor insurance quote
--  add 4% of past motor vehicle claims to the premium

Select p.Premium + cs.additional as proposed_premium
  from  ITSO.P_TYPE  p,
      (select coalesce(sum(cast(cl.amount AS DECIMAL(10,2)))*0.04,0)
                AS additional
        from ITSO.Customers c, ITSO.Insurances i, ITSO.claims cl
           where uri = 'file:C:\SG246994\Scenario\InsuranceHistory.xml '
             and c.customer_id = i.customer_fid
             and i.insurance_id = cl.insurance_fid
             and rtrim(c.name) = rtrim('Shrinivas')||' '||rtrim('Kulkarni')

      ) AS  cs
  where rtrim(p.plan_name) = rtrim('Motor')
;
```

This query is also provided with the scenario as:

```
C:\sg246994\Scenario>GetProposedPremiumWrapper
```

We can also use the CLAIMS_V view that we created to return this information. Using this view makes the query a lot easier to code as shown in Example 14-4.

Example 14-4 Query using CLAIMS_V view

```
Select p.Premium + cs.additional as proposed_premium
  from  ITSO.P_TYPE  p,
       (select coalesce(sum(cast(cl.amount AS DECIMAL(10,2))) * 0.04 , 0)
               AS additional
          from ITSO.claims_v cl
          where cl.uri = 'file:C:\SG246994\Scenario\InsuranceHistory.xml '
            and rtrim(cl.name) = rtrim('Shrinivas')||' '||rtrim('Kulkarni')
       ) AS  cs
```

```
    where rtrim(p.plan_name) = rtrim('Motor')
;
```

The result of both queries should be:

```
PROPOSED_PREMIUM
---------------------------------
                        556.3360
```

14.3 Scenario 2: storing insurance policies

In this scenario, we look at how to process the insurance policies that come in as XML documents from insurance brokers after they have been signed by the customers. As mentioned before, there are two techniques for doing this:

- Shredding the documents into relational tables
- Storing the documents intact in an XML column

14.3.1 Shredding XML documents into relational tables

In this section, we provide an example of how you can shred incoming XML documents and store them in a set of relational DB2 tables using DB2 XML Extender.

Context

After receiving our policy quote back from the Web service, usually after some additional negotiations with the customer, the insurance broker draws up an insurance policy and sends it to our fictitious insurance company, ITSO Insurance, Inc. As mentioned before, insurance brokers are XML fans, so the policy information that they send us is an XML document.

Since we already have our existing infrastructure to handle new insurance policies, we want to keep that existing infrastructure in place and reuse it for policies coming in through our new broker channel. Remember that we normally use self-employed agents to sell our insurance policies.

Our existing infrastructure is based on a relational database model using DB2 to store and process all of our data. Because the policies coming in through the broker channel are XML documents, we need to transform and shred them so we can store them in an existing set of tables to be processed by our application that handles new policies.

As an alternative to shredding the incoming broker XML documents into the actual DB2 tables for processing, you can shred the XML documents into a set of shadow tables. These shadow tables have an identical layout to the tables of the application that handles the new policies. Another application can first validate the data from the shadow tables before putting it into the actual tables that are used by the application that handles the new policies. Since we do not have the same validation process in place for incoming information from brokers as we do from agents, this might be a wise step to take.

14.3.2 Shredding implementation details

The following items are discussed:

- Insurance policy DTD
- Relational data model to be used for shredding XML documents
- Shredding sequence
- Shredding details

Insurance policy DTD

The insurance broker sends us a new policy in the form of an XML document. This document adheres to the following DTD (Example 14-5). More details on the DTD syntax can be found in “Document type definition” on page 560.

Example 14-5 Policy DTD

```
<!--
  DTD for an insurance policy.
-->
<!--
  Insurance policy entities:

  - MaritalStatus gives the marital status of an owner
    or insured person. We use an attribute because it allows
    us to restrict the set of legal values.
    S= Single, M=Married, D=Divorced, W=Widowed

-->
<!ENTITY % MaritalStatus "MaritalStatus (S | M | D | W) #REQUIRED">
<!ENTITY % ID "ID CDATA #REQUIRED">
<!--
  Policy is the root element type.
-->
<!ELEMENT Policy (Number, Owner, Insured, PolicyType, Broker)>
<!ELEMENT Number (#PCDATA)>
<!--
  Owner is the entity that pays for the insurance policy and is
  the beneficiary of it. An owner can be a person or an organization.
-->
<!ELEMENT Owner (Address, (Person | Organization))>
<!ATTLIST Owner oID CDATA #REQUIRED>
<!--
  Address is a generic address. State is optional because not
  all countries have states.
-->
<!ELEMENT Address (Street1, Street2?, City, State?, PostCode, Country)>
<!ELEMENT Street1 (#PCDATA)>
<!ELEMENT Street2 (#PCDATA)>
<!ELEMENT City (#PCDATA)>
<!ELEMENT State (#PCDATA)>
<!ELEMENT PostCode (#PCDATA)>
<!ELEMENT Country (#PCDATA)>
<!--
  Person describes a person who owns (pays for and benefits
  from) an insurance policy.
-->
<!ELEMENT Person (Name)>
<!ATTLIST Person
   %MaritalStatus;
>
<!ELEMENT Name (FirstName, LastName)>
<!ELEMENT FirstName (#PCDATA)>
<!ELEMENT LastName (#PCDATA)>
<!--
  Organization describes an organization that owns (pays for and benefits
  from) an insurance policy.
-->
<!ELEMENT Organization (OrganizationName, TradingName1?, TradingName2?)>
<!ELEMENT OrganizationName (#PCDATA)>
<!ELEMENT TradingName1 (#PCDATA)>
<!ELEMENT TradingName2 (#PCDATA)>
<!--
  Insured describes a person who is covered by a insurance policy.
-->
<!ELEMENT Insured (Name, Age)>
<!ATTLIST Insured %MaritalStatus;>
<!ATTLIST Insured iID  CDATA #REQUIRED>
<!ELEMENT Age (#PCDATA)>
<!--
  PolicyType describes one of a pre-defined set of policies,
  such as life insurance for someone who is of a certain age.
-->
<!ELEMENT PolicyType (PlanName, PlanType, ActualPremium)>
<!ATTLIST PolicyType pID CDATA #REQUIRED>
<!ELEMENT PlanName (#PCDATA)>
<!ELEMENT PlanType (#PCDATA)>
```

```
<!ELEMENT ActualPremium (#PCDATA)>
<!--
  Broker describes a third party (broker) who sells insurance
  policies on behalf of the insurance company.
-->
<!ELEMENT Broker (BrokerName, Address)>
<!ATTLIST Broker bID CDATA #REQUIRED>
<!ELEMENT BrokerName (#PCDATA)>
```

As you can see, the structure of our scenario is not very complex. XML documents adhering to this DTD contain the following important information that needs to be mapped to a set of relational tables during the shredding process:

- Policy information, such as a policy number
- Owner information, such as the person who owns the policy or organization that pays the premium
- Information about the insured, such as the person who is actually covered by the policy
- Policy type information, such as the kind of policy (health or homeowners, etc.) being quoted
- Broker information, such as the broker who closes the deal on the policy

Example 14-6 shows a sample policy document that adheres to the DTD demonstrated in Example 14-5 on page 417.

Example 14-6 Sample policy XML document

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Policy SYSTEM "policy.dtd">
<Policy>
   <Number>99</Number>
   <Owner oID="123">
      <Address>
         <Street1>550 W Hamilton Avenue</Street1>
         <Street2/>
         <City>Campbell</City>
         <State>California</State>
         <PostCode>95008</PostCode>
         <Country>US</Country>
      </Address>
      <Person MaritalStatus="S">
         <Name>
            <FirstName>Olivier</FirstName>
            <LastName>Guyennet</LastName>
         </Name>
      </Person>
   </Owner>
   <Insured MaritalStatus="S" iID="987">
      <Name>
         <FirstName>Shrinivas</FirstName>
         <LastName>Kulkarni</LastName>
      </Name>
      <Age>25</Age>
   </Insured>
   <PolicyType pID="789">
      <PlanName>MyPlan</PlanName>
      <PlanType>It is a very nice plan type</PlanType>
      <ActualPremium>1234.56</ActualPremium>
   </PolicyType>
   <Broker bID="456">
      <BrokerName>Stephen Priestley</BrokerName>
      <Address>
         <Street1>Wombat Street</Street1>
         <City>Melbourne</City>
         <PostCode>5555</PostCode>
         <Country>Australia</Country>
      </Address>
   </Broker>
</Policy>
```

Relational model to be used in the shredding process

When we shred incoming XML documents, we use a set of existing tables (or a set of shadow tables). To start off, we show you the relational data model that we use in this scenario. Figure 14-10 on page 421 shows the logical data model that is used by ITSO Insurance, Inc.

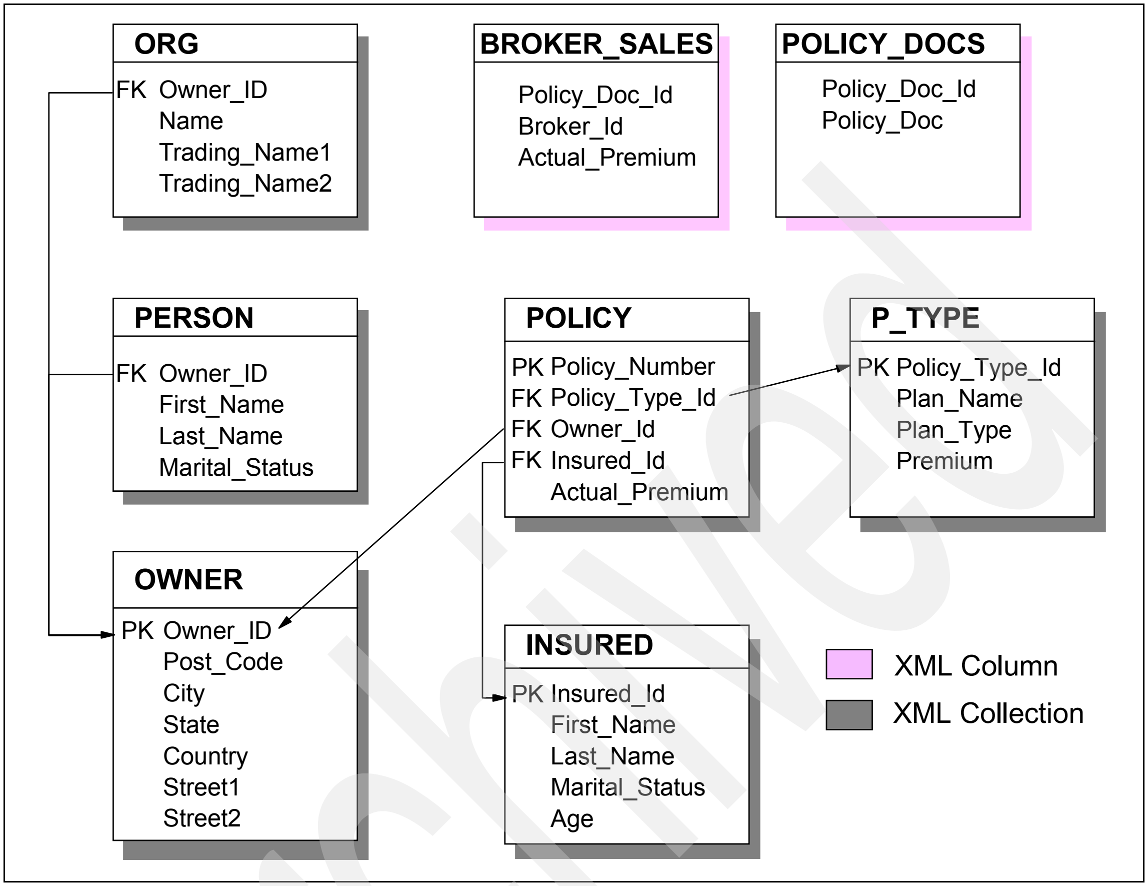


Figure 14-10 Logical data model

The heart of the model is the POLICY table. It contains the Policy_Number as the unique identifier of any one policy as well as the Actual_Premium that is to be paid for this policy. The rest of the columns of the POLICY table contain references (foreign keys) to other tables, such as INSURED, OWNER and P_TYPE.

P_TYPE describes the different policy types, the plans offered by ITSO Insurance, Inc., as well as a base premium (starting point for calculations) for each policy type or plan.

The INSURED table contains information about the person who is being protected by the policy from the risks it covers.

Another table in the model is the OWNER table. It contains information about the person who is paying for the insurance. In our case, this can be a person or an organization. More detailed information about the person(s) picking up the bill can be found in the PERSON table. If an organization is providing health insurance coverage for their employees, more details can be found in the ORG table. For any Owner_ID in the OWNER table, more details can be found in either in the ORG table or in the PERSON table, never in both.

The POLICY_DOCS and BROKER_SALES tables are only shown for completeness. They are not used during the shredding process. These tables are used for storing the intact XML documents using XML columns. This is described in more detail in 14.3.3, “Storing XML documents intact in XML columns” on page 429.

Shredding sequence

In many real life cases, it will be impossible to take an XML document and shred it directly into a set of relational tables. Our insurance company scenario is no exception.

Because of the referential integrity relationships that exist between the tables in the relational model, you must always have a parent row before you can insert a child row into a dependent table. They way the XML document is structured, the way the DAD has to be built for shredding, as well as the fact that DB2 always enforces INSERT rules at the statement level (not at the unit of work level), make it impossible to shred our policy document in a single shredding operation.

Because we want to shred into an existing relational model (that is fixed because it is used by existing applications) and are also forced to stick to the layout of the XML document (imposed on us by the broker companies), we must transform the XML document prior to shredding, or use multiple shred operations of parts of the document into a few tables at a time. We will use the multiple-step method in our scenario.

Figure 14-11 on page 423 shows the sequence we used to shred the XML documents into our set of relational tables in a three-step process:

1. First, we do the initial shredding operation to populate the OWNER table and tables directly related to it: ORG and PERSON. Notice that we always shred into the OWNER table, but only to one or the other ORG or PERSON tables for any given incoming policy.
2. Next, we shred into the INSURED table.
3. Last, we populate the POLICY table.

Note: We do not shred into the P_TYPE table. Even though this information is provided in the XML document, it is ignored. The reason for ignoring this data is that a broker sending in an insurance policy is not allowed to send a policy with a new policy type without the new policy type first being approved by ITSO Insurance, Inc. Therefore, it is reasonable to assume that the P_TYPE table is already populated by other business processes used by ITSO Insurance, Inc.

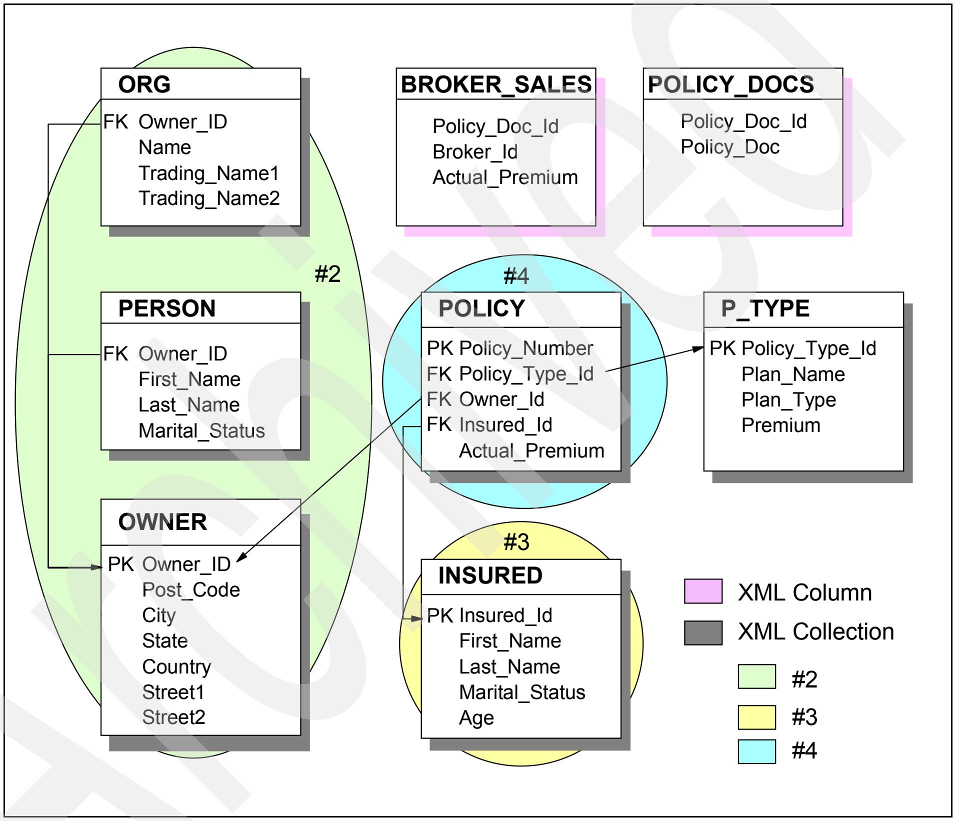


Figure 14-11 Shredding sequence

Shredding details

As stated in the note above, even though the P_TYPE information is provided in the incoming XML document, there is no need to shred this information because it is already present in the database. The PK-FK relationship between POLICY and P_TYPE guarantees that a valid policy type is provided in the incoming XML document.

Shredding into OWNER, ORG and/or PERSON

To shred the OWNER, ORG or PERSON data from the XML document into the corresponding relational tables, we use the DAD file shown in Example 14-7.

Example 14-7 DAD file to shred into ORG, OWNER and/or PERSON

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DAD SYSTEM "C:\SG24-6994\Scenario\dad.dtd">
<DAD>
   <validation>NO</validation>
   <Xcollection>
      <prolog>?xml version="1.0"?</prolog>
      <doctype>!DOCTYPE Policy SYSTEM "policy.dtd"</doctype>
      <root_node>
         <element_node name="Policy" multi_occurrence="YES">
            <RDB_node>
               <table name="OWNER"/>
               <table name="PERSON"/>
               <table name="ORG"/>
               <condition>
                  OWNER.owner_id=PERSON.owner_id AND
                  OWNER.owner_id=ORG.owner_id
               </condition>
            </RDB_node>
            <element_node name="Owner" multi_occurrence="YES">
               <attribute_node name="oID">
                  <RDB_node>
                     <table name="OWNER"/>
                     <column name="owner_id" type="smallint"/>
                  </RDB_node>
               </attribute_node>
               <!-- BEG OF ADDRESS -->
               <element_node name="Address" multi_occurrence="YES">
                  <element_node name="Street1">
                     <text_node>
                        <RDB_node>
                           <table name="OWNER"/>
                           <column name="street1" type="char(50)"/>
                        </RDB_node>
                     </text_node>
                  </element_node>
                  <element_node name="Street2">
                     <text_node>
                        <RDB_node>
                           <table name="OWNER"/>
                           <column name="street2" type="char(50)"/>
                        </RDB_node>
                     </text_node>
                  </element_node>
                  <element_node name="City">
                     <text_node>
                        <RDB_node>
                           <table name="OWNER"/>
                           <column name="city" type="char(50)"/>
                        </RDB_node>
                     </text_node>
                  </element_node>
                  <element_node name="State">
                     <text_node>
                        <RDB_node>
                           <table name="OWNER"/>
                           <column name="state" type="char(10)"/>
                        </RDB_node>
                     </text_node>
                  </element_node>
                  <element_node name="PostCode">
                     <text_node>
                        <RDB_node>
                           <table name="OWNER"/>
                           <column name="post_code" type="char(10)"/>
                        </RDB_node>
                     </text_node>
                  </element_node>
                  <element_node name="Country">
                     <text_node>
                        <RDB_node>
                           <table name="OWNER"/>
                           <column name="country" type="char(10)"/>
                        </RDB_node>
                     </text_node>
                  </element_node>
               </element_node>
               <!-- END OF ADDRESS -->
               <!-- BEG OF PERSON -->
               <element_node name="Person" multi_occurrence="YES">
                  <attribute_node name="MaritalStatus">
                     <RDB_node>
                        <table name="PERSON"/>
                        <column name="marital_status" type="char(2)"/>
                     </RDB_node>
                  </attribute_node>
                  <element_node name="Name">
                     <element_node name="FirstName">
                        <text_node>
                           <RDB_node>
                              <table name="PERSON"/>
                              <column name="first_name" type="char(20)"/>
```

```
                           </RDB_node>
                        </text_node>
                     </element_node>
                     <element_node name="LastName">
                        <text_node>
                           <RDB_node>
                              <table name="PERSON"/>
                              <column name="last_name" type="char(20)"/>
                           </RDB_node>
                        </text_node>
                     </element_node>
                  </element_node>
               </element_node>
               <!-- END OF PERSON -->
               <!-- BEG OF ORGANIZATION -->
               <element_node name="Organization" multi_occurrence="YES">
                  <element_node name="OrganizationName">
                     <text_node>
                        <RDB_node>
                           <table name="ORG"/>
                           <column name="name" type="char(50)"/>
                        </RDB_node>
                     </text_node>
                  </element_node>
                  <element_node name="TradingName1">
                     <text_node>
                        <RDB_node>
                           <table name="ORG"/>
                           <column name="trading_name1" type="char(50)"/>
                        </RDB_node>
                     </text_node>
                  </element_node>
                  <element_node name="TradingName2">
                     <text_node>
                        <RDB_node>
                           <table name="ORG"/>
                           <column name="trading_name2" type="char(50)"/>
                        </RDB_node>
                     </text_node>
                  </element_node>
               </element_node>
               <!-- END OF ORGANIZATION -->
            </element_node>
         </element_node>
      </root_node>
   </Xcollection>
</DAD>
```

The root node must specify all tables involved as well as the join conditions between those tables. The rest of the DAD file is a description of how the elements and attributes of the XML document map to the relational tables. Note that you must use RDB node mapping in the DAD file when shredding.

> **Note:** To verify the validity of your DAD file, you can download a DAD checker utility from:
>
> `http://www.ibm.com/software/data/db2/extenders/xmlext/downloads.html`

To invoke the shredding operation, DB2 XML Extender provides you with a little program, called *dxxshrd*. For example to shred the policy1.xml document, you can use:

```
dxxshrd INSURA PolicyShredSeq2.dad “policy1.xml”
```

where `INSURA` is the name of the DB2 database, and `PolicyShredSeq2.dad` is the name of the DAD file.

Similar logic is used to shred into the INSURED and POLICY tables.

Shredding into the INSURED table

To shred part of the policy document into the INSURED table, we use the PolicyShredSeq3.dad file.

Shredding into the POLICY table

To shred part of the policy document into the POLICY table, we use the PolicyShredSeq4.dad file.

To invoke the shredding function, you can use the following batch file:

```
C:\sg246994\Scenario>ShredXMLDocs
```

This command file shreds our policy documents into our relational tables. It is normal that the shredding of policy4.xml generates an error message when shredding, because the same OWNER_ID already exists. This is due to the fact that policy3.xml has the same OWNER_ID. You can just ignore this message. Real life applications should test for proper error code, making sure it is safe to ignore the error. The same applies to INSURED, because we already have this information from another policy.

`ShredXMLDocs` uses 2 batch files as shown in Example 14-8 on page 428 and Example 14-9 on page 428.

Example 14-8 ShredXMLDocs.bat

```
REM Shred (4) XML policy documents into multiples tables
REM PARAM 1 = DB Name

call ShredSeq.bat %1 "policy1.xml"
call ShredSeq.bat %1 "policy2.xml"
call ShredSeq.bat %1 "policy3.xml"
call ShredSeq.bat %1 "policy4.xml"
```

Example 14-9 ShredSeq.bat

```
REM Shreds the given Policy document into multiples tables
REM PARAM 1 = DB Name
REM PARAM 2 = XML Document file name
REM (Ex.Policy1.xml passed in using double quotes)

REM PolicyShredSeq2.dad specifies shredding into ORG, OWNER and PERSON
REM PolicyShredSeq3.dad specifies shredding into INSURED table
REM PolicyShredSeq4.dad specifies shredding into POLICY table

dxxshrd %1 PolicyShredSeq2.dad %2
dxxshrd %1 PolicyShredSeq3.dad %2
dxxshrd %1 PolicyShredSeq4.dad %2
```

The dxxshrd program invokes the dxxShredXML shredding stored procedure. The program dxxshrd comes with DB2 XML Extender and resides in the your-DB2-install-dir\SQLLIB\BIN directory.

Notice that DB2 XML Extender can only insert new data into a set of specified relational tables when shredding an XML document. If the data already exists, you will get a negative SQL code indicating that a primary key with that value already exists.

To make sure your data is handled properly, applications should test for these situations and take proper action when shredding. This may be another case where it makes sense to shred into a set of shadow tables instead of directly shredding into the actual tables used by other applications.

Due to time constraints, we did not implement such proper testing in this scenario. In real life applications, you should do this.

14.3.3 Storing XML documents intact in XML columns

In this section, we demonstrate how ITSO Insurance, Inc. can store incoming XML documents intact using the DB2 XML Extender, XML column feature.

Context

As mentioned before, we want to start using independent insurance brokers as additional channels to sell our insurance policies. However, all our existing applications assume that policies come in through our network of self-employed agents. Our current IT systems are not capable of handling information pertaining to brokers. On the other hand, we do not want to wait for the existing IT infrastructure to be changed before we start selling insurance policies through the broker channel.

How do we solve this problem?

We can implement full broker support in stages. Because we want to start using the broker channel as soon as possible, we do the minimum amount of work to get the process up and running.

This means that all the information (except the broker information) that we need for processing policies as we normally do is extracted from the incoming XML policy document and is stored in existing (or shadow) relational tables as described in 14.3.1, “Shredding XML documents into relational tables” on page 416.

So as not to loose any of the information coming in via XML policy documents through the broker channel, and also for legal reasons, we store the XML documents in the XML column verbatim.

The broker information (and corresponding details that are part of the insurance policy XML document) is not shredded at this stage. It is stored as part of the entire XML document in the XML column using XML Extender’s XML column feature.

Storing the full incoming XML document intact in an XML column also gives us the possibility to reprocess these documents during a subsequent implementation phase when there are more resources available to implement full broker support in our IT operating environment.

However, even in this first implementation phase, we may encounter cases that require us to retrieve customer records for a particular broker. To achieve this, we create a side table (a feature of using an XML column to provide fast access to frequently needed elements or attributes of an XML document) on BROKER_ID and ACTUAL_PREMIUM.

14.3.4 XML column implementation details

We describe the following implementation steps:

- Steps to enable the XML column
- Inserting data into the XML column

Enabling the POLICY_DOC column as an XML column

To be able to store intact XML documents in an XML column, you must first insert the DTD into the DTD_REF table (created by XML Extender when the database was enabled for XML usage) as well as enable the XML column. This is described in more detail in 4.4, “Storing intact XML documents with XML Extender” on page 101. The DTD_REF table information is used when you specify that validation is required.

We used the following .bat file to set up the XMLColumn as shown in Example 14-10.

```
C:\sg246994\Scenario>SetupXMLColumn
```

If you have been following all the steps of this scenario, you have already enabled the XML column as was covered in “Setting up the INSURA database” on page 402.

Example 14-10 SetupXMLColumn.bat

```
REM -- Assumes all related files are located under C:\SG246994\Scenario
REM -- Should be modified in commands below in case path differs
REM -- PARM 1 DBname

db2 connect to %1

db2 "insert into db2xml.dtd_ref (dtdid,content,usage_count,author,creator,
     updator) values ('C:\SG246994\Scenario\policy.dtd',
     db2xml.XMLCLOBFromFile('C:\SG246994\Scenario\policy.dtd'),0,'xml','xml','
    xml')

dxxadm enable_column INSURA Policy_Docs Policy_Doc
                     "C:\SG246994\Scenario\BrokerXcolumn.dad" -r POLICY_DOC_ID
```

As you can see, we first inserted the policy.dtd information into the DTD_REF control table. Then we enabled the Policy_Doc column in the Policy_Docs table in the INSURA database. In the enable_column command, you can also indicate the DAD file that is to be used when storing information into the XML column.

Note: The `dtdid` does not need to be the same as the file name. In fact, since our DAD in Example 14-11 specifies that no validation is required, inserting the DTD into the DTD_REF table is unnecessary. However, we included this in our example in case we may want to switch the validation on later.

When you are storing intact XML documents in an XML column, the DAD file is used to indicate what side table(s) need to be created, what information (columns) makes up the side table, and how to retrieve that information from the XML document.

Example 14-11 shows the BrokerXcolumn DAD file that we used for the scenario. As you can see, we use a single side table, Broker_Sales, with two columns, Broker_Id and Actual_Premium. For both columns we use a location path expression to indicate where this information can be found in the XML document. For example for Broker_Id, the path is the bID attribute that can be found using the /Policy/Broker/ path into the XML document. Notice that we also indicate the DB2 data type that is to be used for the side table columns.

Example 14-11 BrokerXcolumn.dad

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DAD SYSTEM "C:\SG24-6994\Scenario\dad.dtd">
<DAD>
   <dtdid/>
   <validation>NO</validation>
   <Xcolumn>
      <table name="Broker_Sales">
         <column name="Broker_Id" type="smallint"
                                  path="/Policy/Broker/@bID"
                                  multi_occurrence="NO"/>
         <column name="Actual_Premium" type="decimal(10,2)"
                                       path="/Policy/PolicyType/ActualPremium"
                                       multi_occurrence="NO"/>
      </table>
   </Xcolumn>
</DAD>
```

When you enable the XML column, XML Extender processes the information from the DAD file and creates the side table(s). It also creates a set of triggers (an insert, an update and a delete trigger) for maintaining the side tables. Using this method, the side tables get automatically populated when an insert is done into the POLICY_DOC column of the POLICY_DOCS table. The same is true for an update or delete operation. When dealing with XML columns, the side tables are kept current when update and delete operations occur on information in the XML column. (Remember that this is not the case when doing shredding.)

Inserting data into the POLICY_DOCS table

Once the XML column is enabled, you just insert, update or delete into the XML-enabled column like you would for any other table. The triggers on the XML-enabled column ensure that the side tables are kept in sync with the base data in the XML column.

To store all our policy documents in the XML column, you can use the following batch file:

```
C:\sg246994\Scenario>StoreXMLColumn
```

This command file stores the policy documents as complete XML documents in our XML column (`Policy_Doc` in the `Policy_Docs` table) and populates the side tables. Example 14-12 shows how to insert a single XML document into the XML column.

Example 14-12 Insert data into the XML column

```
db2 "insert into POLICY_DOCS
       values (1,db2xml.XMLCLOBFromFile('C:\SG24-6994\Scenario\policy1.xml'))"
```

To guarantee that the side tables and XML column information are kept in sync, you must have a unique key on the table that contains the XML-enabled column. You can provide this information when enabling the XML column by providing a root name (-r option on the dxxadmin enable_column command). We used POLICY_DOC_ID in Example 14-10 on page 430.

14.4 Scenario 3: composing XML documents

In this scenario, we demonstrate how the ITSO Insurance, Inc. can publish XML documents from data that is stored, either in a set of relational tables (XML collection or regular tables), or from data stored in an XML column.

14.4.1 Publishing XML from relational

This section shows how we can create XML documents based on data that is stored in relational tables. We show how ITSO Insurance, Inc. can generate reports based on information stored in the shredded documents.

Context

We like to provide our corporate and other business customers with an overview of the types of policies they have with us, the employees who are insured, and the amounts of their individual premiums. We call this report the orgreport (organization report).

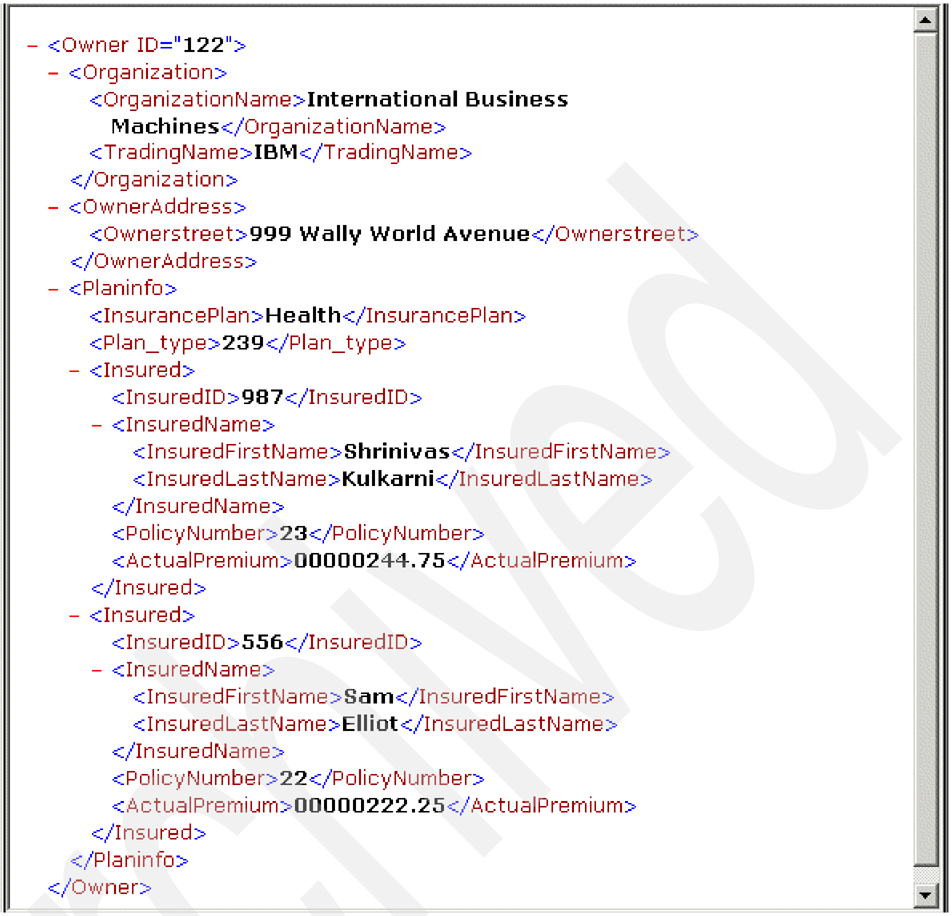
14.4.2 Publishing XML from relational data implementation

As has been described throughout this publication, there are different ways to publish XML documents from data stored in relational tables. The preferred method is to use SQL/XML. In the following pages, we describe how to publish XML documents from relational tables using SQL/XML, a DAD using RDB_node mapping, and a DAD using an SQL statement as input.

Using SQL/XML

As stated earlier, the objective in this scenario is to list all the policy types an organization has with our insurance company as well as the people who are insured through these policies.

Figure 14-12 on page 434 shows the results of an SQL/XML query that satisfies the stated objectives. Let's first analyze its structure.



```
- <Owner ID="122">
  - <Organization>
      <OrganizationName>International Business
        Machines</OrganizationName>
      <TradingName>IBM</TradingName>
    </Organization>
  - <OwnerAddress>
      <Ownerstreet>999 Wally World Avenue</Ownerstreet>
    </OwnerAddress>
  - <Planinfo>
      <InsurancePlan>Health</InsurancePlan>
      <Plan_type>239</Plan_type>
    - <Insured>
        <InsuredID>987</InsuredID>
      - <InsuredName>
          <InsuredFirstName>Shrinivas</InsuredFirstName>
          <InsuredLastName>Kulkarni</InsuredLastName>
        </InsuredName>
        <PolicyNumber>23</PolicyNumber>
        <ActualPremium>00000244.75</ActualPremium>
      </Insured>
    - <Insured>
        <InsuredID>556</InsuredID>
      - <InsuredName>
          <InsuredFirstName>Sam</InsuredFirstName>
          <InsuredLastName>Elliot</InsuredLastName>
        </InsuredName>
        <PolicyNumber>22</PolicyNumber>
        <ActualPremium>00000222.25</ActualPremium>
      </Insured>
    </Planinfo>
  </Owner>
```

Figure 14-12 Result of SQL/XML query

SQL/XML queries themselves normally only produce so-called XML fragments. You can of course concatenate some strings to make it a well-formed complete XML document or use XSLT for that purpose.

The structure that we use here is an organization (IBM), the different policy types the organization has with us (only Health in this case), the insured employees (Sam and Shrinivas), and their respective policy numbers and premiums.

The SQL/XML query that was used to produce the results in Figure 14-12 is shown in Example 14-13 on page 435. It also comes as part of the scenario files and you can invoke this query using the following batch file:

```
C:\sg246994\Scenario>GetOrgReportSQLXML3
```

Example 14-13 SQL/XML query

```
WITH insur(policy_type_id, insurelement)AS
  (select  ITSO.POLICY.POLICY_TYPE_ID,
             xmlelement(name"Insured",
                xmlelement(name"InsuredID",ITSO.POLICY.INSURED_ID),
                xmlelement(name"InsuredName",
                   xmlelement(name"InsuredFirstName", ITSO.INSURED.FIRST_NAME),
                   xmlelement(name"InsuredLastName", ITSO.INSURED.LAST_NAME)
                          ),
                xmlelement(name"PolicyNumber" ,ITSO.POLICY.POLICY_NUMBER),
                xmlelement(name"ActualPremium" ,ITSO.POLICY.ACTUAL_PREMIUM)
                       )
    FROM
      ITSO.POLICY ,
      ITSO.INSURED
    WHERE
      ITSO.POLICY.INSURED_ID = ITSO.INSURED.INSURED_ID

   ),

   planinfo (plan_name,planelement) AS

     ( select     ITSO.P_TYPE.PLAN_NAME,
                  xmlelement(name"Planinfo",
                       xmlelement(name"InsurancePlan",ITSO.P_TYPE.PLAN_NAME),
                       xmlelement(name"Plan_type",ITSO.P_TYPE.POLICY_TYPE_ID),
                       (SELECT XMLAGG(insur.insurelement)
                           FROM insur
                            WHERE
                             ITSO.P_TYPE.POLICY_TYPE_ID = insur.POLICY_TYPE_ID
                       )
              )
         from ITSO.P_TYPE
      )

select
 varchar(
  xml2clob(
          xmlelement( name "Owner",
                  xmlattributes (ITSO.OWNER.OWNER_ID AS "ID"),
                  xmlelement(name "Organization",
                       xmlelement(name"OrganizationName", ITSO.ORG.NAME),
                       xmlelement(name"TradingName", ITSO.ORG.TRADING_NAME1)
                                  ),
                     xmlelement(name "OwnerAddress",
                xmlelement(name "Ownerstreet", ITSO.OWNER.STREET1)
                    ),
                ( select XMLAGG(planelement)
                                 from planinfo
                                 where planinfo.plan_name = ITSO.P_TYPE.PLAN_NAME)
                 )
        ),3000)

FROM
      ITSO.OWNER,
      ITSO.ORG,
   ITSO.POLICY,
   ITSO.P_TYPE,
      ITSO.INSURED

WHERE
   ITSO.OWNER.OWNER_ID = ITSO.ORG.OWNER_ID  AND
   ITSO.OWNER.OWNER_ID = ITSO.POLICY.OWNER_ID  AND
   ITSO.POLICY.POLICY_TYPE_ID = ITSO.P_TYPE.POLICY_TYPE_ID AND
   ITSO.POLICY.INSURED_ID = ITSO.INSURED.INSURED_ID AND
   ITSO.ORG.TRADING_NAME1='IBM'
GROUP BY ITSO.P_TYPE.PLAN_NAME,ITSO.OWNER.OWNER_ID
,ITSO.ORG.NAME,ITSO.ORG.TRADING_NAME1,ITSO.OWNER.STREET1
 ;
```

This seems like a difficult query to write, but let us look at it in more detail. When we do, we find that it really is not all that bad.

The current version of DB2 does not support nesting of XMLAGG functions, so we have to use common table expressions (CTE), as in our example above, or inline views. You need to create a CTE for each fragment nesting level you want to create, then nest the fragments by joining them using their PK-FK relationships.

First of all, we are looking for the policy the IBM organization has with us:

```
ITSO.ORG.TRADING_NAME1='IBM'
```

When coding the CTEs, we start at the lowest (deepest) level in the XML document. This is the insured person. Using this method, we build the *insur* CTE as shown in Example 14-14 on page 437:

Example 14-14 Building the insur CTE

```
WITH insur(policy_type_id, insurelement)AS
  (select  ITSO.POLICY.POLICY_TYPE_ID,
             xmlelement(name"Insured",
                xmlelement(name"InsuredID",ITSO.POLICY.INSURED_ID),
                xmlelement(name"InsuredName",
                   xmlelement(name"InsuredFirstName",
                                   ITSO.INSURED.FIRST_NAME),
                   xmlelement(name"InsuredLastName",
                                   ITSO.INSURED.LAST_NAME)
                          ),
                xmlelement(name"PolicyNumber" ,ITSO.POLICY.POLICY_NUMBER),
                xmlelement(name"ActualPremium" ,ITSO.POLICY.ACTUAL_PREMIUM)
                       )
    FROM
      ITSO.POLICY ,
      ITSO.INSURED
    WHERE
      ITSO.POLICY.INSURED_ID = ITSO.INSURED.INSURED_ID

   )
```

The insur CTE has two columns, which are described in Table 14-1:

Table 14-1 insur CTE

Column name	Definition
insurelement	This column constructs the <Insured> element and its sub-elements <InsuredID>, <InsuredName>, <PolicyNumber> and <ActualPremium>. To do so we join the POLICY and INSURED tables on INSURED_ID.
policy_type_id	This column is required so we can use the insur CTE later with the planinfo CTE information.

The *planinfo* CTE looks like Example 14-15 on page 438:

Example 14-15 Planinfo CTE

```
planinfo (plan_name,planelement) AS
     ( select     ITSO.P_TYPE.PLAN_NAME,
                  xmlelement(name"Planinfo",
                   xmlelement(name"InsurancePlan",ITSO.P_TYPE.PLAN_NAME),
                   xmlelement(name"Plan_type",ITSO.P_TYPE.POLICY_TYPE_ID),
                   (SELECT XMLAGG(insur.insurelement)
                      FROM INSUR
                      WHERE
                          ITSO.P_TYPE.POLICY_TYPE_ID = insur.POLICY_TYPE_ID
                   )
                  )
         from ITSO.P_TYPE
   )
```

Notice that in Example 14-15, we aggregate the insurelement that we built in the insur CTE inside the planinfo CTE and correlate it to the POLICY_TYPE_ID column. This is why we needed to extract that column during the insur CTE execution.This CTE creates the `<planinfo>` element (in the planelement column). Also notice that this time, we extract an extra column (plan_name) to be able to join this information in the top-level query that we discuss later.

Example 14-16 Top-level SQL statement

```
select
 varchar(
  xml2clob(
             xmlelement( name "Owner",
                  xmlattributes (ITSO.OWNER.OWNER_ID AS "ID"),
                  xmlelement(name "Organization",
                   xmlelement(name"OrganizationName", ITSO.ORG.NAME),
                   xmlelement(name"TradingName",
                                   ITSO.ORG.TRADING_NAME1)
                            ),
                  xmlelement(name "OwnerAddress",
                  xmlelement(name "Ownerstreet", ITSO.OWNER.STREET1)
                   ),
             ( select XMLAGG(planelement)
                 from planinfo
                  where planinfo.plan_name = ITSO.P_TYPE.PLAN_NAME
             )
          )
        ),3000)

FROM
      ITSO.OWNER,
      ITSO.ORG,
       ITSO.POLICY,
       ITSO.P_TYPE,
       ITSO.INSURED
WHERE
   ITSO.OWNER.OWNER_ID = ITSO.ORG.OWNER_ID  AND
   ITSO.OWNER.OWNER_ID = ITSO.POLICY.OWNER_ID  AND
   ITSO.POLICY.POLICY_TYPE_ID = ITSO.P_TYPE.POLICY_TYPE_ID AND
   ITSO.POLICY.INSURED_ID = ITSO.INSURED.INSURED_ID AND
   ITSO.ORG.TRADING_NAME1='IBM'
GROUP BY ITSO.P_TYPE.PLAN_NAME,ITSO.OWNER.OWNER_ID
        ,ITSO.ORG.NAME,ITSO.ORG.TRADING_NAME1,ITSO.OWNER.STREET1
```

In Example 14-16 on page 438, we include the XML2CLOB function to convert from the internal XML data type to a CLOB data type as this is a top-level query. To make it easy to process for us in the DB2 command window, we convert the CLOB to a VARCHAR.

Again, we use the same techniques as before, creating some elements and attributes, and nesting in the result from the previous planinfo CTE using the XMLAGG function.

Notice that we have to join all five tables to be able to obtain all relevant information relevant to the IBM organization. As SQL does not allow GROUP BY PLAN_NAME and OWNER_ID, we also include the other columns.

Using SQL statement mapping in the DAD

Let us now try to construct a similar output using the SQL statement mapping technique.

Example 14-17 uses a DAD file with an SQL statement to construct the same organization report that we used in the previous section. It can also be obtained using the following batch file from the scenario:

```
C:\sg246994\Scenario>GetOrgReportSQL3
```

Example 14-17 SQL statement DAD

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "C:\SQLLIB\samples\db2xml\dtd\dad.dtd">
<DAD>
<validation>YES</validation>
<Xcollection>
<SQL_stmt>
SELECT ITSO.OWNER.OWNER_ID,
       ITSO.ORG.NAME,
      ITSO.ORG.TRADING_NAME1,
      ITSO.OWNER.STREET1,
      ITSO.P_TYPE.POLICY_TYPE_ID,
      ITSO.P_TYPE.PLAN_NAME,
      ITSO.POLICY.INSURED_ID,
      ITSO.POLICY.POLICY_NUMBER,
      ITSO.INSURED.LAST_NAME,
      ITSO.INSURED.FIRST_NAME,
      ITSO.POLICY.ACTUAL_PREMIUM

 FROM
   ITSO.OWNER,
   ITSO.ORG,
   ITSO.POLICY,
   ITSO.P_TYPE,
   ITSO.INSURED
  WHERE
   ITSO.OWNER.OWNER_ID = ITSO.ORG.OWNER_ID  AND
   ITSO.OWNER.OWNER_ID = ITSO.POLICY.OWNER_ID  AND
   ITSO.POLICY.POLICY_TYPE_ID = ITSO.P_TYPE.POLICY_TYPE_ID AND
   ITSO.POLICY.INSURED_ID = ITSO.INSURED.INSURED_ID AND
   ITSO.ORG.TRADING_NAME1='IBM'
  ORDER BY OWNER_ID , POLICY_TYPE_ID, INSURED_ID
</SQL_stmt>
<prolog>?xml version="1.0"?</prolog>
<doctype>!DOCTYPE OWNER POLICYS  "C:\db2xml\dtd\orgreport.dtd"</doctype>
<root_node>
<element_node name="Owner">
   <attribute_node name="ID">
      <column name="OWNER_ID"/>
   </attribute_node>
   <element_node name="Organization">
      <element_node name="OrganizationName">
         <text_node><column name="NAME"/></text_node>
      </element_node>
      <element_node name="TradingName1">
         <text_node><column name="TRADING_NAME1"/></text_node>
      </element_node>
   </element_node>
   <element_node name="OwnerAddress">
      <element_node name="OwnerStreet">
         <text_node><column name="STREET1"/></text_node>
      </element_node>
   </element_node>
   <element_node name="Planinfo" multi_occurrence="YES">
      <element_node name="InsurancePlan">
         <text_node><column name="PLAN_NAME"/></text_node>
      </element_node>
      <element_node name="Plan_type">
```

```
         <text_node><column name="POLICY_TYPE_ID"/></text_node>
      </element_node>
         <element_node name="Insured" multi_occurrence="YES">
            <element_node name="InsuredID">
               <text_node><column name="INSURED_ID"/></text_node>
            </element_node>
            <element_node name="InsuredName" >
               <element_node name="InsuredFirstName">
                  <text_node><column name="FIRST_NAME"/></text_node>
               </element_node>
               <element_node name="InsuredLastName">
                  <text_node><column name="LAST_NAME"/></text_node>
               </element_node>
            </element_node>
            <element_node name="PolicyNumber">
               <text_node><column name="POLICY_NUMBER"/></text_node>
            </element_node>
            <element_node name="ActualPremium">
               <text_node><column name="ACTUAL_PREMIUM"/></text_node>
            </element_node>
         </element_node>
   </element_node>
   </element_node>
</root_node>
</Xcollection>
</DAD>
```

The DAD file is fairly straightforward. The SQL statement in the <SQL_stmt> tag gathers the data that you need to put into the resulting XML document. To obtain all information related to plans and insured people for the IBM organization, we need to use a five-way join. It is easiest to list the columns in the select clause in the same order as they appear in the XML document (although that is not a requirement). You also need to specify an ORDER BY to be able to handle the multi_occurrence=”yes” elements properly. After the </SQL_stmt> tag, the layout of the XML document starts and the mapping of the columns returns the SQL statement to the elements and attributes of the XML document.

To invoke the generation of the XML document, you can use the following statement (Example 14-18 on page 442):

Example 14-18 Invoking SQL statement based DAD document generation

```
db2 call db2xml.dxxgenxmlclob(
          db2xml.xmlclobfromfile('C:\SG246994\Scenario\OrgReportSQL3.dad'),
          0,
          '',
          ?,
          ?,
          ?,
          ?,
          ?)
```

Using RDB_node mapping in the DAD

In Example 14-19, we produce a similar report, listing the plans and employees within the plan for the organization. However, in this case we use RDB_node mapping to generate the results.

Example 14-19 RDB_node mapping for retrieval from an XML collection

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DAD SYSTEM "C:\SQLLIB\samples\db2xml\dtd\dad.dtd">
<DAD>
   <validation>NO</validation>
   <Xcollection>
      <prolog>?xml version="1.0"?</prolog>
      <doctype>!DOCTYPE Report SYSTEM "C:\SG246994\Scenario\orgreport.dtd"</doctype>
      <root_node>
      <element_node name="Owner">
            <RDB_node>
               <table name="ITSO.ORG"/>
               <table name="ITSO.OWNER"/>
               <table name="ITSO.P_TYPE"/>
               <table name="ITSO.POLICY"/>
               <table name="ITSO.INSURED"/>
               <condition>
                  ITSO.P_TYPE.POLICY_TYPE_ID=ITSO.POLICY.POLICY_TYPE_ID AND
                  ITSO.OWNER.OWNER_ID=ITSO.ORG.OWNER_ID AND
                  ITSO.ORG.OWNER_ID=ITSO.POLICY.OWNER_ID AND
                  ITSO.POLICY.INSURED_ID=ITSO.INSURED.INSURED_ID

               </condition>
            </RDB_node>
            <attribute_node name="ID">
               <RDB_node>
                  <table name="ITSO.ORG"/>
```

```
         <column name="OWNER_ID"/>
      </RDB_node>
   </attribute_node>
<element_node name="Organization">
   <element_node name="OrganizationName">
      <text_node>
         <RDB_node>
            <table name="ITSO.ORG"/>
            <column name="NAME"/>
         </RDB_node>
      </text_node>
   </element_node>
   <element_node name="TradingName1">
      <text_node>
         <RDB_node>
            <table name="ITSO.ORG"/>
            <column name="TRADING_NAME1"/>
            <condition>trading_name1 = 'IBM'</condition>
         </RDB_node>
      </text_node>
   </element_node>
</element_node>
<element_node name="OwnerAddress">
   <element_node name="OwnerStreet">
      <text_node>
         <RDB_node>
            <table name="ITSO.OWNER"/>
            <column name="STREET1"/>
         </RDB_node>
      </text_node>
   </element_node>
</element_node>
<element_node name="Planinfo" multi_occurrence="YES">
   <element_node name="Plan_type">
      <text_node>
         <RDB_node>
            <table name="ITSO.P_TYPE"/>
            <column name="POLICY_TYPE_ID"/>
         </RDB_node>
      </text_node>
   </element_node>
   <element_node name="InsurancePlan">
      <text_node>
         <RDB_node>
            <table name="ITSO.P_TYPE"/>
            <column name="PLAN_NAME"/>
         </RDB_node>
      </text_node>
   </element_node>
```

```
         <element_node name="Insured" multi_occurrence="YES">
            <element_node name="InsuredID">
               <text_node>
                  <RDB_node>
                     <table name="ITSO.POLICY"/>
                     <column name="INSURED_ID"/>
                  </RDB_node>
               </text_node>
            </element_node>
            <element_node name="InsuredName" multi_occurrence="YES">
               <element_node name="InsuredFirstName">
                  <text_node>
                     <RDB_node>
                        <table name="ITSO.INSURED"/>
                        <column name="FIRST_NAME"/>
                     </RDB_node>
                  </text_node>
               </element_node>
               <element_node name="InsuredLastName">
                  <text_node>
                     <RDB_node>
                        <table name="ITSO.INSURED"/>
                        <column name="LAST_NAME"/>
                     </RDB_node>
                  </text_node>
               </element_node>
            </element_node>
            <element_node name="PolicyNumber">
               <text_node>
                  <RDB_node>
                     <table name="ITSO.POLICY"/>
                     <column name="POLICY_NUMBER"/>
                  </RDB_node>
               </text_node>
            </element_node>
            <element_node name="ActualPremium">
               <text_node>
                  <RDB_node>
                     <table name="ITSO.POLICY"/>
                     <column name="ACTUAL_PREMIUM"/>
                  </RDB_node>
               </text_node>
            </element_node>
         </element_node>
      </element_node>
      </element_node>
      </root_node>
   </Xcollection>
</DAD>
```

In Example 14-19 on page 442, the root node <RDB_node> tag contains the list of tables involved as well as the <condition> tag that indicates how they relate to each other. Each attribute that is present in the resulting document has an <RDB_node> tag describing the table and column that the value for that attribute is coming from. Each element has a <text_node> element with the <RDB_node> tag inside, describing the table and column that the value for that element is coming from. You can have additional filtering conditions inside the <RDB_node> tag. In our example, we only want to see the information for a particular organization, based on its trading name (<condition>trading_name1 = 'IBM'</condition>).

You can generate the XML document using the RDB_node method by using the statement shown in Example 14-20.

Example 14-20 Creating an RDB_node based XML document

```
db2 call db2xml.dxxgenxmlclob(
                db2xml.xmlclobfromfile(
                    'C:\SG246994\Scenario\OrgReportRDBnode3.dad'),
                0,
                '',
                ?,
                ?,
                ?,
                ?,
                ?)
```

You can also use the following batch file to generate it:

```
C:\sg246994\Scenario>GetOrgReportRDBnode3
```

The XML document that is produced using RDB_node mapping is shown in Example 14-21.

Example 14-21 Resulting XML document from RDB_node mapping

```
Value of output parameters
  --------------------------
  Parameter Name  : RESULTDOC
  Parameter Value : <?xml version="1.0"?>
<!DOCTYPE Report SYSTEM "C:\SG246994\Scenario\orgreport.dtd">
<Owner ID="122">
  <Organization>
    <OrganizationName>International Business Machines </OrganizationName>
    <TradingName1>IBM </TradingName1>
  </Organization>
  <OwnerAddress>
    <OwnerStreet>999 Wally World Avenue </OwnerStreet>
  </OwnerAddress>
  <Planinfo>
    <Plan_type>789</Plan_type>
    <InsurancePlan>MyPlan </InsurancePlan>
  </Planinfo>
  <Planinfo>
    <Plan_type>232</Plan_type>
    <InsurancePlan>Divorce Plan </InsurancePlan>
  </Planinfo>
  <Planinfo>
    <Plan_type>239</Plan_type>
    <InsurancePlan>Health </InsurancePlan>
    <Insured>
      <InsuredID>556</InsuredID>
      <InsuredName>
        <InsuredFirstName>Sam                 </InsuredFirstName>
        <InsuredLastName>Elliot              </InsuredLastName>
      </InsuredName>
      <PolicyNumber>22</PolicyNumber>
      <ActualPremium>222.25</ActualPremium>
    </Insured>
    <Insured>
      <InsuredID>987</InsuredID>
      <InsuredName>
        <InsuredFirstName>Shrinivas           </InsuredFirstName>
        <InsuredLastName>Kulkarni            </InsuredLastName>
      </InsuredName>
      <PolicyNumber>23</PolicyNumber>
      <ActualPremium>244.75</ActualPremium>
    </Insured>
  </Planinfo>
  <Planinfo>
    <Plan_type>555</Plan_type>
    <InsurancePlan>Motor </InsurancePlan>
  </Planinfo>
  <Planinfo>
    <Plan_type>111</Plan_type>
    <InsurancePlan>Housing </InsurancePlan>
  </Planinfo>
</Owner>

  Parameter Name  : VALID
  Parameter Value : -

  Parameter Name  : NUMDOCS
  Parameter Value : 1

  Parameter Name  : RETURNCODE
  Parameter Value : 0

  Parameter Name  : RETURNMSG
  Parameter Value : DXXQ020I  XML successfully generated.


  Return Status = 0
```

14.4.3 Publishing XML data from XML column information

In the following section, we show how ITSO Insurance, Inc. can generate an XML fragment from data stored inside tables of an XML column.

Context

The current table design for ITSO Insurance, Inc. does not include a table to store broker-related information. Up until now they only worked with agents that work exclusively for ITSO Insurance, Inc. Now that the company is using brokers, ITSO Insurance, Inc. would like to provide brokers with reports showing them how much revenue they've generated through us.

In anticipation of this need, when the complete XML document was stored in an XML column, we extracted this information and stored it in tables (see 14.3.3, "Storing XML documents intact in XML columns" on page 429).

Exploiting side table information

In our BROKER_SALES side table, we have the following information:

- POLICY_DOC_ID
- BROKER_ID
- ACTUAL_PREMIUM

For example, if we want to calculate the revenue generated from premiums by each broker, we can easily do so using the query in Example 14-22 on page 448.

Example 14-22 Total premium by broker

```
select
  broker_id AS "Id" ,sum(actual_premium) AS "Total_Premium_Sales"
from broker_sales
group by broker_id
```

This query uses only information (actual_premium and broker_id) in the side table (broker_sales) to generate the result.

If you want to present the result as an XML document you can use SQL/XML to do this, as shown in Example 14-23, or use the batch file from the scenario below.

```
C:\sg246994\Scenario>GetBrokerSalesSQLXML
```

Example 14-23 Total premium by broker using SQL/XML

```
select
  varchar(
   xml2clob(
    XMLELEMENT(NAME "Broker", XMLATTRIBUTES (BROKER_ID AS "Id"),
               XMLELEMENT(NAME "Total_Premium_Sales", sum(actual_premium))
             )
           ),300
         )
from broker_sales
group by broker_id
```

You can also combine information from the side table with information in the XML column. For example, if after running the previous query and analyzing the results, you discover that broker_id 456 brings in the most revenue, you can list all the policies for broker 456 by using the statement in Example 14-24.

Example 14-24 Broker456Policies.sql

```
select varchar(cast(policy_doc as clob ),2000)
  from broker_sales bs, policy_docs pd
  where bs.policy_doc_id=pd.policy_doc_id and
            broker_id = 456
```

Notice that we use the side table to have easy access to the broker_id information and join the broker_sales side table to the policy_docs table that actually contains the XML documents in the policy_doc XML column.

The VARCHAR function is not really needed; it is just used to make the output easier to read. We also found that if you do not put in the VARCHAR function when using the DB2 Command Center, no actual data is returned. This is most likely a Control Center problem which can be easily circumvented by using the VARCHAR function.

To ensure fast access to the data inside the BROKER_SALES side table, you can create indexes on any of the columns (or combinations of columns) as you would do with any table.

14.5 Scenario 4: Web service requestor

This section deals with ITSO Insurance, Inc. acting as a Web service requestor, also known as a Web service consumer.

14.5.1 Context

As a good insurance company, we receive a lot of money from customers that pay their insurance policy premiums. As we do not want to keep all that money in a stocking under our bed, we reinvest part of the premium money in short- and long-term investments. In this scenario, we look at how part of that money can be invested in the short run by buying shares of stock through the stock market. To obtain information about what is going on at the stock market in order to make sound investment decisions, we use a Web service provided by www.xignite.com. This Web service can provide headline information for particular stocks. Ideally, before we decide to buy a certain stock, our investment specialists can invoke this Web service to get some recent background information about the stock for evaluation purposes.

14.5.2 Implementation

You can find out how to set up DB2 as a Web service consumer by reading 10.3, “DB2 as Web service consumer” on page 294.

Here we show how to use a Web service from within an application program using a simple SQL statement.

As our Web service can return multiple rows (multiple headline items) for a certain stock, in our application, we use a table function reference to retrieve the information from the Web service. Example 14-25 shows how to invoke the stock headline information from the Web service using an SQL statement.

Example 14-25 Retrieving stock headlines using DB2 as Web service consumer

```
SELECT * FROM TABLE (
 DB2XML.EXTRACTVARCHARS(
  DB2XML.XMLCLOB(
   DB2XML.SOAPHTTPC(
    'http://www.xignite.com/xnews.asmx',
    'http://www.xignite.com/services/GetStockHeadlines',
    '<m:GetStockHeadlines xmlns:m="http://www.xignite.com/services/"
       SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
     <m:Symbols>IBM</m:Symbols><m:HeadlineCount>12</m:HeadlineCount>
     </m:GetStockHeadlines>'
                   )
                ), '////StockNews/Headline'
                      )
                    ) AS X
```

Because we ask the Web service to return the last 12 headlines for IBM stock, the result can be more than one row. Therefore we retrieve the result of the Web service using a table function reference. The table function reference provides a way to temporarily store the results of the Web service so that the SQL SELECT statement can retrieve them. Otherwise we might have to create some sort of temporary table to hold the results returned by the Web service. However, using a table function reference is much more flexible.

As you can see, this method makes it easy to integrate Web services into existing applications. If you think your programmers will have a hard time coding an SQL statement like the one in Example 14-25, you can hide this logic from the programmers be encapsulating all of this in a user-defined function like the one shown in Example 14-26.

Example 14-26 Web service consumer UDF wrapper returning a table

```
CREATE FUNCTION itso.GetStockHeadlines (Symbols VARCHAR(100),
                                       HeadlineCount INTEGER)
    RETURNS TABLE (StockNews VARCHAR(3000))
    LANGUAGE SQL CONTAINS SQL
    EXTERNAL ACTION NOT DETERMINISTIC
    RETURN
    Select * from Table (db2xml.extractVarchars(
      DB2XML.XMLCLOB(
          db2xml.soaphttpc(
         'http://www.xignite.com/xnews.asmx',
         'http://www.xignite.com/services/GetStockHeadlines',
         '<m:GetStockHeadlines xmlns:m="http://www.xignite.com/services/"
            SOAP-ENV:encodingStyle=
                      "http://schemas.xmlsoap.org/soap/encoding/">'
            || '<m:Symbols>' || Symbols || '</m:Symbols>'
            || '<m:HeadlineCount>' || rtrim( char(HeadlineCount) ) ||
                                                '</m:HeadlineCount>'
         || '</m:GetStockHeadlines>'
      )
   ), '//StockNews/Headline'
)) as X ;
```

In this "wrapper UDF," we extract only the Headline elements from the result of the Web service, construct a table from these elements, and return the table. In Example 14-26 on page 450, we replaced the fixed symbol tag IBM with a variable Symbols and the fixed headline count of 12 with the HeadlineCount variable. Both Symbols and HeadlineCount are input variables to the table UDF.

You can invoke this UDF with the statement shown in Example 14-27. By issuing this query, we are searching for 10 or less news headlines related to IBM stock. A sample result is also shown in Example 14-27 under the title, Headlines.

Example 14-27 Invoking the table UDF to retrieve stock headlines

```
db2 select substr(stocknews,1,79) as headlines
      from table ( itso.getStockheadlines('IBM',10) ) as y

HEADLINES
---------------------------------------------------------------------------
[external] SPECIAL REPORT: Less Bounce in This Tech Rebound
UPDATE - IBM wins big Delta Lloyd outsourcing contract
IBM switches on Europe's most powerful computer
IBM Embraces Eclipse Environment for Autonomic Computing Technologies
[external] U.S. Equities Close With Losses
[external] Cisco, IBM team up on technology security
Linux Moves In On The Desktop
CSC, facing new woes, may hang on to most IRS work
Multimedia Available: IBM Perfects New Method for Making Low Power
Advanced Digital shares tumble on revenue concerns

  10 record(s) selected.
```

This query is also available as part of this scenario using the following batch file:

```
C:\sg246994\Scenario>getheadlines
```

When you get error messages when trying to invoke these consumer UDFs using the Web service:

- Make sure that the Web service itself is available by checking the Web site: http://www.xignite.com/xNews.asmx?op=GetStockHeadlines
- Make sure that when trying to establish a socket call your requests are not stopped by any intermediate firewalls that may be installed. It may very well be that you can invoke the Web service from the Xignite Web site, but that it fails when invoking it from the DB2 SOAP UDFs.

This query to create the table UDF using SOAP UDFs was constructed manually, You can also use WebSphere Studio Application Developer to generate this table function in a much easier and less error prone way. For more details, see 16.5, “Insurance application as a Web service requestor” on page 518.

15

Scenario with Application Developer tools

So far, the scenario files were all created manually. In this chapter, we demonstrate how to develop these same files with various Application Developer tools.

15.1 Sample scenario report

Let’s first discuss the following report that is provided as part of the scenario. We assume that you have installed the scenario files and run this report command:

```
C:\SG246994\Scenario>GetOrgReportRDBnode
```

The output of the report is shown in Example 15-1.

Example 15-1 Sample report: GetOrgReportRDBnode

```
<?xml version="1.0"?>
<!DOCTYPE Report SYSTEM "C:\SG246994\Scenario\orgreport.dtd">
<Owner ID="122">
  <Organization>
    <OrganizationName>International Business Machines</OrganizationName>
    <TradingName1>IBM</TradingName1>
  </Organization>
  <OwnerAddress>
    <OwnerStreet>999 Wally World Avenue</OwnerStreet>
  </OwnerAddress>
  <Insured>
    <InsuredID>556</InsuredID>
    <ActualPremium>222.25</ActualPremium>
    <InsuredName>
      <InsuredFirstName>Sam</InsuredFirstName>
      <InsuredLastName>Elliot</InsuredLastName>
    </InsuredName>
    <Planinfo>
      <Plan_type>239</Plan_type>
      <InsurancePlan>Health</InsurancePlan>
    </Planinfo>
  </Insured>
  <Insured>
    <InsuredID>987</InsuredID>
    <ActualPremium>244.75</ActualPremium>
    <InsuredName>
      <InsuredFirstName>Shrinivas</InsuredFirstName>
      <InsuredLastName>Kulkarni</InsuredLastName>
    </InsuredName>
    <Planinfo>
      <Plan_type>239</Plan_type>
      <InsurancePlan>Health</InsurancePlan>
    </Planinfo>
  </Insured>
</Owner>
```

This report is created using the IBM DB2 XML Extender.

Remember that the DB2 XML Extender is an extension of DB2 that provides the ability to store XML documents or generate XML documents from existing relational data. The DB2 XML Extender also provides new data types, functions, and stored procedures to manage XML data in DB2.

15.1.1 DAD file

If you examine the GetOrgReportRDBnode.bat command file that is used to generate this report, you can see the DB2 XML Extender command that is invoked:

```
db2 call db2xml.dxxgenxmlclob(db2xml.xmlclobfromfile
         ('C:\SG246994\Scenario\OrgReportRDBnode.dad'),0,'',?,?,?,?,?)
```

Notice the document access definition (DAD) file that this command is based on:

```
C:\SG246994\Scenario\OrgReportRDBnode.dad
```

Remember that a DAD file is an XML-formatted file that associates XML document structure with a DB2 database. It is used by the DB2 XML Extender to compose and decompose XML data into relational data.

Creating a DAD file to be used by DB2 XML Extender is the ultimate goal. But, before you can do that successfully, you must know:

- The tables to be used
- The join conditions linking those tables
- The table columns to be used

Let’s start by analyzing the OrgReportRDBnode.dad file.

If you open the file, you see the tables and the join conditions (Example 15-2 on page 456).

Example 15-2 Root element: RDB_node

```
<RDB_node>
  <table name="ITSO.ORG"/>
  <table name="ITSO.OWNER"/>
  <table name="ITSO.P_TYPE"/>
  <table name="ITSO.POLICY"/>
  <table name="ITSO.INSURED"/>
  <condition>
    ITSO.OWNER.OWNER_ID=ITSO.ORG.OWNER_ID AND
    ITSO.ORG.OWNER_ID=ITSO.POLICY.OWNER_ID AND
    ITSO.P_TYPE.POLICY_TYPE_ID=ITSO.POLICY.POLICY_TYPE_ID AND
    ITSO.POLICY.INSURED_ID=ITSO.INSURED.INSURED_ID
  </condition>
</RDB_node>
```

If you examine the DAD file further, you notice the linkage of table columns to XML elements or attributes as shown in Table 15-1.

Table 15-1 XML to table column mapping

XML Element/Attribute	Table	Column
ID	ITSO.ORG	OWNER_ID
OrganizationName	ITSO.ORG	NAME
TradingName1	ITSO.ORG	TRADING_NAME1
OwnerStreet	ITSO.OWNER	STREET1
InsuredID	ITSO.POLICY	INSURED_ID
ActualPremium	ITSO.POLICY	ACTUAL_PREMIUM
InsuredFirstName	ITSO.INSURED	FIRST_NAME
InsuredLastName	ITSO.INSURED	LAST_NAME
Plan_type	ITSO.P_TYPE	POLICY_TYPE_ID
InsurancePlan	ITSO.P_TYPE	PLAN_NAME

Next, you should take note of all of the XML elements that have as an attribute: `multi_occurrence="YES"`. They are: `Insured`, `InsuredName`, and `Planinfo`.

The presence of these attributes indicates to the DB2 XML Extender that the marked elements can repeat (that is, they can have multiple rows). The method that is used to include these attributes uses the required DTD file, which describes the schema of the output XML file generated be DB2 Extender.

We have not yet mentioned that to use the Application Developer tools to create a DAD file, you must have already created a DTD file. We will elaborate on this a bit later.

Finally, notice that there is a condition on the XML element: `TradingName1`. Here is the condition:

```
<condition>trading_name1 = 'IBM'</condition>
```

Restriction: Although this is valid from the standpoint of the DB2 XML Extender, the Application Developer tool used to create the RDB table to XML mapping does not support conditions at non-root levels. In other words, if you need such a condition, you will have to manually edit the generated DAD file.

So, now that we are finished with the analysis of the DAD file already included with the scenario files, let's use the Application Developer tools to recreate the same DAD file.

15.1.2 DTD file

The first step is to create a DTD file describing the structure of the XML file that DB2 XML Extender is to output.

If you use the XML file above and reverse engineer it, the DTD file shown in Example 15-3 describes the XML output.

Actually, there is one aspect of the final DTD file that is not clear from looking at the XML output presented above. You cannot discern that the *InsuredName* and the *Planinfo* elements are repeating. That is because there is not enough data in the scenario database to make that clear (for example, there are no multiple names for insured).

Example 15-3 OrgReport.dtd

```
<?xml version='1.0' encoding="UTF-8"?>
<!ELEMENT ActualPremium (#PCDATA)>
<!ELEMENT InsurancePlan (#PCDATA)>
<!ELEMENT InsuredFirstName (#PCDATA)>
<!ELEMENT InsuredID (#PCDATA)>
<!ELEMENT InsuredLastName (#PCDATA)>
<!ELEMENT OrganizationName (#PCDATA)>
<!ATTLIST Owner ID CDATA #IMPLIED>
<!ELEMENT OwnerStreet (#PCDATA)>
```

```
<!ELEMENT Plan_type (#PCDATA)>
<!ELEMENT TradingName1 (#PCDATA)>
<!ELEMENT Owner (Organization,OwnerAddress,Insured+)>
<!ELEMENT Organization (OrganizationName,TradingName1)>
<!ELEMENT OwnerAddress (OwnerStreet)>
<!ELEMENT Insured (InsuredID,ActualPremium,InsuredName+,Planinfo+)>
<!ELEMENT InsuredName (InsuredFirstName,InsuredLastName)>
<!ELEMENT Planinfo (Plan_type,InsurancePlan)>
```

You can use Application Developer to help you create a DTD file using these steps:

1. Run the GetOrgReportRDBnode.bat file to create an output file:

   ```
   GetOrgReportRDBnode.bat >OrgReportRDBnode.output
   ```

2. Edit the result file (OrgReportRDBnode.output) and remove the front and back to leave only the result XML file (see Example 15-1 on page 454). Save the file as OrgReportRDBnode.xml.
3. Import the XML file into an Application Developer project.
4. Select the XML file and click **Generate -> DTD** (context).
5. Edit the DTD to make the InsuredName and Planinfo elements repeat.

15.1.3 DAD file

Once you have the DTD file, you can use the RDB to XML Mapping wizard (see 12.2.9, “Mapping the relational table to XML” on page 365) to create an RMX file. The RDB to XML Mapping editor is designed to work in conjunction with the DB2 XML Extender. It simplifies development tasks by:

- Providing a visual interface to easily define mappings between relational data and XML elements and attributes.
- Providing automatic generation of DAD files that would normally need to be coded by hand and could be quite error prone.
- Providing automatic generation of a test harness on multi-platforms that can be used to enable the DB2 database for use by the DB2 XML Extender, as well as store and retrieve XML files from relational data.

The RMX file can then be used to generate a DAD file by selecting **Generate DAD** (see 12.2.11, “Generating a DAD file” on page 370).

Now that we have a high-level overview of the process of creating a DAD file with Application Developer, let’s do it.

15.2 Using Application Developer for the scenario

In this section, we use Application Developer to recreate the files for the `GetOrgRDBnode` report.

15.2.1 Creating project and folders

We have to prepare a project and import the underlying files to prepare the DTD that is used for the report.

Creating the project

First, we define a Simple project called `InsuranceScenario`:

1. Open the XML perspective.
2. Select **File -> New -> Other -> Simple -> Project**.
3. Enter `InsuranceScenario` as the project name and click **Finish**.

Creating the folders

In the `InsuranceScenario` project, create four folders (select the project and click **New -> Folder** from the context menu):

- `database`—import of INSURA database schema
- `xmlsource`—XML files, DTD
- `xmlmapping`—mapping files (RMX, DAD)
- `xmltest`—generated files for testing

15.2.2 Connecting to the database and importing a local copy

Follow the instructions in 12.2.1, “Creating a database connection” on page 340 to import the tables into the `ItsoInsuranceWeb` project:

1. You can define a new connection to the INSURA database, or use the existing `ConInsura` connection. If the existing connection is disconnected, select the connection and click **Reconnect** (context).
2. Select the **ITSO** schema for import. Select the **InsuranceScenario/database folder** as the target.
3. Click **Finish** and confirm the prompts to create the folders.

15.2.3 Creating the DTD

We can create the DTD manually (Example 15-3 on page 457), or we can use an existing XML file to create the DTD:

1. Select the **xmlsource** folder and click **Import** (context). Select **File system** and locate the sample OrgReportRDBnode.xml file.
2. Import the file. Note that the Tasks lists shows an error because of the missing DTD that is referenced. Remove the DTD reference:

   ```
   <!DOCTYPE Report SYSTEM "C:\SG246994\Scenario\orgreport.dtd">
   ```

3. Select the **OrgReportRDBnode.xml** file and click **Generate DTD** (context). Change the output name to `OrgReport.dtd`. Accept the output folder (xmlsource).
4. The OrgReport.dtd file opens in the editor (Example 15-4). It is formatted very differently from Example 15-3 on page 457, but logically the same, with one exception. We have to change the `Insured` element to allow multiple `InsuredName` and `Planinfo`:

   ```
   <!ELEMENT Insured
      (InsuredID,ActualPremium,InsuredName+,Planinfo+)
   >
   ```

Example 15-4 Generated DTD (abbreviated)

```
<?xml version='1.0' encoding="UTF-8"?>
<!ELEMENT ActualPremium
   (#PCDATA)
>
<!ELEMENT InsurancePlan
   (#PCDATA)
>
<!ELEMENT Insured
   (InsuredID,ActualPremium,InsuredName+,Planinfo+)          <=== changed
>
<!ELEMENT InsuredFirstName
   (#PCDATA)
>
<!ELEMENT InsuredID
   (#PCDATA)
>
<!ELEMENT InsuredLastName
   (#PCDATA)
>
<!ELEMENT InsuredName
   (InsuredFirstName,InsuredLastName)
>
<!ELEMENT Organization
   (OrganizationName,TradingName1)
>
<!ELEMENT OrganizationName
   (#PCDATA)
```

```
>
<!ELEMENT Owner
   (Organization,OwnerAddress,Insured+)
>
<!--
   <annotation>
      <appinfo source='com.ibm.DDbEv2.Models.AbstractXModel.POSSIBLE'>
         <restriction base='string'/>
      </appinfo>
   </annotation>
-->
..........................
```

5. Optionally open the OrgReportRDBnode.xml file and add the reference to the DTD:

```
<?xml version="1.0"?>
<!DOCTYPE Owner SYSTEM "OrgReport.dtd">
<Owner ID="122">
```

6. Save and close the files.

15.2.4 Creating the RDB to the XML mapping session

Before you can create your mappings, you have to create an RDB to XML mapping session. The information about this mapping session, including source tables, the target DTD file, any mappings created, and join conditions are stored in a file with a .rmx extension. This file is necessary for you to persist information about any mappings you have created so that you can easily reload or modify them in the future. The steps to create your initial RDB to XML mapping session are:

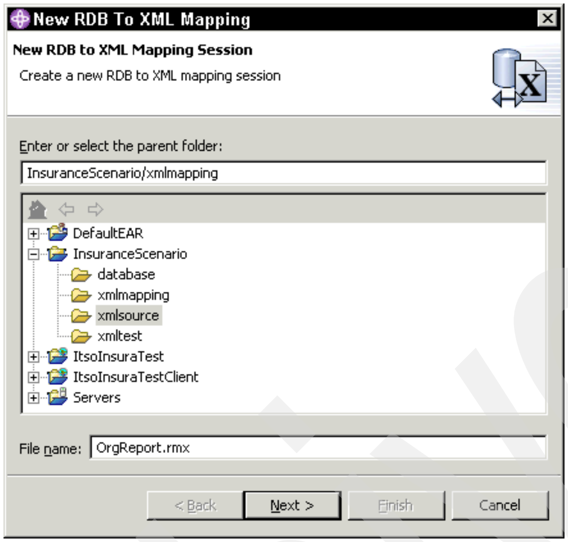
1. Start the RDB to XML mapping wizard.
2. Name the RDB to XML mapping session.
3. Choose the type of mapping.
4. Choose the source RDB tables.
5. Choose the target DTD file.
6. Choose the root element for the target DTD file.

Starting the RDB to XML Mapping wizard

Select **File -> New -> Other -> XML -> RDB to XML Mapping**. Click **Next**. The New RDB to XML Mapping Session wizard opens. There is also an icon in the XML perspective to start the wizard ().

Naming the RDB to XML Mapping session

On this page, you specify the name and location where you want the session file to be created (Figure 15-1):



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: InsuranceScenario/xmlmapping

File name: OrgReport.rmx

Figure 15-1 RDB to XML mapping: target folder and file

1. Select the InsuranceScenario/xmlmapping folder (you created it earlier).
2. Specify the file name as `OrgReport.rmx`.
3. Click **Next**.

Choosing the type of mapping

The RDB to XML Mapping tools provide two ways of mapping relational information to XML data, either through the RDB table mapping or the SQL statement mapping. Both of these methods of mapping use the DAD file to represent their mappings, but use different tags:

- **RDB table mapping**— maps RDB columns to XML data. Optional join conditions can be specified. This method is useful if you want to simply extract or store information in database tables. The DAD file created from an RDB table mapping can be used for both decomposing an XML file and storing it as relational data, and also for taking relational data and composing an XML file from it.
- **SQL statement mapping**—takes an SQL statement and enables you to map the result columns from executing an SQL statement to XML attributes and elements. This method is useful if you have complex queries that you would like to represent as XML data. These SQL statements must have been

created using the SQL wizard or SQL Builder (see “Creating an SQL statement” on page 346).

The key difference between the RDB table mapping and SQL statement mapping is that the SQL statement map can only be used for taking relational data and composing an XML file from it, not the reverse.

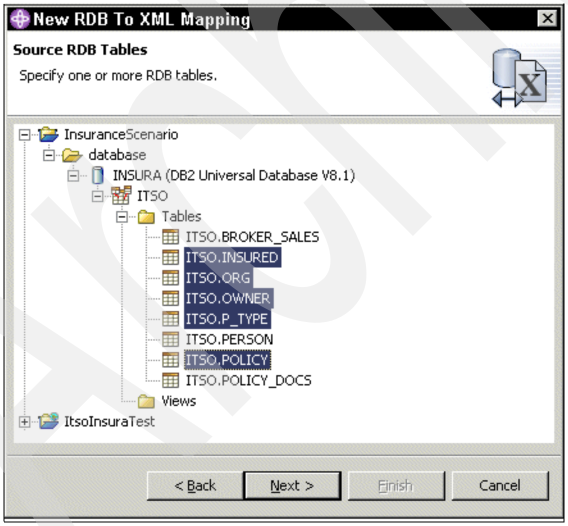
For the scenario you will be working with an RDB table mapping. Select **RDB table to XML mapping** and click **Next**.

Choose the source RDB tables

In this page, you select the tables that you want to map columns from. For the scenario, you are interested in these tables:

- `ITSO.INSURED`
- `ITSO.ORG`
- `ITSO.OWNER`
- `ITSO.P_TYPE`
- `ITSO.POLICY`

Expand the InsuranceScenario project and select the required tables (Figure 15-2).



New RDB To XML Mapping

Source RDB Tables

Specify one or more RDB tables.

Figure 15-2 RDB to XML mapping: table selection

Selecting the target DTD file

Expand the InsuranceScenario project and select the DTD file:

```
InsuranceScenario/xmlsource/OrgReport.dtd
```

Selecting the root element for the target DTD file

Here, you have to select the root element of the target DTD file. This will be used to allow you to represent the DTD file in an XML format, so that you can map RDB table columns to particular elements and attributes of an XML document:

1. Select **Owner** in the Root element list.
2. Click **Finish**.

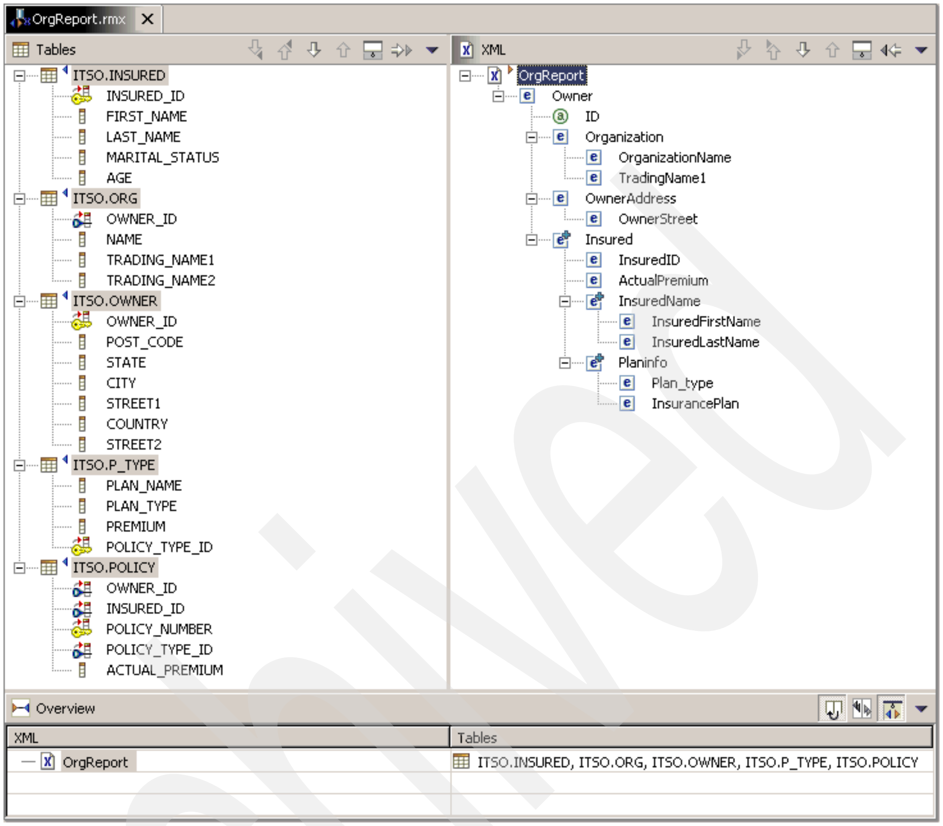
You have now created an RDB to XML mapping session. The session is opened in the RDB to XML mapping editor.

15.2.5 Using the RDB to XML mapping editor

The RDB to XML mapping editor is where you do most of the work:

- Create mappings for the RDB columns to XML elements and attributes.
- Specify join conditions for the source tables.
- Generate a DAD file and a test harness.

In the mapping editor, you see two panes (Figure 15-3 on page 465). The left pane (Tables) shows the tables, expanded with column, and the right pane (XML) shows the XML structure according to the DTD, with `Owner` as the root element (expand the structure to see all the elements).



XML	Tables
OrgReport	ITSO.INSURED, ITSO.ORG, ITSO.OWNER, ITSO.P_TYPE, ITSO.POLICY

Figure 15-3 RDB to XML mapping tool

Besides the RDB to XML Mapping view, two other views are available:

- **Overview** view, which shows a summary view of the XML document as well as everything that has been mapped to the XML document attributes and elements.
- **Outline** view, which shows all of the current mappings. If the Outline view is not open, select **Window -> Show View -> Outline**, then move the view to the bottom left, for example.

Creating mappings

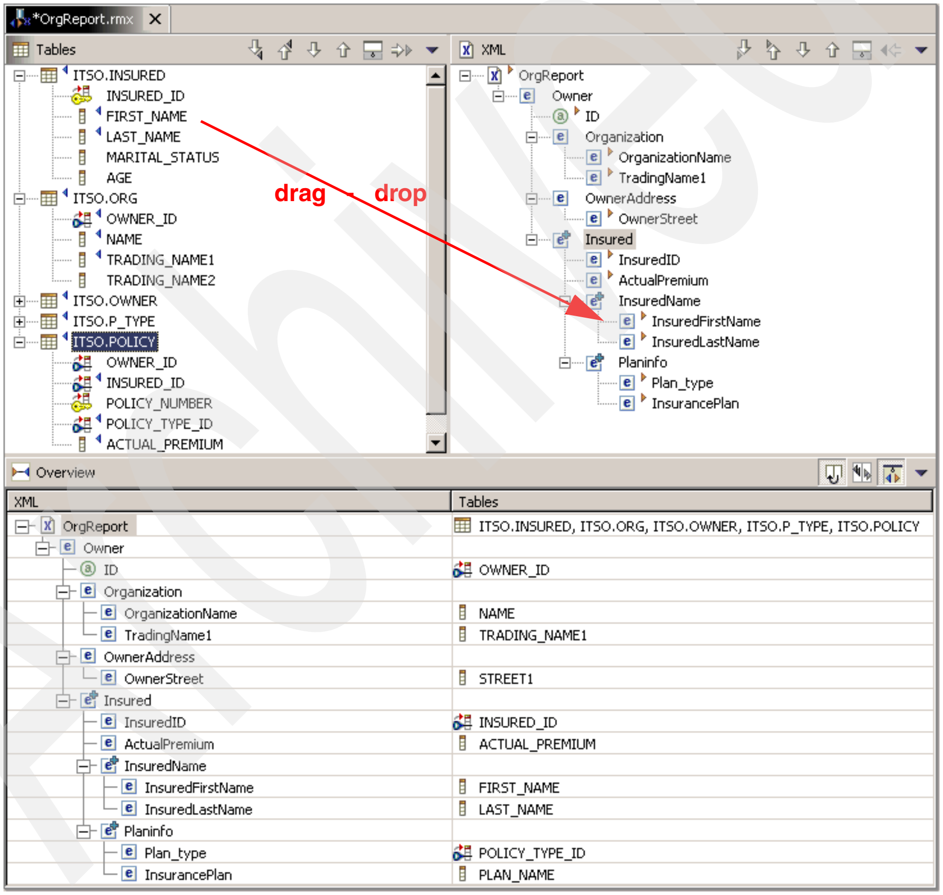
In the RDB to XML Mapping view, you can define mappings between relational data and XML elements and attributes. To create a mapping between a column and an element, you have two choices:

- Select a column in the left pane, an element or attribute in the right pane, and click **Create Mapping** (context menu in either pane).

- Select a column in the left pane and drag it over the element or attribute in the right pane. Release the cursor when an arrow appear. You can also drag from right to left.

Completed mappings are indicated with a small arrow icon by the column and element, and they also appear in the Overview and Outline views. When you generate a DAD file, this mapping information will be reflected in it and can be used by the DB2 XML Extender to compose or decompose XML files to relational data.

Perform the mappings as shown in Figure 15-4. If you make a mistake, select the mapping in the Overview and click **Remove Mapping** (context).



XML	Tables
OrgReport	ITSO.INSURED, ITSO.ORG, ITSO.OWNER, ITSO.P_TYPE, ITSO.POLICY
Owner	
ID	OWNER_ID
Organization	
OrganizationName	NAME
TradingName1	TRADING_NAME1
OwnerAddress	
OwnerStreet	STREET1
Insured	
InsuredID	INSURED_ID
ActualPremium	ACTUAL_PREMIUM
InsuredName	
InsuredFirstName	FIRST_NAME
InsuredLastName	LAST_NAME
Planinfo	
Plan_type	POLICY_TYPE_ID
InsurancePlan	PLAN_NAME

Figure 15-4 RDB to XML mapping complete

- `ITSO.ORG.OWNER_ID` column <==> `ID` XML attribute under `Owner`
- `ITSO.ORG.NAME` <==> `OrganizationName` under `Organization`
- `ITSO.ORG.TRADING_NAME1` <==> `TradingName1` under `Organization`
- `ITSO.OWNER.STREET1` <==> `OwnerStreet` under `OwnerAddress`
- `ITSO.POLICY.INSURED_ID` <==> `InsuredID` under `Insured`
- `ITSO.POLICY.ACTUAL_PREMIUM` <==> `ActualPremium` under `Insured`
- `ITSO.INSURED.FIRST_NAME` <==> `InsuredFirstName` under `Insured`
- `ITSO.INSURED.LAST_NAME` <==> `InsuredLastName` under `Insured`
- `ITSO.P_TYPE.POLICY_TYPE_ID` <==> `Plan_type` under `Insured`
- `ITSO.P_TYPE.PLAN_NAME` <==> `InsurancePlan` under `Insured`

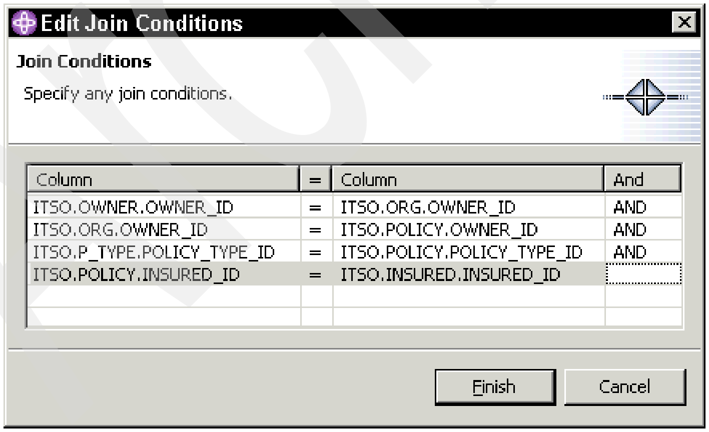
Specifying join conditions

Because you have more than one table, you have to specify join conditions for the tables. A join condition is used to specify which column in a table is to be matched with a column in another table. If a join condition between two tables is not specified, a result set will be returned containing all possible combinations of the two tables and this may return incorrect results.

If you try to generate a DAD file, and you have more than one source table and no join conditions, a warning message will appear.

Here are the join conditions that have to be entered (Figure 15-5):

- `ITSO.OWNER.OWNER_ID=ITSO.ORG.OWNER_ID`
- `ITSO.ORG.OWNER_ID=ITSO.POLICY.OWNER_ID`
- `ITSO.P_TYPE.POLICY_TYPE_ID=ITSO.POLICY.POLICY_TYPE_ID`
- `ITSO.POLICY.INSURED_ID=ITSO.INSURED.INSURED_ID`



Column	=	Column	And
ITSO.OWNER.OWNER_ID	=	ITSO.ORG.OWNER_ID	AND
ITSO.ORG.OWNER_ID	=	ITSO.POLICY.OWNER_ID	AND
ITSO.P_TYPE.POLICY_TYPE_ID	=	ITSO.POLICY.POLICY_TYPE_ID	AND
ITSO.POLICY.INSURED_ID	=	ITSO.INSURED.INSURED_ID	

Figure 15-5 RDB to XML mapping: join conditions

To specify join conditions, follow these steps:

1. Select **Mapping -> Edit Join Conditions** or use the icon.
2. Select the first row and double-click it in the first column. Select the first column name from the pull-down. An "=" is populated in the second column (you cannot edit this value, it is the only possible value).
3. Double-click in the third column. Select the second column name from the pull-down.
4. When you add more join conditions, the `AND` column is populated as well.

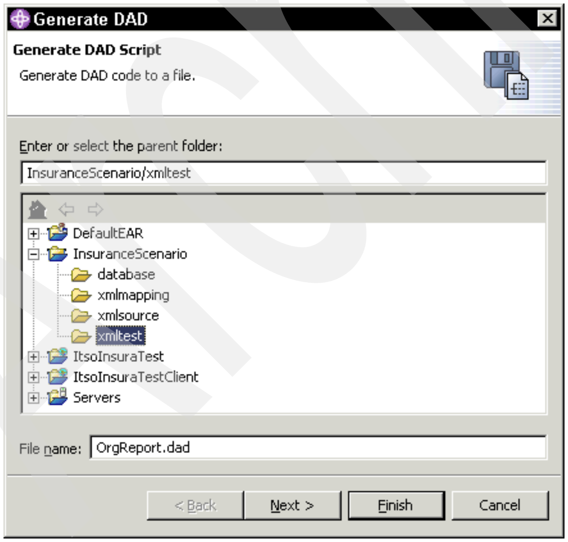
Once the appropriate join conditions have been created, click **Finish**. Save the OrgReport.rmx file.

Generating the DAD file and test harness

After all of the mappings have been created and all of the join conditions specified, it is time to generate a DAD file.

Select **Mapping -> Generate DAD** or use the icon to start the wizard.

You must specify the location and name of the DAD file to be generated (Figure 15-6). Select the **InsuranceScenario/xmltest** folder and set the name as `OrgReport.dad`.



Generate DAD

Generate DAD Script

Generate DAD code to a file.

Enter or select the parent folder: InsuranceScenario/xmltest

File name: OrgReport.dad

Figure 15-6 Generate DAD: location and name

If you were only interested in generating a DAD file, you could click **Finish** and generate the file. However, to specify an enclosing tag and generate a test harness, click **Next**.

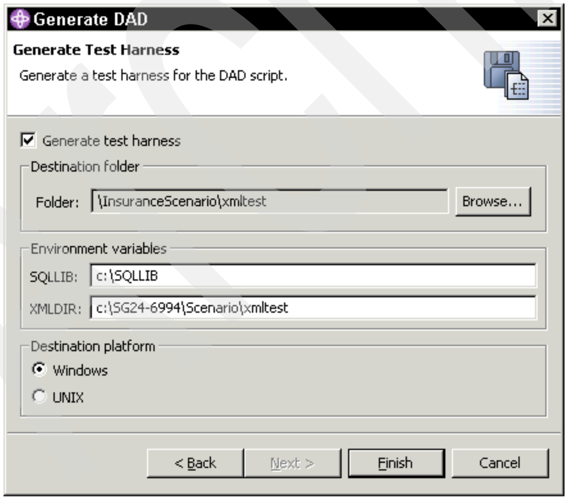
Enclosing tag

Specifying an enclosing tag is an option that allows you to specify a tag to enclose the entire document. This is useful in the situation where you are composing an XML file, and the top element of an XML document contains PCDATA or an attribute, and the value of this PCDATA or attribute can have multiple values. In this case, multiple XML files may be generated as output, since we have multiple top elements. The enclosing tag option allows a convenient way to specify a single tag that would be used to enclose the results, and consequently, only one XML document will be returned. In the case where this option is chosen, the store XML feature will not work with the generated DAD file. In our scenario, you do not specify an enclosing tag. Click **Next**.

Test harness

The purpose of the test harness is to provide operating system-specific command line files that will enable the DB2 database for the XML Extender, and process the DAD file in order to generate XML from relational data or to decompose XML data into relational data.

To generate the test harness, select **Generate Test Harness**, and complete the panel (Figure 15-7):



Generate Test Harness

Generate a test harness for the DAD script.

☑ Generate test harness

Destination folder

Folder: \InsuranceScenario\xmltest

Environment variables

SQLLIB: c:\SQLLIB

XMLDIR: c:\SG24-6994\Scenario\xmltest

Destination platform

● Windows

○ UNIX

Figure 15-7 Generate DAD: test harness

1. In the Destination Folder field, click **Browse** to locate the folder:

   ```
   \InsuranceScenario\xmltest
   ```

   Now you provide the environment variable values that contain specific path information about the destination machine where the test harness will be executed. This does not necessarily have to be on the same machine or operating system that you are currently running.

2. In the SQLLIB field, type the location of the DB2 SQLLIB directory.
3. In the DB2 EXTENDER field, type the location of the DB2 XML Extender directory. Generally, this directory contains dxx in it (that is, e:\dxx on Windows or /home/db2inst1/dxx on Linux). If you are working with V8.1 of DB2 UDB, this field does not appear as this directory is the same as the DB2 UDB directory (that is, DB2 XML Extender is included with V8.1 of DB2 UDB).
4. In the XMLDIR field, type the location where you want the generated XML file to be placed. This is the same location where `storeXML` will look for source XML files. This directory must exist before you execute the test harness. We use c:\SG246994\Scenario\xmltest.
5. Now select the destination platform. The destination platform is the platform on which the test harness files will actually be executed. This is very important because it determines OS-specific information in the generated test harness files, such as file separators and how environment variables are accessed.

   Click **Finish** to generate the test harness files.

The test harness files are generated in the directory specified in the Folder field. The DAD file is opened in an editor. The changes made to the generated DAD file are local to the DAD file, and are not reflected in the RDB to XML Mapping session file. As a result, the next time a regeneration of the DAD file occurs, any previous changes are not restored.

Specifying non-root conditions

Our DAD file requires a condition at a non-root element level. We have to add a condition element to the `TradingName1` element, as shown in Example 15-5 on page 471.

Example 15-5 Updating the DAD file with a non-root condition

```
<element_node name="TradingName1">
  <text_node>
    <RDB_node>
      <table name="ITSO.ORG"/>
      <column name="TRADING_NAME1" type="Character(50)"/>
      <condition>trading_name1 = 'IBM'</condition>
    </RDB_node>
    </text_node>
</element_node>
```

Generating files

The xmltest folder contains the generated files:

- OrgReport.dad—the DAD file
- readme.txt—instructions for the test harness
- setup.bat—sets up environment variables for the other commands
- retrieveXML.bat—generates an XML file from the database tables
- storeXML.bat—stores the contents of an XML file in the database tables
- updateDTD_REF.bat—stores the DTD file into DB2 for XML validation

15.2.6 Enabling DB2 for the XML Extender

Before you can store and retrieve XML documents from DB2 with the DB2 XML Extender, you have to enable the database for XML. As stated in the DB2 XML Extender documentation, this will:

- Create all the user-defined types (UDT) and user-defined functions (UDF).
- Create and populate control tables with the necessary metadata that the XML Extender requires.
- Create the `db2xml` schema and assign the necessary privileges.

The enabling of the database for XML is done for you in the test harness files, retrieveXML and storeXML. For more information, and to find out how to manually enable the database for XML, refer to the DB2 XML Extender documentation.

15.2.7 Using the test harness to execute the DAD files

Execution of the DAD file requires the DTD. Copy the OrgReport.dtd file from xmlsource into the xmltest folder (select the DTD file and click **Copy**, then select the **xmltest** folder and click **Paste**).

In Windows, open a Windows command prompt to execute the test harness.

Go to the directory where the test harness was generated:

```
<WSAD_WORKSPACE_ROOT>\InsuranceScenario\xmltest
```

Alternatively, export the xmltest folder to the c:\SG246994\Scenario directory. (Select the folder and click **Export**, select **File system**, and set the target directory.) Then go to the c:\SG246994\Scenario directory at the command prompt.

Make sure that the database has been started before executing any of the test harness files.

Running the setup command

Run setup.bat in the command prompt. This file sets up the environment variables that the other files use. After running setup.bat on Windows, an initialized DB2 shell appears. All other test harness files have to be run from this shell.

Running the updateDTD_REF command

Note: For our scenario, you do not have to run this command.

If you require XML validation or want the DTD file to be inserted into the `DB2XML.DTD_REF` table:

1. Edit the OrgReport.dad file and specify `YES` for `<validation>`. By default, its value is `NO`.
2. Run the updateDTD_REF file.

Running the retrieveXML command

The retrieveXML file is used to enable the DB2 database for DB2 XML Extender, as well as retrieve relational data and store it in an XML file following the mappings defined in the DAD file.

The result of enabling the database, as well as the resulting XML data, are displayed to the screen. The resulting XML data is also be written to a file in the XMLDIR directory you specified, and will have the same name as the DTD file but with a .xml extension. In our example, this file will be called OrgReport.xml. If multiple XML files are returned, only the first one will be written to XMLDIR.

The output file is generated as:

```
c:\SG246994\Scenario\xmltest\OrgReport.xml
```

Running the storeXML command

Note: For our scenario, you do not have to run this command.

The storeXML file is used to enable the DB2 database for DB2 XML Extender and store the data of the XML elements and attributes in tables.

Be careful, because the command deletes all the rows in the tables before it inserts the data from the XML file.

This command takes the input XML file from the XMLDIR directory. It looks for a file with the same name as your session file, but with a .xml extension. In this example, c:\SG246994\Scenario\xmltest\OrgReport.xml.

To validate that the store worked correctly:

- Check the database manually for the data that you expect to be there, or
- Run the `retrieveXML` command, copy the file for later (renamed), run the `storeXML` command with the output file, then run the retrieve again and compare the resulting XML file to the original file.

16

DB2 Web services and XML with Application Developer

In this chapter, we create DB2 Web services-based applications for a hypothetical insurance company called ITSO Insurance, Inc. We do not attempt to cover the full business processes used by a real insurance company. Our scenario is very simple, and its only purpose is to illustrate some of the ways in which you may incorporate XML documents in a database-oriented environment, using DB2, DB2 XML Extender, and DB2 Information Integrator.

16.1 Insurance scenario

The Insurance scenario was developed to demonstrate how multiple features of WebSphere Studio are combined in an end-to-end development scenario to build an e-business application. To develop this application, you will rapidly employ DB2 capabilities both as Web service provider and requestor using Application Developer XML and Web service tools.

16.1.1 Business context

As explained in Chapter 13, “Worked scenario” on page 393, ITSO Insurance, Inc. (hereafter referred to as ITSO Insurance, or just the company) is a well established insurance company that mainly works with their self-employed agents that only work for ITSO Insurance, Inc. To expand their horizon, ITSO Insurance, Inc. has decided to also sell insurance policies through freelance insurance brokers.

The first application ITSO Insurance, Inc. decided to implement is to provide insurance brokers with insurance quotes for certain types of policies that ITSO Insurance, Inc. wants to sell through the broker channel. This is implemented as a Web service.

ITSO Insurance, Inc. invests the money it gets from insurance premiums in stocks. The online quotes and headline information for a certain stock can be obtained by invoking a free Web service available in third-party Web sites. Therefore, before the company decides to buy a certain stock, the investment specialist can invoke these Web services to get some recent background information about the stock they think is a good investment.

16.1.2 Technical implementation overview

The technical implementation of the Insurance scenario is composed of four business cases:

1. Create an Insura Web service to provide broker applications with required information in XML format. Here, we are demonstrating DB2 capabilities as a Web service provider. We also publish the Web service to the IBM UDDI Business Test Registry.
2. Create and publish a Web page where brokers can get a policy quote for a given customer. The policy quote takes into account the insurance history of the customer (XML file), which can be reached either in the local file system or over the Internet. This part serves as a showcase, demonstrating DB2 XMLwrapper’s ability to provide a relational view of XML files.

3. Provide ITSO Insurance, Inc.'s investment specialists with a Web page where they can receive relevant information (quote and news headlines) on a certain stock, showing DB2 as a Web service consumer.
4. Create a broker Web page serving as a client to the previously published Insura Web service and provide a user with a humanly readable interface to the Web service response. Here, we demonstrate XSL transformation of SOAP XML to HTML format.

The sample application that we develop consists of the Insura Web service application and the broker client application running on logically different application servers. The Insurance application server has an XML-enabled DB2 database as a back end, runs a DADX Web service, and provides users with data access Web pages. The broker application server provides a front end to the Insura Web service. Figure 16-1 shows a logical implementation diagram of the complete application.

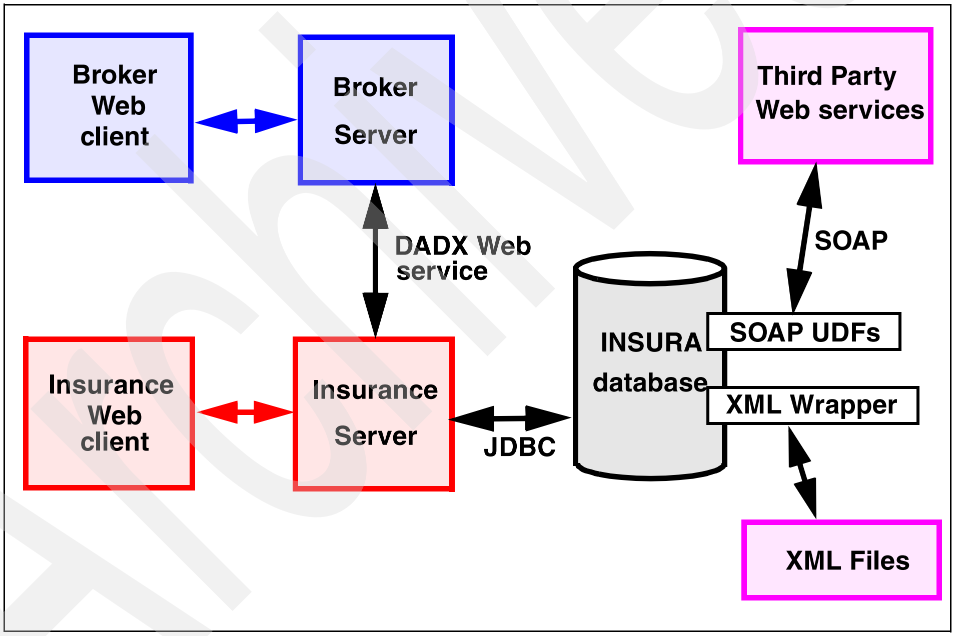


Figure 16-1 Insurance scenario: high-level implementation diagram

As was discussed in Chapter 12, “XML and database tools in Application Developer” on page 337, Application Developer provides a set of tools for discovering, creating, and publishing Web services that are created from a number of artifacts such as JavaBeans, DADX files, Enterprise JavaBeans, and URLs.

The Document Access Definition Extension (DADX) Web service enable you to wrap DB2 XML Extender Document Access Definition (DAD) files or regular SQL statements inside a Web service. DADX is an XML document that specifies how to create a Web service using a set of operations that are defined by DAD documents and SQL statements. The DADX file defines the operations available to the DADX runtime environment, and the input and output parameters for the SQL operation.

Web tools assist you in developing Web applications that you can configure as a Web service. Web applications are developed in a Web project, and server tools enable you to use the unit test environment to test and deploy your Web services.

Implementation of the Web services requestor part of the Insurance application is based on Application Developer tools capable of generating DB2 Web service requestor user-defined functions (UDFs).

16.1.3 Preparing for the insurance application

Before you can develop the sample application, you have to perform the following steps:

- Install the prerequisite software on your workstation
- Create the INSURA database

The following sections provide details about each of these tasks.

System prerequisites

You need the following software installed on your workstation:

- DB2 Universal Database™ (UDB) for Windows V8.1 FixPak 2, or V7.2 FixPak 7 or later
- WebSphere Studio Application Developer 5.0 or 5.1
- WebSphere Application Server 5.0 or 5.0.2 for deployment of the application

See Appendix A, “Installation” on page 545 for detailed instructions.

Creating the INSURA database

The INSURA database must be created and enabled. See “Setting up the INSURA database” on page 402 for detailed instructions.

Once you have completed the preparation tasks, you are ready to develop the Insurance application.

16.2 Insurance application as Web service provider

In this section, we are implementing the business case #1 of the Insurance scenario in two ways:

- Starting from a prearranged DADX file
- Creating an SQL statement and the DADX file

To create the DB2 Insura Web Services using Application Developer tooling, we perform these steps:

1. Set the Web services interoperability compliance level.
2. Create the Web project that will contain the Insurance application.
3. Create a DADX group within the Web project.
4. Create SQL statements that query the INSURA database for policy quotes and business amount information.
5. Create a proxy and a test client for the Insura Web Service.
6. Publish a business entity and a Web service to a UDDI registry.

We use Application Developer wizards wherever possible.

The Insura Web Services respond to requests for policy quotes and business amount information from the broker server. They implement a DADX Web services to query the INSURA database and return the results to the broker application as a SOAP-encoded XML document.

16.2.1 Setting the Web services interoperability compliance level

WS-I is an organization designed to promote Web service interoperability across platforms, operating systems, and programming languages. For more information on WS-I, refer to their Web site:

http://www.ws-i.org/

This site contains resources such as an overview of Web services interoperability, usage scenarios, and specifications.

The WS-I Basic Profile is a outline of requirements to which WSDL and Web service protocol (SOAP/HTTP) traffic must comply in order to claim WS-I conformance. The Web services WS-I validation tools currently support WS-I Basic Profile 1.0. To view the specifications, refer to the WS-I Web site and under Resources, select **Documentation**.

Depending on the type of Web service being created, you may or may not want your Web service to comply with the WS-I Basic Profile. WebSphere Studio allows you to set your level of compliance. The default level of compliance is to generate a warning if a non-complaint Web service option is selected.

By default, the level of WS-I compliance is set to `Suggest`, which means that any non-compliant choices that are selected generate a warning dialog box, but you will be able to continue.

The insurance sample may generate a non-compliant Web service, depending on the value of the group property *Use document style* (see Figure 12-45 on page 377). If you get a warning message, click **Ignore**. This warning message currently occurs even if you set Use document style to `true`, although the generated WSDL file is WS-I compliant.

To change the level of WS-I compliance:

1. Select **Window -> Preferences**.
2. Expand Web Services and select **WS-I Compliance**.
3. Select **Ignore WS-I compliance**.

16.2.2 Creating the Web project

Web services are created within a Web project. All the resources required by the Web services, such as your Web settings and WSDL files, must exist within the project. To create a Web project, follow these steps in Application Developer:

1. Switch to the Web perspective (**Window -> Open Perspective -> Other -> Web**).
2. From the workbench, click **File -> New -> Dynamic Web Project**.
3. Type `ItsoDB2XMLInsuraWeb` in the Project name field. Select **Configure advanced options** and click **Next**.
4. Type `ItsoDB2XMLInsura` in the EAR project field. Leave the J2EE level as 1.3. Click **Finish**.

The ItsoDB2XMLInsuraWeb and ItsoDB2XMLInsura projects are created and your Project Navigator view should look like Figure 16-2 on page 481. The enterprise application project stores the Web project as a WAR file embedded in an EAR file that can be exported to a server.

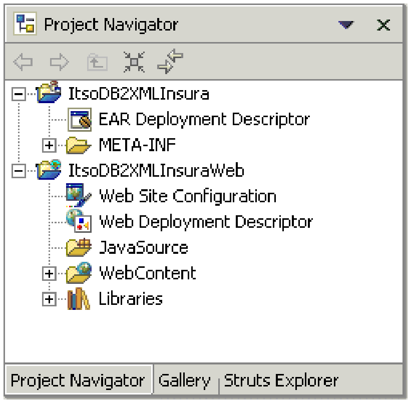


Figure 16-2 ItsoDB2XMLInsura enterprise application and Web project

16.2.3 Creating the DADX group

The DADX group contains connection and other information that is shared between DADX files. To create the DADX group:

1. In the Project Navigator, select **ItsoDB2XMLInsuraWeb**.
2. Select **File -> New -> Other -> Web Services** in order to display the various Web service wizards. Select the **Web Service DADX Group Configuration** wizard. Click **Next**.
3. Select **ItsoDB2XMLInsuraWeb** and click **Add group**. In the DADX group name text field, type `insuraGroup`. Click **OK**.
4. Expand ItsoDB2XMLInsuraWeb, select **insuraGroup**, and click **Group Properties**. In the DADX Group Property dialog (see Figure 12-45 on page 377):

   a. Change the DB URL to `jdbc:db2:insura`

   b. Enter the Context factory as: `com.ibm.websphere.naming.WsnInitialContextFactory`

   c. Enter the Datasource as: `jdbc/insura`

   d. Set Use document style to `true` to generate a WS-I compliant Web service

   e. Click **OK**

   Database connections for DADX Web services can be made using JDBC drivers or data sources. Data sources are tried first; JDBC drivers are used if the data source access fails.

5. Click **Finish**. The DADX group is generated in the directory ItsoDB2XMLInsuraWeb\JavaSource\groups (Figure 16-3).

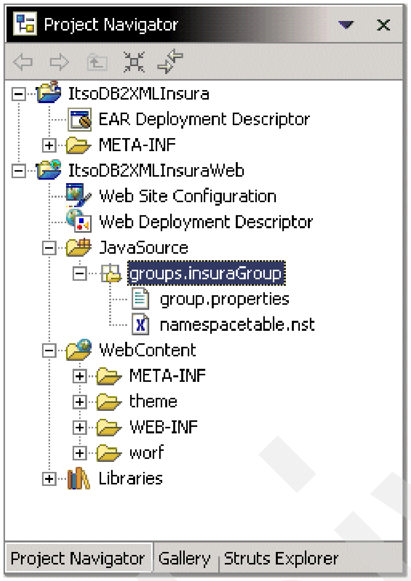


Figure 16-3 DADX insuraGroup in Project Navigator view

16.2.4 DADX file

The DAD Extension (DADX) file is an XML document that specifies how to create a Web service using a set of operations that are defined by DAD documents and SQL statements. The DADX file can be constructed from one or more SQL statements, stored procedures, XML Extender DAD files, or a combination of all three types.

In this section, we are going to generate the insura.dadx file containing the DB2 queries and a description of the required account parameter.

Important: You can either import the insura.dadx file, or generate your own by creating the necessary SQL queries and/or stored procedures and generating a DADX file from the results using the SQL from XML wizard.

- To import the insura.dadx file, follow the directions in 16.2.5, "Importing the DADX file" on page 483", then skip "16.2.6, "Creating the SQL query" on page 483" and "16.2.7, "Creating the DADX file" on page 490".
- To create the insura.dadx file, skip 16.2.5, "Importing the DADX file" on page 483 and follow the directions in 16.2.6, "Creating the SQL query" on page 483 and 16.2.7, "Creating the DADX file" on page 490.

16.2.5 Importing the DADX file

To import the DADX file, follow this path:

- In the Project navigator view, select the **groups.insuraGroup** folder (under JavaSource) and click **File -> Import** to open the Import wizard.
- Click **File system** and **Next**.
- In the Directory text field, specify the following location of the DADX file. Use the **Browse** button if necessary:

```
c:\SG246994\Scenario\wsad
```

- Select only the **insura.dadx** file and click **Finish**. The file is imported into the ItsoDB2XMLInsuraWeb project.
- Now that you have imported the DADX file, continue the scenario with 16.2.8, "Creating the DADX Web services" on page 490.

16.2.6 Creating the SQL query

To create the SQL query, you perform these tasks:

- Define the database connection
- Define the SQL statement
- Test the query

The following sections provide details about each of these tasks.

Defining the database connection

First, you have to define the connection to the INSURA database:

1. Open the Data perspective (**Window -> Open Perspective -> Other -> Data**).

2. In the DB Servers view, select **New Connection** (context). The New Connection wizard opens.
3. In the Connection Name entry field, type `insura_connection`.
4. In the Database entry field, type `insura`.
5. In the Database vendor type field, ensure that the correct database driver is selected.
6. In the JDBC driver field, ensure that the correct JDBC driver is selected (**IBM DB2 APP DRIVER**).
7. In the Class location field, ensure that the path to your to your JDBC driver class (in db2java.zip) is correct.
8. Click **Finish**. A connection to the INSURA database is defined.

Your DB Servers view should resemble the following (Figure 16-4):

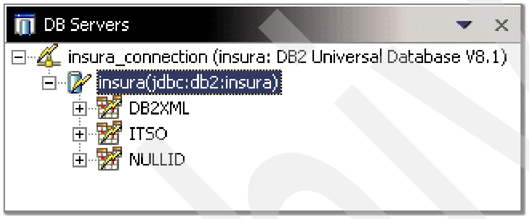


Figure 16-4 INSURA database connection

With the database connection defined, you have to copy the table definitions to the ItsoDB2XMLInsuraWeb project:

1. In the DB Servers view, expand insura_connection, select **insura(jdbc:db2:insura)** and **Import to Folder** (context). The Import to Folder dialog opens. Click **Browse** and select the **ItsoDB2XMLInsuraWeb** project.

   Note that you can also select only the **ITSO** schema and click **Import to Folder**. We only use the ITSO.xxxxxx tables for the SQL statement.
2. Click **OK**, then click **Finish**.

Click **Yes** to confirm the creation of the databases folder. If you expand the ItsoDB2XMLInsuraWeb project, you get a folder structure similar to the one shown in Figure 16-5 on page 485.

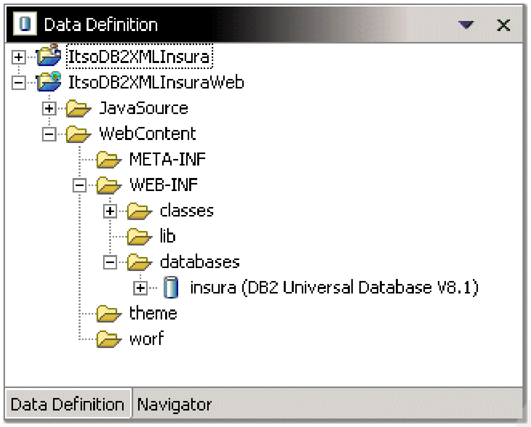


Figure 16-5 Database connection imported into the Web project

Defining the policyQuote SQL statement

There are two alternative ways of creating an SQL statement in Application Developer:

- SQL Statement Wizard—The SQL Statement Wizard is a guided dialog through a number of panels, with an SQL statement as the result.
- SQL Query Builder—The SQL Query Builder is an editor for an advanced user.

Both tools can be used to build an SQL statement. After using the SQL Statement Wizard, you can use the SQL Query Builder to update the SQL statement or, alternatively, you can build an SQL statement from scratch using SQL Query Builder.

For detailed instructions on how to use SQL Statement Wizard, refer to 12.2.4, “Creating an SQL statement” on page 346. For this example, we are developing our SELECT statements against the INSURA database using the SQL Query Builder.

We start by designing the policy quote query. Because our insurance brokers want to get a quote (a price) for a certain policy type, we return the insurance policy plan name, plan type, and the standard and actual premium for a selected type of policy. Basically, we construct this SQL statement:

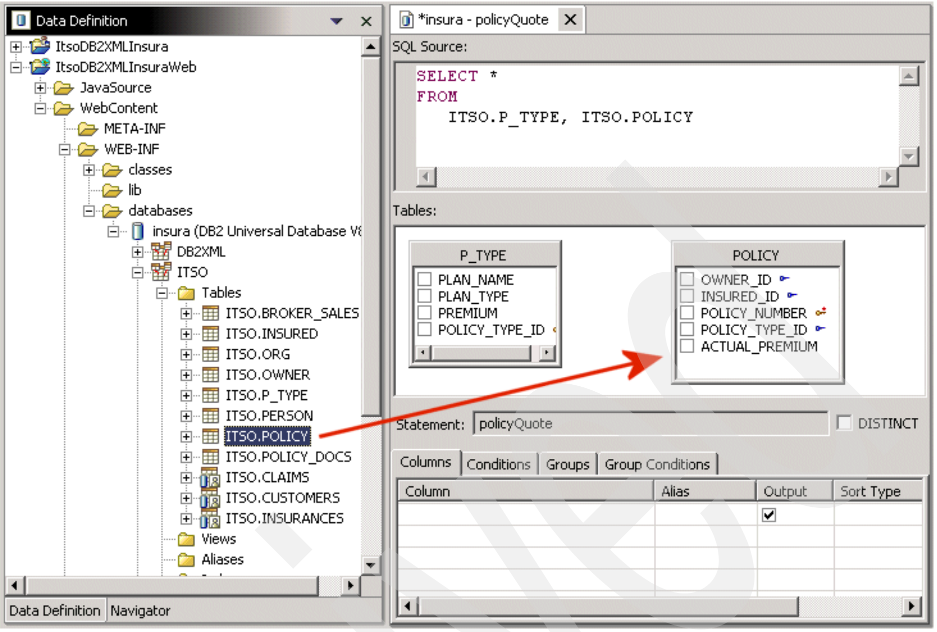
```
SELECT
   ITSO.P_TYPE.PLAN_NAME,
   ITSO.P_TYPE.PLAN_TYPE,
   ITSO.P_TYPE.PREMIUM,
   ITSO.POLICY.ACTUAL_PREMIUM
FROM
```

```
      ITSO.POLICY, ITSO.P_TYPE
   WHERE
      ITSO.POLICY.POLICY_TYPE_ID = ITSO.P_TYPE.POLICY_TYPE_ID
      AND ITSO.P_TYPE.PLAN_NAME = :planname
```

See the policyQuote.sql file in c:\SG246994\Scenario\wsad.

To define the SQL statement in Application Developer, follow these steps:

1. In the Data perspective Data Definition view, click **ItsoDB2XMLInsuraWeb -> WebContent -> WEB-INF -> databases -> insura database -> Statements** (folder).
2. Select **Statements** and **New -> Select Statement** (context).
3. Enter `policyQuote` as name and click **OK**. This launches the SQL Query Builder.
4. In the Outline view, or in the middle pane labeled Tables, select **Add Table** (context). The Add Table dialog box opens.
5. From the Table name list select **ITSO.P_TYPE**, then click **OK**. This specifies that the `policyQuote` SQL statement includes the P_TYPE table belonging to the `ITSO` schema. As you can see, the table has been added to the `SELECT` statement in the top pane.
6. You can also use the “drag-and-drop” technique to incorporate the POLICY table into the query (Figure 16-6 on page 487).



```
SELECT *
FROM
   ITSO.P_TYPE, ITSO.POLICY
```

P_TYPE
- □ PLAN_NAME
- □ PLAN_TYPE
- □ PREMIUM
- □ POLICY_TYPE_ID

POLICY
- □ OWNER_ID
- □ INSURED_ID
- □ POLICY_NUMBER
- □ POLICY_TYPE_ID
- □ ACTUAL_PREMIUM

Statement: policyQuote □ DISTINCT

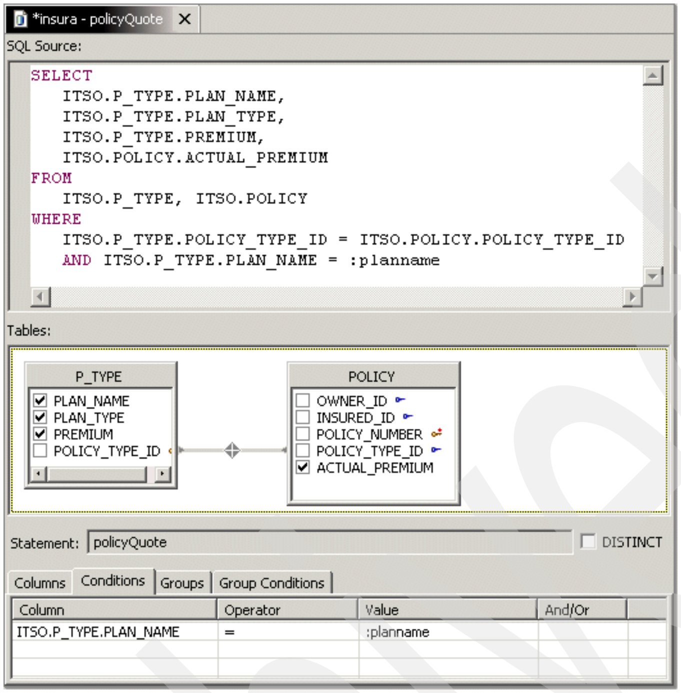
Column	Alias	Output	Sort Type
		☑	

Figure 16-6 Adding tables to the SQL query

7. Next, select the columns from each table using the check boxes. For the P_TYPE table select all columns except POLICY_TYPE_ID. For the POLICY table only select the **ACTUAL_PREMIUM** column. As you select the columns, the `SELECT` statement is updated in the top pane and the columns are added in the bottom pane.

8. Next, join the tables together. To join the tables, select the **POLICY_TYPE_ID** column in the P_TYPE table and drag it across to the corresponding column in the POLICY table. A link symbol is shown between the tables, and the SELECT statement is updated with the corresponding WHERE clause.

9. Finally, we want to add the a condition (`PLAN_NAME = :planname`). Use the Conditions tab in the bottom pane to add the condition using the drop-down menus, or type it directly into the SQL statement and the Conditions tab is updated (Figure 16-7 on page 488).



*insura - policyQuote

SQL Source:

```
SELECT
   ITSO.P_TYPE.PLAN_NAME,
   ITSO.P_TYPE.PLAN_TYPE,
   ITSO.P_TYPE.PREMIUM,
   ITSO.POLICY.ACTUAL_PREMIUM
FROM
   ITSO.P_TYPE, ITSO.POLICY
WHERE
   ITSO.P_TYPE.POLICY_TYPE_ID = ITSO.POLICY.POLICY_TYPE_ID
   AND ITSO.P_TYPE.PLAN_NAME = :planname
```

Tables:

P_TYPE
- ☑ PLAN_NAME
- ☑ PLAN_TYPE
- ☑ PREMIUM
- □ POLICY_TYPE_ID

POLICY
- □ OWNER_ID
- □ INSURED_ID
- □ POLICY_NUMBER
- □ POLICY_TYPE_ID
- ☑ ACTUAL_PREMIUM

Statement: policyQuote □ DISTINCT

Column	Operator	Value	And/Or
ITSO.P_TYPE.PLAN_NAME	=	:planname	

Figure 16-7 SQL Query Builder: adding conditions and finalizing query

Restriction: Do not use an underscore in a parameter (`:plan_name`). Creating a DADX Web services based on parametric SQL statements where the parameter name contains an underscore sign (_) fails in Application Developer V5.0 and 5.1.

Save the statement. You are prompted for the host variables; just click **Cancel** to dismiss the dialog.

For more detailed information how to use Application Developer SQL tools, please refer to *WebSphere Studio Application Developer V5 Programming Guide*, SG24-6957.

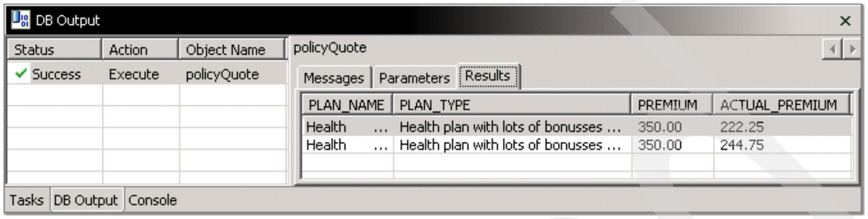
Testing the query

After creating an SQL query, you should test it to ensure that it returns the correct result. To test the query:

1. Select the statement in the Statements folder an click **Execute** (context) or select **SQL -> Execute** in the menu bar.

2. Enter `'Health'` as the value for the `:planname` variable in the Host Variable Values window, press **Enter** and then click **Finish** to execute the query.

The result of the query appears in the DB Output pane (Figure 16-8). If the query was successful, the Status column will indicate `Success`.



Status	Action	Object Name
✓ Success	Execute	policyQuote

PLAN_NAME	PLAN_TYPE	PREMIUM	ACTUAL_PREMIUM
Health ...	Health plan with lots of bonusses ...	350.00	222.25
Health ...	Health plan with lots of bonusses ...	350.00	244.75

Figure 16-8 Testing an SQL statement

Defining the brokerSales SQL statement

Now we are going to construct the second query that answers the question of how much business has been generated with us by the broker. This SQL statement will be further referred to as `brokerSales` and looks like this (brokerSales.sql file in c:\SG246994\Scenario\wsad):

```
SELECT
   ITSO.BROKER_SALES.BROKER_ID,
   SUM(ITSO.BROKER_SALES.ACTUAL_PREMIUM) AS Total_Premium_Sales
FROM
   ITSO.BROKER_SALES
WHERE
   ITSO.BROKER_SALES.BROKER_ID = :brokerid
GROUP BY
   ITSO.BROKER_SALES.BROKER_ID
```

To construct the `brokerSales` statement, use the SQL Query Builder. To create the `SUM` function, select **Build expression** in the Column drop-down menu of the Columns tab. Select **Function**, select **SUM**, **SUM(DECIMAL) -> DECIMAL**, then select the **ACTUAL_PREMIUM** column. Enter `Total_Premium_Sales` as the Alias. Define the grouping in the Groups tab by selecting the column.

Save the query and test it by entering `456` as the value for the `:brokerid` variable. If you have done everything correctly, you should get 11111.1 as the result.

Now that you have created the SQL queries, continue the scenario with “16.2.7, “Creating the DADX file” on page 490”.

16.2.7 Creating the DADX file

To create the DADX file, follow the instructions in 12.3.2, “Creating a DADX file from an SQL statement” on page 378:

1. In the Web perspective, select **File -> New -> Other -> Web Services -> DADX File**. Click **Next**. The Create DADX wizard opens.
2. Expand the ItsoDB2XMLInsuraWeb project and select both **policyQuote** and **brokerSales** statements, then click **Next**.
3. Click **Next** to bypass the Select DAD files page.
4. In the DADX generation page, type `insura.dadx` in the File name field.
5. Select **/ItsoDB2XMLInsuraWeb/JavaSource/groups/insuraGroup** as the output folder.
6. Click **Finish**.

The DADX file opens in the XML editor. Examine the file then close the editor. Note that both constructed SQL statements are incorporated in the newly created DADX file.

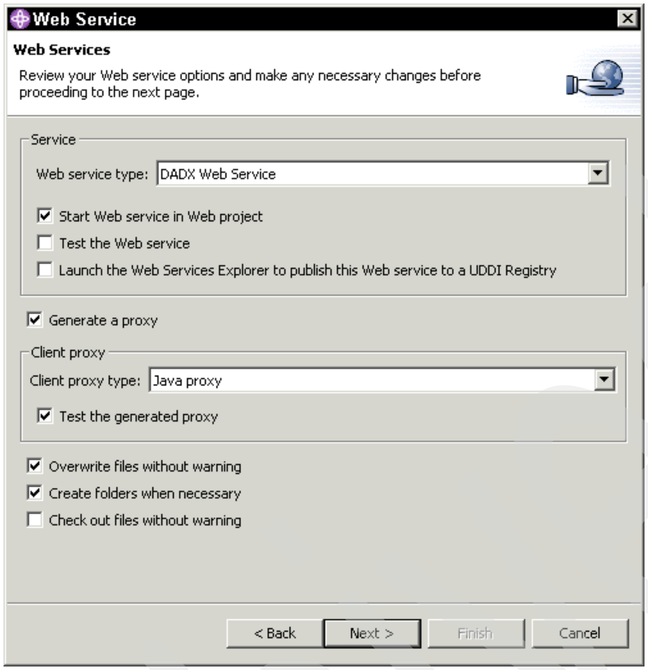
Now that you have generated the DADX file, continue the scenario with “16.2.8, “Creating the DADX Web services” on page 490”.

16.2.8 Creating the DADX Web services

We now implement the `policyQuote` and `brokerSales` information queries as DADX Web services. These Web services enable the Insurance application to retrieve information about a certain type of policy and a broker’s generated business from the INSURA database using a DB2 query. The results are returned to the Insurance server as a SOAP-encoded XML document (Figure 16-1 on page 477). The Insurance application uses a JSP file to display the corresponding records.

To create the Insura Web Services from the insura.dadx file:

1. In the Web perspective expand the `ItsoDB2XMLInsuraWeb` project and select the **JavaSource/groups.insuraGroup/insura.dadx** file.
2. Select **File -> New -> Other -> Web Services -> Web Service** and click **Next** to start the Web service wizard.
3. In the Web Services page of the wizard, ensure that **DADX Web service** is selected from the Web service type menu. Select **Start Web service in Web project**, **Generate a proxy, Overwrite file without warning** and **Test the generated proxy** (Figure 16-9 on page 491). Click **Next**.



Web Service

Web Services

Review your Web service options and make any necessary changes before proceeding to the next page.

Service

Web service type: DADX Web Service

☑ Start Web service in Web project

□ Test the Web service

□ Launch the Web Services Explorer to publish this Web service to a UDDI Registry

☑ Generate a proxy

Client proxy

Client proxy type: Java proxy

☑ Test the generated proxy

☑ Overwrite files without warning

☑ Create folders when necessary

□ Check out files without warning

Figure 16-9 Creating DADX Web Service: options

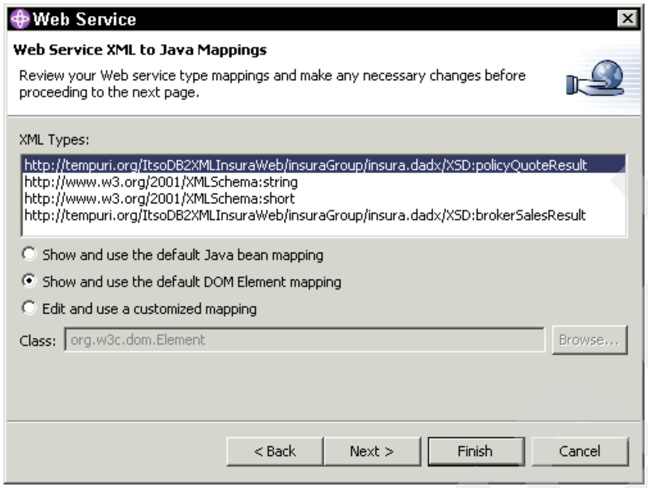
4. In the Service Deployment Configuration page, ensure that the server is set to `WebSphere v5.0 Test Environment`, **ItsoDB2XMLInsuraWeb** is selected as the Service Web project, and `ItsoDB2XMLInsuraWebClient` is set as the Client Web Project. Click **Next**.

5. In the Web Service DADX File Selection page, the **insura.dadx** file is preselected. Click **Next**.

6. The Web Service DADX Group properties page of the wizard is used to update your group properties. Ensure the DB URL field displays `jdbc:db2:insura`. Click **Next**.

7. In the Web Service Binding Proxy Generation page, select **soap binding**, leave the proxy as `proxy.soap.insuraProxy`, and select **Show mappings**. The client proxy provides a remote procedure call interface to your Web service. Click **Next**.

8. In the Web Service XML to Java Mappings page of the wizard (Figure 16-10 on page 492), select each entry and look at the mapping. Leave the JavaBean mapping for the two parameters, but select **Show and use the default DOM Element mapping** for the two results (`http://tempuri.org/ItsoDB2XMLInsuraWeb/...`). Click **Next**.



Web Service XML to Java Mappings

Review your Web service type mappings and make any necessary changes before proceeding to the next page.

XML Types:

http://tempuri.org/ItsoDB2XMLInsuraWeb/insuraGroup/insura.dadx/XSD:policyQuoteResult
http://www.w3.org/2001/XMLSchema:string
http://www.w3.org/2001/XMLSchema:short
http://tempuri.org/ItsoDB2XMLInsuraWeb/insuraGroup/insura.dadx/XSD:brokerSalesResult

○ Show and use the default Java bean mapping
● Show and use the default DOM Element mapping
○ Edit and use a customized mapping

Class: org.w3c.dom.Element Browse...

Figure 16-10 Creating DADX Web services: XML to Java mapping

9. In the Web Service SOAP Binding Mapping Configuration page, review the SOAP binding mapping configurations. Click **Next** to accept the default values.
10. In the Web Service Test page, ensure that **Test the generated proxy** and **Run test on server** are selected and that `Web service sample JSPs` appears in the Test facility text field (note that the Universal Test Client is not well suited to test Web services that return a DOM element). Leave the target folder for the test client as `sample/insura`. Click **Finish**.

It may take a few minutes for the Web services to be generated. The Web services are deployed to the WebSphere Application Server Test Environment, the server is started and an internal Web browser is launched to demonstrate the sample application.

Now that you have created the Insura Web Services, deployed and launched them on the built-in Application Server, you may want to test the methods of the Insurance DADX Web services, `policyQuote` and `brokerSales`.

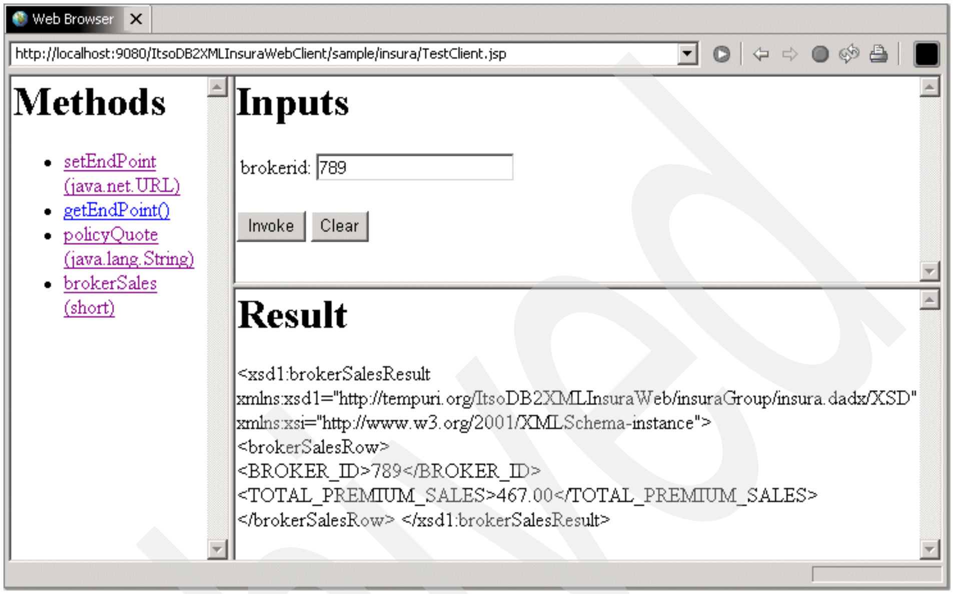
16.2.9 Testing the DADX Web services

The browser is launched with the test client. You can also manually start the test client by selecting the **TestClient.jsp** in the sample/insura folder of the client project (ItsoDB2XMLInsuraWebClient).

To examine the methods of the sample Web application:

1. Select the **brokerSales** method in the Methods pane.

2. In the *brokerid* field of the Inputs pane, type `456` or `789` and click **Invoke**. The Result pane displays the total of the broker's generated business (Figure 16-11).



http://localhost:9080/ItsoDB2XMLInsuraWebClient/sample/insura/TestClient.jsp

brokerid: 789

```
<xsd1:brokerSalesResult
xmlns:xsd1="http://tempuri.org/ItsoDB2XMLInsuraWeb/insuraGroup/insura.dadx/XSD"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<brokerSalesRow>
<BROKER_ID>789</BROKER_ID>
<TOTAL_PREMIUM_SALES>467.00</TOTAL_PREMIUM_SALES>
</brokerSalesRow> </xsd1:brokerSalesResult>
```

Figure 16-11 Testing the DADX Web service

3. In the same manner, you can test the `policyQuote` method. Enter `Health` (or `MyPlan` or `Divorce Plan`) into the *planname* text field and click **Invoke**. When you have finished examining the methods of the Insurance Web application, exit the Web browser.

> **Important:** Any changes you make to your Web services can be retested by returning to the Web browser. When running a test environment, the server is running against the resources that are in the workbench. The server will pick up any changes you make to the Web project without being restarted.

Testing the Web service with the Web Services Explorer

The Web Services Explorer can be used to test a Web service based on the generated WSDL file:

1. Expand the ItsoDB2XMLInsuraWeb project (WebContent/wsdl), select the **insuraService.wsdl** file and click **Web Services -> test with Web Services Explorer** (context).
2. An internal server starts; be patient. When the Web Services Explorer opens, expand the service in the Navigator pane (left side).

3. Select the **brokerSales** or **policyQuote** method.
4. Enter a parameter value in the entry field (right side) and click **Go**.
5. Select **Source** in the Status pane (bottom) and study the input and output SOAP messages, for example, the `brokerSales` output:

```
- <SOAP-ENV:Envelope
      xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
- <SOAP-ENV:Body>
- <ns1:brokerSalesResponse
      xmlns:ns1="http://tempuri.org/insuraGroup/insura.dadx"
      SOAP-ENV:encodingStyle="http://xml.apache.org/xml-soap/literalxml">
- <return>
- <xsd1:brokerSalesResult
   xmlns:xsd1="http://tempuri.org/ItsoDB2XMLInsuraWeb/insuraGroup
   /insura.dadx/XSD" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
- <brokerSalesRow>
  <BROKER_ID>456</BROKER_ID>
  <TOTAL_PREMIUM_SALES>11111.10</TOTAL_PREMIUM_SALES>
  </brokerSalesRow>
  </xsd1:brokerSalesResult>
  </return>
  </ns1:brokerSalesResponse>
  </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
```

16.2.10 Publishing the Insura Web service to the Test Registry

For the external world to be able to use the Web service, it should be published to a UDDI registry. You have a number of choices for publishing of a Web service:

- **UDDI Business registry**—Official UDDI registry, a replicated registry run by IBM, Microsoft, and SAP. You can find the IBM Business registry at:

  https://uddi.ibm.com/ubr/registry.html

- **IBM UDDI Test Registry**—a test registry where anybody can make entries for testing, You can find the IBM Test Registry (Microsoft and SAP also provide test registries) at:

  https://uddi.ibm.com/testregistry/registry.html

- **Private registry**—IBM ships a private registry product with WebSphere Application Server V5 Network Deployment. This registry can be installed into a WebSphere server.

- **Unit Test UDDI**—a private registry that can be installed within Application Developer (select **New -> Web Services -> Unit Test UDDI** then select **Cloudscape™** or **DB2** as the database).

To illustrate UDDI publishing, we are using the IBM UDDI Test Registry.

IBM UDDI Test Registry

Before you publish your Web service to the test registry, you must get an IBM ID and password.

Prerequisite: Register with the IBM UDDI Test Business Registry.

Important: To register with the IBM UDDI Test Business Registry, follow the instructions provided by the registry's Web site. In order to activate your account, you must initially log in to the registry from:

https://uddi.ibm.com/testregistry/registry.html

WebSphere Studio provides a Web Services Explorer tool that enables you to publish and maintain your business entity, business services, and service interfaces. For more information on UDDI data structure types, refer to the Web service tools in the online help for WebSphere Studio.

The business entity contains information about the business, for example contact information and URLs.

Important: The IBM UDDI Test Registry allows for only one business entity to be published per user ID. If you previously published a business entity to the IBM UDDI Registry, you can either remove the existing business entity, or publish the new Web service using your existing business entity. For information on removing a business entity, refer to the Web services tools in the online help.

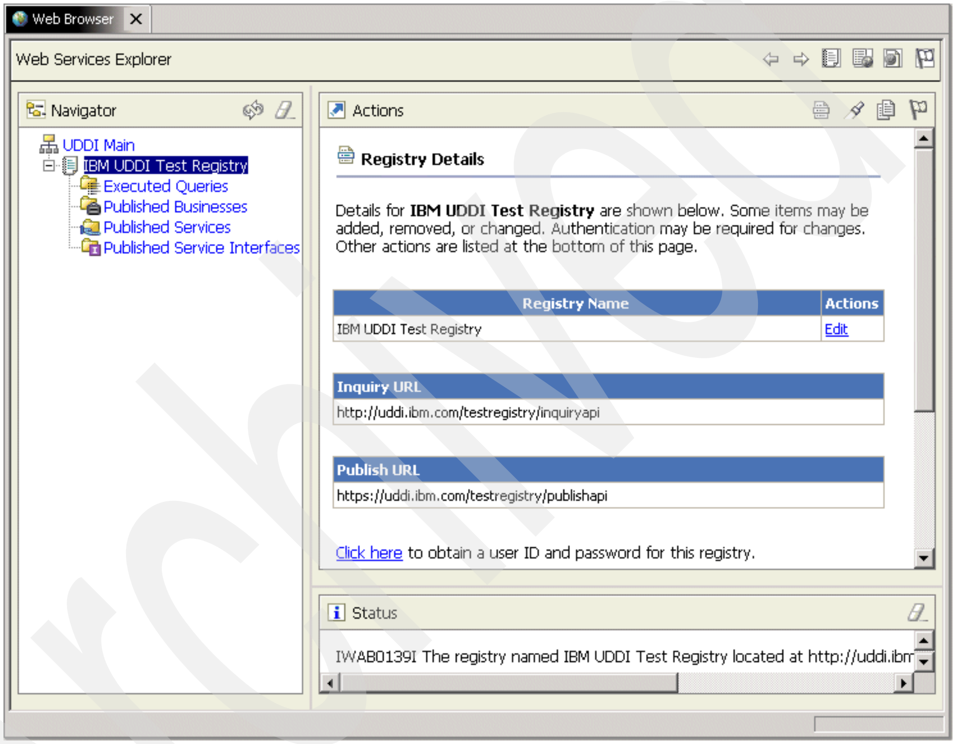
To remove the existing business entity, refer to “Removing a business entity, Web service, or service interface from a registry” in the online help. Once you have unpublished your business entity, follow the steps in ”Publishing the business entity” and ”Publishing the Web service”.

To publish the Insura Web service using your existing business entity, skip the steps in “Publishing the business entity” on page 496 and follow the directions in ”Discovering the business entity” and “Publishing the Web service” on page 497.

Publishing the business entity

To publish your business entity using the Web Services Explorer:

1. Launch the Web Services Explorer by selecting **Run -> Launch the Web Services Explorer** in the main toolbar.
2. In the Navigator pane, select the **UDDI Main** node. In the Actions pane, `IBM UDDI Test Registry` will appear in the Registry Name field. Click **Go** (Figure 16-12).



Details for **IBM UDDI Test Registry** are shown below. Some items may be added, removed, or changed. Authentication may be required for changes. Other actions are listed at the bottom of this page.

Registry Name	Actions
IBM UDDI Test Registry	Edit

Inquiry URL
http://uddi.ibm.com/testregistry/inquiryapi

Publish URL
https://uddi.ibm.com/testregistry/publishapi

Click here to obtain a user ID and password for this registry.

IWAB0139I The registry named IBM UDDI Test Registry located at http://uddi.ibm

Figure 16-12 Publishing a business entity to the BM UDDI Test Registry

3. In the toolbar of the Actions pane, click the **Publish** icon .
4. From the Publish list, select **Business**.
5. Select **Simple**.
6. In the Publish URL field, keep the default URL. Enter your user ID, password, a suitable business entity name (in further discussion to be referred to as *YourBusinessEntityName*), and a description of the business entity in the respective fields. In this particular instance, we used `ItsoDB2XMLRedBook` as our business entity name and `test web service` as a description.

Be sure to select your own business name and not `YourBusinessEntityName`.

7. Click **Go**. The Web Services Explorer is automatically updated with your published business entity.

Discovering the business entity

You can find your business entity using the Web Services Explorer in this way:

1. In the Navigator pane, select the **IBM UDDI Test Registry** node.
2. In the toolbar of the Actions pane, click the **Find** icon.
3. From the *Search for* list, select **Businesses**.
4. Give your query a descriptive name, in our case I`tsoDB2XMLRedBook Business Entity`.
5. Select **Simple**.
6. In the *Name* field of the Actions pane, enter the `YourBusinessEntityName` name and click **Go**.

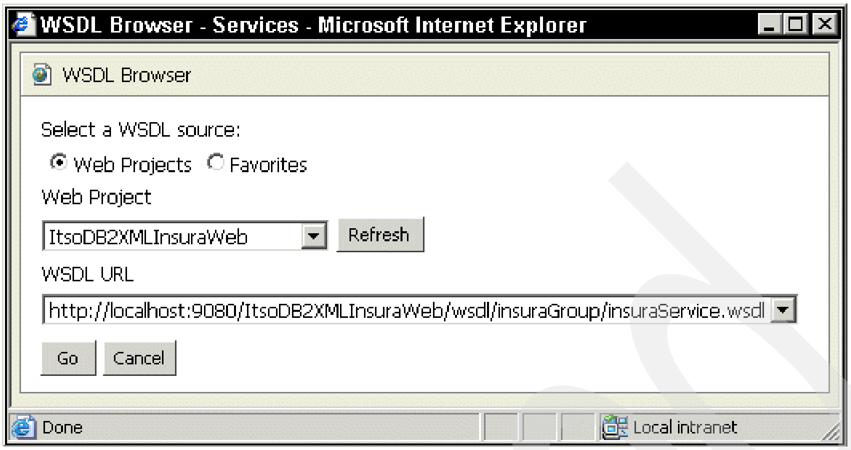
Publishing the Web service

To publish a Web service under your business entity follow these steps:

1. In the Navigator pane, select **YourBusinessEntityName** name under the *Published Businesses* node.
2. In the toolbar of the Actions pane, click the **Publish Service** icon.
3. Select **Simple**.
4. To enter the WSDL URL, click **Browse** to select the WSDL URL for your Web project. If your Web service is currently running, your pop-up WSDL Browser window should look like Figure 16-13 on page 498 with a WSDL URL automatically selected:

   `http://localhost:9080/ItsoDB2XMLInsuraWeb/wsdl/insuraGroup/insuraService.wsdl`

   If it is not the case, just select the WSDL file from the drop-down list. Click **Go**.



WSDL Browser - Services - Microsoft Internet Explorer

WSDL Browser

Select a WSDL source:

● Web Projects ○ Favorites

Web Project

ItsoDB2XMLInsuraWeb

WSDL URL

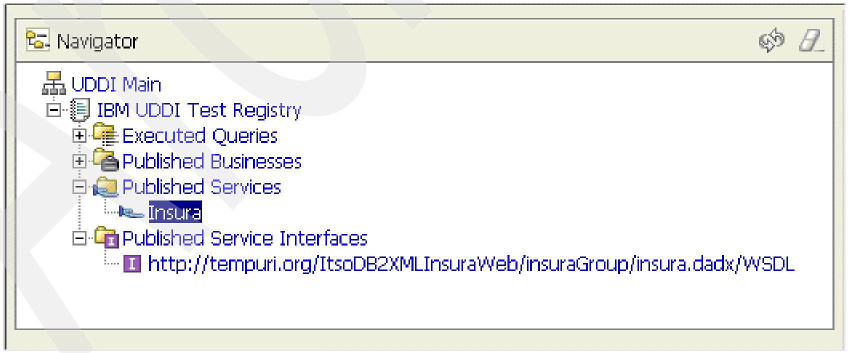
http://localhost:9080/ItsoDB2XMLInsuraWeb/wsdl/insuraGroup/insuraService.wsdl

Figure 16-13 Publishing Insura Web service: defining WSDL URL

> **Note**: Because the address points to localhost:9080 you can only interact with the registry if your server is running.

5. In the Name field of the Actions pane, type `Insura`, and in the Description field type `Insura Test Web Service`.
6. When you have finished entering the service information, click **Go**.

The Web Services Explorer is automatically updated with your published Web service. If your update is successful, the Status pane informs you that the Web service interface and Web service were successfully published. The Navigator pane of the Web Services Explorer is shown in Figure 16-14



Navigator

- UDDI Main
  - IBM UDDI Test Registry
    - Executed Queries
    - Published Businesses
    - Published Services
      - Insura
    - Published Service Interfaces
      - http://tempuri.org/ItsoDB2XMLInsuraWeb/insuraGroup/insura.dadx/WSDL

Figure 16-14 Web service published to the IBM UDDI Test Registry

When you have finished publishing your Web service, you may exit the Web Services Explorer.

16.2.11 Summary

In this section, we successfully imported the INSURA database resources into our workspace and created two SQL statements, `policyQuote` and `brokerSales`. We created a DADX Group configuration file and a DADX file that contains both queries. Using Application Developer tools, we generated the Web service and a test client application that helped us to examine and visualize the Web services in real time through a set of generated JSP files. The output seen was an XML file. Then we published the Web service to the IBM UDDI Test Registry.

16.3 Discovering the Insura Web service

This part of our scenario shows how another developer in our partner broker company could use WebSphere Studio to discover the Web service that you just registered (business case #4). You will begin by creating a Web project into which you will import the discovered WSDL document. You will now play the role of a broker discovering a Insura Web service.

16.3.1 Creating the broker Web project

We will discover the Web service and then implement a Web client application that uses the Web service. Therefore, we create another Web project to simulate the broker client. We could use the generated ItsoDB2XMLInsuraWebClient project, but it is cleaner to show the process in a new project, simulating the fact that this would be done on another machine.

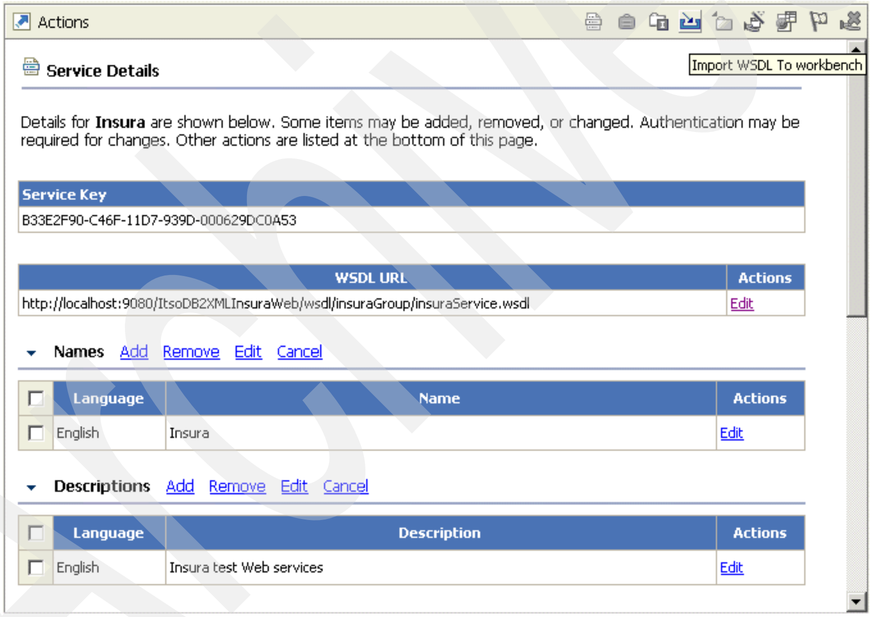
We are going to make our instructions brief because the process is the same as for the ItsoDB2XMLInsuraWeb project. Create an ItsoDB2XMLBrokerWebClient project attached to an ItsoDB2XMLBroker enterprise application. Follow the instructions in “Creating the Web project” on page 480.

16.3.2 Discovering the Web service

You can discover a Web service by searching for a business entity, business service, or service interface. In this sample, you will query the registry with the name of business entity that you created when you were playing the role of a developer publishing a Web service. To discover the Insura Web service using the Web Services Explorer:

1. Launch the Web Services Explorer by clicking **Run -> Launch the Web Services Explorer** in the main toolbar.
2. In the Web Services Explorer toolbar, select the **Favorites** icon .

3. Expand Favorite UDDI Registries and select the **IBM UDDI Test Registry** node.
4. In the Actions toolbar, click the **Add to UDDI Page** icon (or link under the same name at the bottom of the pane).
5. In the Actions toolbar, click the **Find** icon (or link with the same name at the bottom).
6. In the Actions pane, type `Insura` as the query name.
7. Select **Simple**. In the Name text field, type the name of the business that published the Insura Web service: `YourBusinessEntityName`. Click **Go**. If your query was successful, you receive all related Business Details for your YourBusinessEntityName in the Actions pane.
8. In the Actions toolbar, click the **Get Services** icon. The Actions pane is shown in Figure 16-15.



Service Details

Details for **Insura** are shown below. Some items may be added, removed, or changed. Authentication may be required for changes. Other actions are listed at the bottom of this page.

Service Key
B33E2F90-C46F-11D7-939D-000629DC0A53

WSDL URL	Actions
http://localhost:9080/ItsoDB2XMLInsuraWeb/wsdl/insuraGroup/insuraService.wsdl	Edit

Names Add Remove Edit Cancel

□	Language	Name	Actions
□	English	Insura	Edit

Descriptions Add Remove Edit Cancel

□	Language	Description	Actions
□	English	Insura test Web services	Edit

Figure 16-15 Discovering Insura Web service

Important: Remember, each UDDI registry service entry only references the actual Web server where this Web service runs. In our case, WSDL points to our *localhost* machine where we deployed the Insura Web service on the WebSphere V5.0 Test Environment Application server. That is why our Web service, though registered on IBM UDDI Test Business Registry, won't be reachable from the external world, unless you replace localhost with your machine's actual IP.

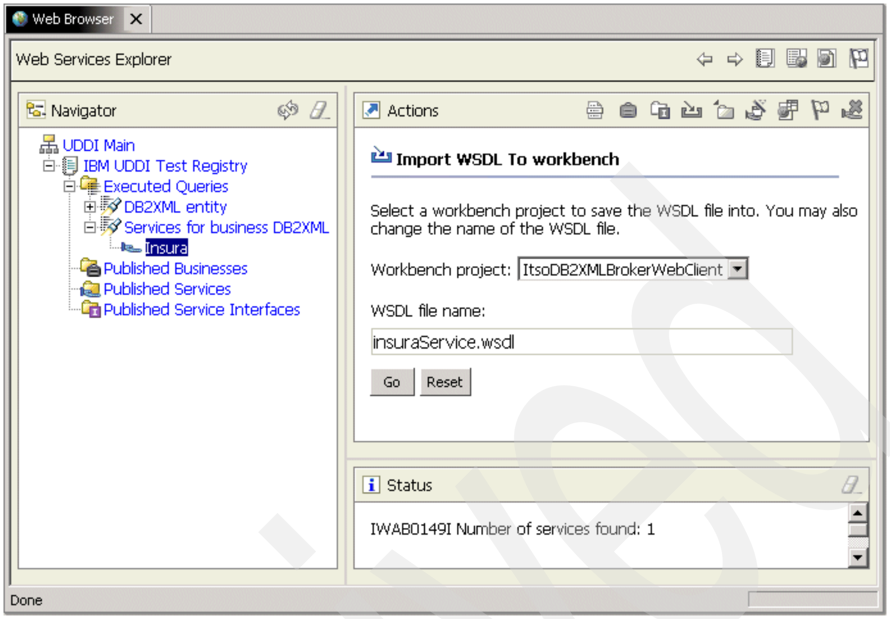
To be able to discover your Web services, you have to keep the WebSphere V5.0 Test Environment server running.

Once you have discovered a Web service, you can import the WSDL document to your Web project and launch the Web Service Client wizard to test the Web service.

Importing the WSDL document

To import the WSDL document into the ItsoDB2XMLBrokerWebClient project:

1. In the Actions toolbar of Web Services Explorer, click the **Import WSDL To workbench** icon .
2. In the Import WSDL to workbench page of the Actions pane, select **ItsoDB2XMLBrokerWebClient** from the Workbench project: drop-down list. Leave the WSDL file name field with insuraService.wsdl by default (Figure 16-16 on page 502). Click **Go**. The insuraService.wsdl file is imported in the ItsoDB2XMLBrokerWebClient project folder.



Import WSDL To workbench

Select a workbench project to save the WSDL file into. You may also change the name of the WSDL file.

Workbench project: ItsoDB2XMLBrokerWebClient

WSDL file name:

insuraService.wsdl

IWAB0149I Number of services found: 1

Figure 16-16 Importing WSDL file into Web project

Creating the Web service client

To implement a client application, we require a proxy and (optionally) a test client. These are generated by the Web Service Client wizard. To generate a proxy and a test client for the Insura Web service, perform these steps:

1. Expand the ItsoDB2XMLBrokerWebClient folder and select the **insuraService.wsdl** file (Figure 16-17 on page 503). Select **File -> New -> Other -> Web Services-> Web Service Client**. Click **Next**.

   Note: In Application Developer V5.1 you can select **Web Services -> Generate Client** as a short cut.

   You are forwarded to the first page of the Web Service Client wizard.

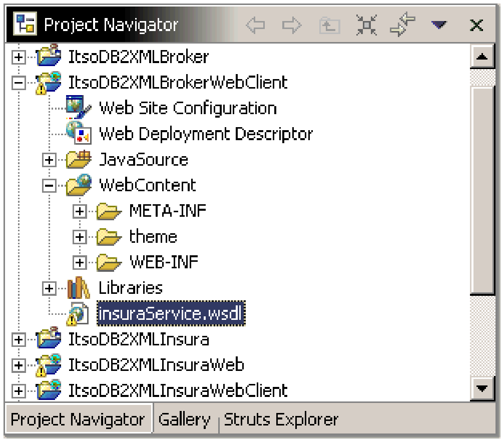
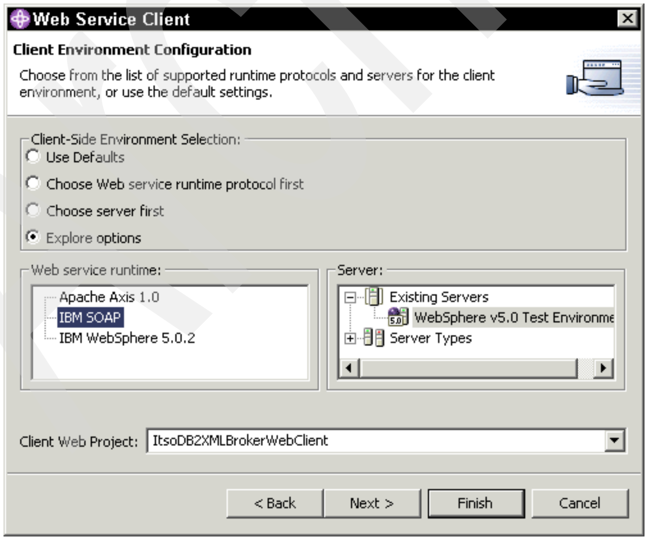


Figure 16-17 Creating Broker Web service client

2. Make sure that the *Client proxy type* is Java Proxy, and that **Test the generated proxy, Overwrite file without warning** and **Create folders when necessary** are selected. Click **Next**.
3. In the *Client Environment Configuration* page of the wizard select **Explore options**. In the *Web service runtime:* pane, select **IBM SOAP** and ensure that in the Server pane **WebSphere v5.0 Test Environment** is selected and that the Client Web Project is `ItsoDB2XMLBrokerWebClient` (Figure 16-18). Click **Next**.



Web Service Client

Client Environment Configuration

Choose from the list of supported runtime protocols and servers for the client environment, or use the default settings.

Client-Side Environment Selection:

○ Use Defaults

○ Choose Web service runtime protocol first

○ Choose server first

● Explore options

Web service runtime:

- Apache Axis 1.0
- IBM SOAP
- IBM WebSphere 5.0.2

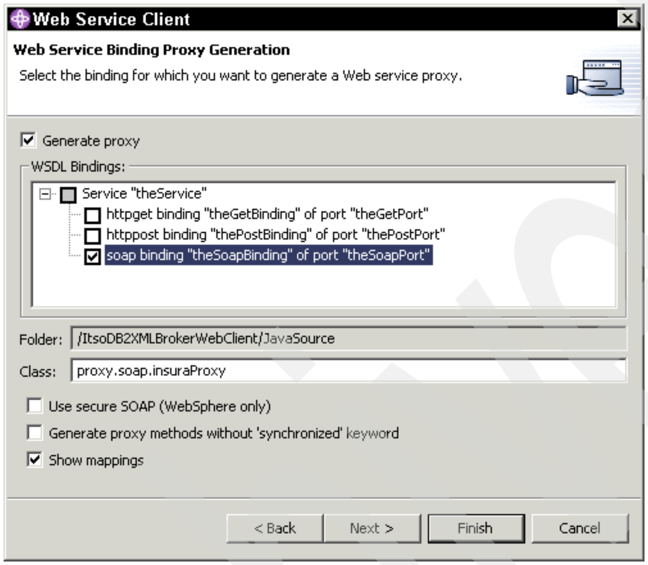
Server:

- Existing Servers
  - WebSphere v5.0 Test Environme
- Server Types

Client Web Project: ItsoDB2XMLBrokerWebClient

Figure 16-18 Selecting the client runtime environment

4. The Web Service Selection page provides you with options to enter a URI to a WSDL or other defining Web service document. Click **Next** to accept the prefilled name of your WSDL file.
5. In the Web Service Binding Proxy Generation page, select **soap binding** and **Show mappings** (Figure 16-19). Click **Next**.



Web Service Client

Web Service Binding Proxy Generation

Select the binding for which you want to generate a Web service proxy.

☑ Generate proxy

WSDL Bindings:

- □ Service "theService"
  - □ httpget binding "theGetBinding" of port "theGetPort"
  - □ httppost binding "thePostBinding" of port "thePostPort"
  - ☑ soap binding "theSoapBinding" of port "theSoapPort"

Folder: /ItsoDB2XMLBrokerWebClient/JavaSource

Class: proxy.soap.insuraProxy

□ Use secure SOAP (WebSphere only)

□ Generate proxy methods without 'synchronized' keyword

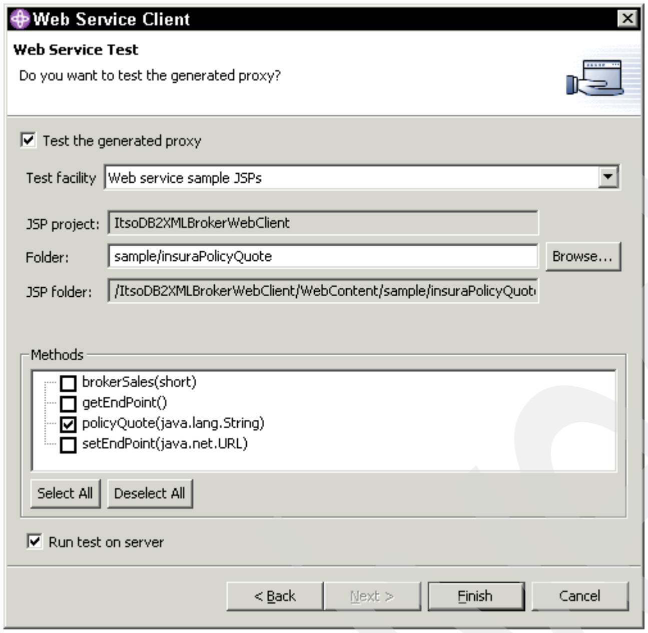
☑ Show mappings

Figure 16-19 Proxy generation

6. In the Web Service XML to Java Mappings page of the wizard, leave the settings for the parameters untouched, but change the mapping of the two results to `Show and use the default DOM Element mapping` (see Figure 16-10 on page 492). Click **Next**.
7. In the Web Service SOAP Binding Mapping Configuration page, review the SOAP binding mapping configurations. Click **Next** to accept the default values.
8. In the Web Service Test page, ensure that the **Test the generated proxy** and **Run test on server** check boxes are selected and that `Web service sample JSPs` appears in the *Test facility* text field.

   For simplicity, we limit our considerations to only one method of the Insura Web service that returns a policy quote (Figure 16-20 on page 505):

   a. Enter `sample/insuraPolicyQuote` as the target folder
   b. Deselect all other methods in the Methods pane.
   c. Click **Finish**.



Web Service Test

Do you want to test the generated proxy?

☑ Test the generated proxy

Test facility: Web service sample JSPs

JSP project: ItsoDB2XMLBrokerWebClient

Folder: sample/insuraPolicyQuote

JSP folder: /ItsoDB2XMLBrokerWebClient/WebContent/sample/insuraPolicyQuot

Methods

- □ brokerSales(short)
- □ getEndPoint()
- ☑ policyQuote(java.lang.String)
- □ setEndPoint(java.net.URL)

☑ Run test on server

Figure 16-20 Methods of the test client

The ItsoDB2XMLBroker EAR application is deployed to the WebSphere Application Server Test Environment, the server is started and an internal Web browser is launched to test the Web service.

As you might have already noticed, we very closely repeated steps that were performed when we created DADX Web service.

Test the Insura Web service by follow the instructions in 16.2.9, “Testing the DADX Web services” on page 492. Your resulting Web browser view is basically the same as Figure 16-11 on page 493, with the exception that it contains only one method (`policyQuote`).

Now that you have created the Broker Web client application for the Insura Web service, deployed it on the server, and tested it by invoking methods of the actual Insura Web service, you, as a broker’s software developer, want to make a nicer Web front end.

16.3.3 Creating a real client using XSL transformation

In this section of our scenario, you, as a broker’s software developer, are going to take the XML file that is being produced by the Web service and use XSL to transform the output to an HTML file containing the result in a table.

We take the generated Result.jsp of the test client and modify it to apply an XSL stylesheet. We invoke the JSP from a simple HTML page.

Preparation

Create a folder named *client* in the ItsoDB2XMLBrokerWebClient project (under WebContent).

Import these three files from c:\SG246994\Scenario\wsad into the client folder:

- **policyQuote.xml**—this is a sample result from running the Web service. You could create this file yourself by running the Web service in the test client and copy/paste the Results pane into a new XML file.
- **ClientInput.html**—a simple HTML file with a form to enter the plan name and to invoke the JSP.
- **ClientResult.jsp**—A simplified copy of the Results.jsp that runs the Web service and applies an XSL style sheet named policyQuote/xsl.

  The ClientResult.jsp has these modifications from the original Result.jsp:

  – The heading is replaced by:

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  <?xml-stylesheet type="text/xsl" href="policyQuote.xsl"?>
  ```

  – The trailer is deleted (`</BODY></HTML>`).

  – The `markup` method does not perform any substitution:

  ```
  public static String markup(String text) {
      return text;
  }
  ```

  – Two instances of `<br>` are deleted. Note that for the second one, the preceding `if` is deleted as well.

  – The logic of the method selection is deleted; we only run one method.

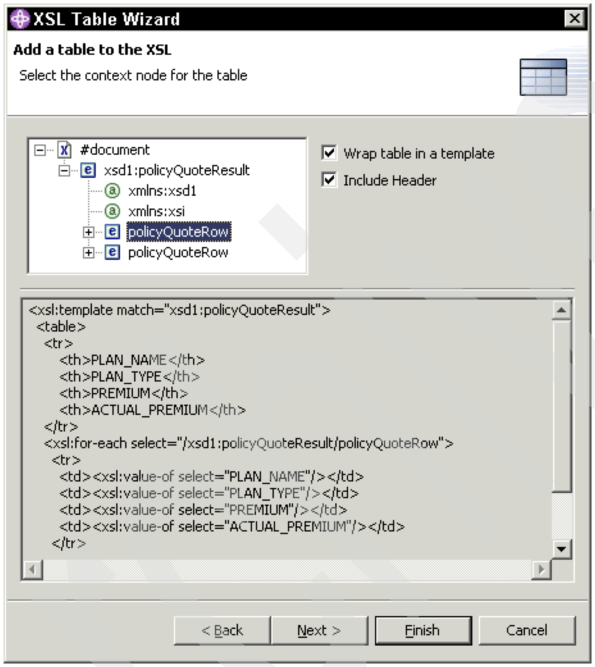
Creating the XSL stylesheet

Although you could import the policyQuote.xsl file as well, it can be generated easily from the sample XML result file:

1. Create an XSL file using **File -> New -> Other -> XML -> XSL**.
2. Set the output folder to the `client` folder and the name to `policyQuote.xsl`.
3. To select an XML file, expand the ItsoDB2XMLBrokerWebClient project and select the **policyQuote.xml** file.
4. Click **Finish**.

The XSL editor opens and we can now map our sample XML file into an HTML formatted table:

1. Add a few lines in front of the trailing `</xsl:stylesheet>` tag. Make sure that your cursor stays at one of these blank lines.
2. Launch the *Create a table in XSL* wizard by clicking the icon or select **XSL -> HTML Table**.
3. Expand and select the first **policyQuoteRow** element. Select both **Wrap table in a template** and **Include header** (Figure 16-21).



Add a table to the XSL

Select the context node for the table

☑ Wrap table in a template

☑ Include Header

```
<xsl:template match="xsd1:policyQuoteResult">
 <table>
  <tr>
    <th>PLAN_NAME</th>
    <th>PLAN_TYPE</th>
    <th>PREMIUM</th>
    <th>ACTUAL_PREMIUM</th>
  </tr>
  <xsl:for-each select="/xsd1:policyQuoteResult/policyQuoteRow">
   <tr>
    <td><xsl:value-of select="PLAN_NAME"/></td>
    <td><xsl:value-of select="PLAN_TYPE"/></td>
    <td><xsl:value-of select="PREMIUM"/></td>
    <td><xsl:value-of select="ACTUAL_PREMIUM"/></td>
   </tr>
```

Figure 16-21 Defining XSL transformation

4. Click **Next**. Specify `2` in the Border field and click **Finish**.
5. Add a heading before the table:

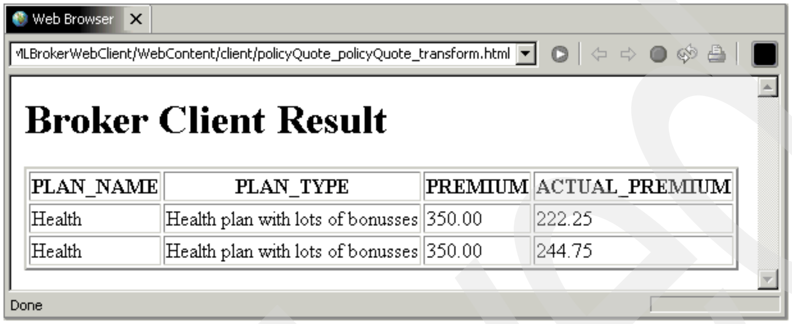
```
<xsl:template match="xsd1:policyQuoteResult">
  <h1>Broker Client Result</h1>
  <table border="2">
  ......
```

6. Save and close the XSL file. Compare your file with the policyQuote.xsl file provided in c:\SG246994\Scenario\wsad.

Testing the XSL transformation

We can test the transformation on our sample XML file:

Select both the **policyQuote.xml** and **policyQuote.xsl** files and click **Transform -> XML** (context). A Web browser opens with the resulting HTML file (Figure 16-22).


Web Browser
MLBrokerWebClient/WebContent/client/policyQuote_policyQuote_transform.html
Done

Broker Client Result

PLAN_NAME	PLAN_TYPE	PREMIUM	ACTUAL_PREMIUM
Health	Health plan with lots of bonusses	350.00	222.25
Health	Health plan with lots of bonusses	350.00	244.75

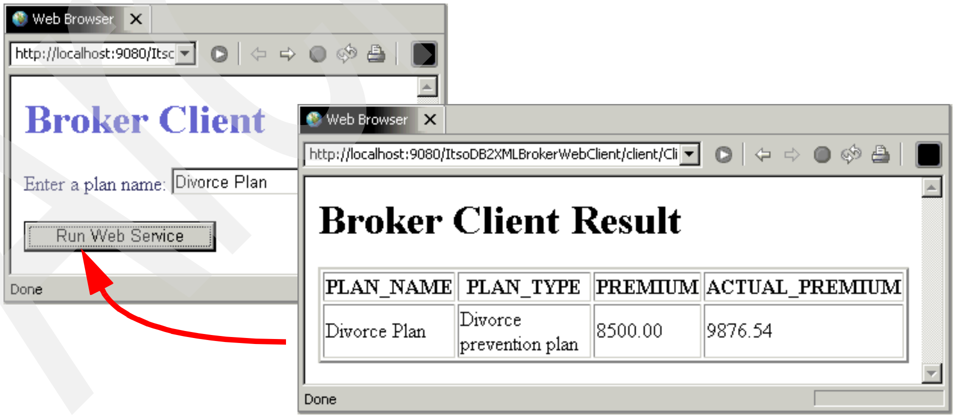
Figure 16-22 Testing the Web service with XSL transformation

Running the Web service

Finally, we can run the real client application:

1. Select the **ClientInput.html** file and **Run on Server**.
2. Enter a plan name (`Health`) and click **Run Web Service**.

The result is shown in Figure 16-23.


Web Browser
http://localhost:9080/Itsc
Done
Web Browser
http://localhost:9080/ItsoDB2XMLBrokerWebClient/client/Cli
Done

Broker Client

Enter a plan name: Divorce Plan

Run Web Service

Broker Client Result

PLAN_NAME	PLAN_TYPE	PREMIUM	ACTUAL_PREMIUM
Divorce Plan	Divorce prevention plan	8500.00	9876.54

Figure 16-23 Testing the real broker client application

16.3.4 Using a data source for DB2 Web services

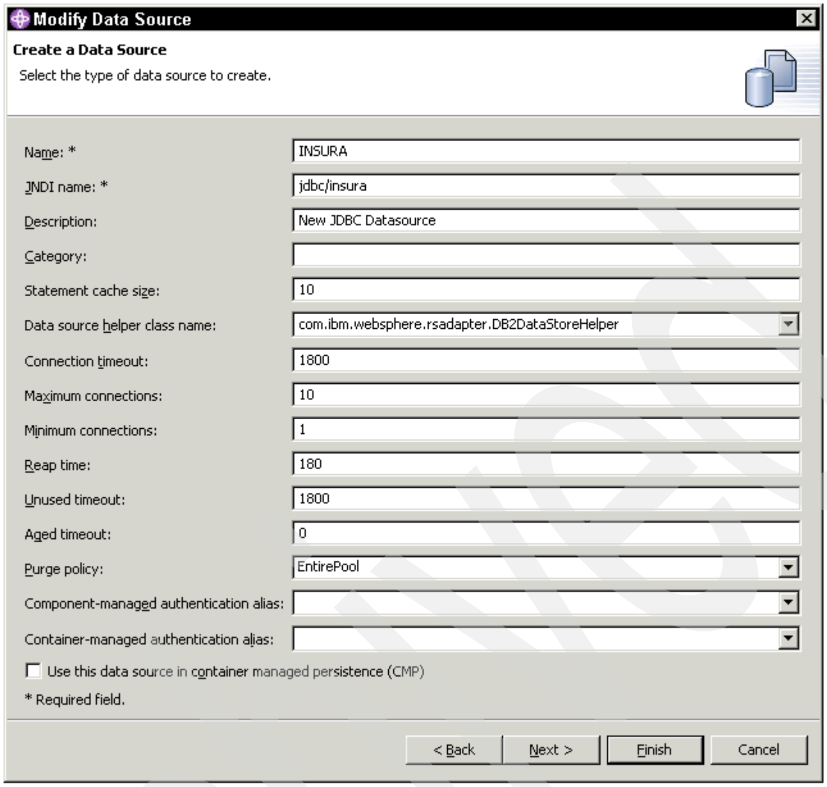
The group configuration (see "Creating the DADX group" on page 481 and Figure 12-45 on page 377) allows the specification of a JDBC driver and/or a data source. If a data source is specified, it is tried first at execution time.

For a data source to be used, it must be configured in the server. In the Server Configuration view, open the configuration of the WebSphere V5.0 Test Environment server (double-click):

1. Go to the Data source page and select the **Default DB2 JDBC Driver**.
2. Click **Add** next to the data source list.
3. Select the **DB2 Legacy** driver, **V5.0 data source**, and click **Next**.
4. Complete the data source information (Figure 16-24 on page 510). The JNDI name must match the specification in the group properties. *Authentication aliases* are a way to store user IDs and passwords used by WORF to connect in WebSphere as opposed to group.properties. Please refer to the following article on the Web for more details:

   `http://www-106.ibm.com/developerworks/db2/library/techarticle/dm-0404wollscheid/index.html`

5. Click **Next**.



Create a Data Source

Select the type of data source to create.

Name: *	INSURA
JNDI name: *	jdbc/insura
Description:	New JDBC Datasource
Category:	
Statement cache size:	10
Data source helper class name:	com.ibm.websphere.rsadapter.DB2DataStoreHelper
Connection timeout:	1800
Maximum connections:	10
Minimum connections:	1
Reap time:	180
Unused timeout:	1800
Aged timeout:	0
Purge policy:	EntirePool
Component-managed authentication alias:	
Container-managed authentication alias:	

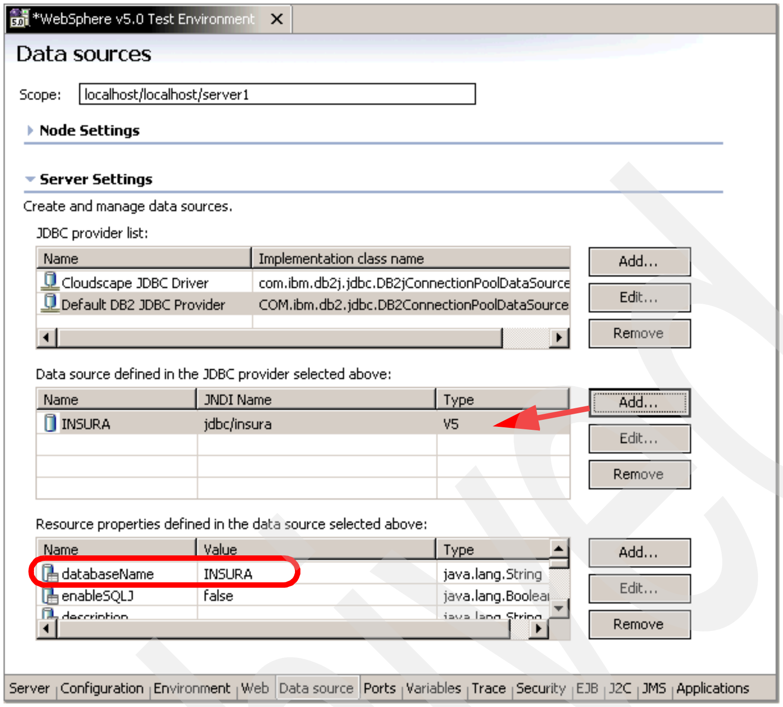
☐ Use this data source in container managed persistence (CMP)

* Required field.

Figure 16-24 Creating a data source in the server configuration

6. For the databaseName property, enter a value of INSURA, and click **Finish**.

   The data source is added to the list (Figure 16-25 on page 511).



*WebSphere v5.0 Test Environment

Data sources

Scope: localhost/localhost/server1

Node Settings

Server Settings

Create and manage data sources.

JDBC provider list:

Name	Implementation class name
Cloudscape JDBC Driver	com.ibm.db2j.jdbc.DB2jConnectionPoolDataSource
Default DB2 JDBC Provider	COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource

Data source defined in the JDBC provider selected above:

Name	JNDI Name	Type
INSURA	jdbc/insura	V5

Resource properties defined in the data source selected above:

Name	Value	Type
databaseName	INSURA	java.lang.String
enableSQLJ	false	java.lang.Boolea
description		java.lang.String

Add...
Edit...
Remove
Server | Configuration | Environment | Web | Data source | Ports | Variables | Trace | Security | EJB | J2C | JMS | Applications

Figure 16-25 Server configuration data source page

7. Save and close the server configuration.
8. Rerun the client application; you will be able to see messages regarding the data source in the Console view.

16.3.5 Summary

In this section, we used WebSphere Studio to discover the Insura Web service that was registered in the preceding section.

Acting as broker's developer, we discovered our partner's business entity in the IBM UDDI Test Registry. After that, we identified the Insura Web service and imported the corresponding WSDL document into our Web application. From the WSDL file, we created the client proxy and a test client. Using the generated test client code, we created a real client that transforms the resulting SOAP XML output into HTML using an XSL stylesheet. We created the XSL file from a sample XML output and formatted the output rows into an HTML table. We tested the transformation and finally run our broker Web service client by invoking the DADX Web service.

16.4 Insurance Web application with DB2 XML access

In this section, we implement business case #2, which is to create and publish a Web page where brokers can get a policy quote for a given customer. The algorithm to calculate the quote for a given customer takes into account the total amount of money claimed so far, using the formula:

```
regular premium + 4% of claims
```

The claims data is stored in the c:\SG246994\Scenario\InsuranceHistory.xml file, which, generally speaking, can be reached either in the local file system or over the Internet.

There are a number of ways in which you can access databases from a Web application. You can write your own Java classes and access the database using standard JDBC calls. Alternatively, Application Developer supplies a library of database access beans, called DB Beans. These can be used in a JSP through the `<useBean>` tag and can also be accessed through a set of JSP tags supplied in a tag library. An application using DB Beans or tags can be generated for you by Application Developer using a wizard based on an SQL statement.

We will use this Database Web Pages wizard to create an application based on JSPs using the DB Beans tag library by starting from an SQL statement.

16.4.1 Creating the proposedPremium SQL query

We implement the Web application in the `ItsoDB2XMLInsuraWeb` project.

To construct the `proposedPremium` statement (Example 16-1 on page 513), you can use the SQL Query Builder and follow the steps that were described in “Defining the policyQuote SQL statement” on page 485“. You have to type the complicated select expression.

Alternatively, you can create a shortcut by copying and pasting the SQL statement into the SQL Source pane of the SQL Query Builder:

```
c:\SG246994\Scenario\wsad\proposedPremium.sql
```

Save the query and test it by entering these values for the host variables:

```
:history_file_uri:  'file:C:\SG246994\Scenario\InsuranceHistory.xml'
:customer_name:     'Shrinivas Kulkarni'
:plan_name:         'Motor'
```

Note. You can copy and paste these values from the file:

```
c:\SG246994\Scenario\wsad\proposedPremiumInputValues.txt
```

If you have done everything correctly, you should get 556.28 as the resulting proposed premium.

Example 16-1 SQL statement for proposed premium

```
SELECT DISTINCT
   (ITSO.P_TYPE.PREMIUM +
   (
      SELECT
         COALESCE((SUM(CAST (ITSO.CLAIMS.AMOUNT AS DECIMAL)) * 0.04), 0)
      FROM
         ITSO.CLAIMS
      WHERE
         ITSO.CUSTOMERS.CUSTOMER_ID = ITSO.INSURANCES.CUSTOMER_FID
         AND ITSO.INSURANCES.INSURANCE_ID = ITSO.CLAIMS.INSURANCE_FID
   )) AS proposed_premium
FROM
   ITSO.CUSTOMERS, ITSO.INSURANCES, ITSO.CLAIMS, ITSO.P_TYPE
WHERE
   ITSO.CUSTOMERS.CUSTOMER_ID = ITSO.INSURANCES.CUSTOMER_FID
   AND ITSO.INSURANCES.INSURANCE_ID = ITSO.CLAIMS.INSURANCE_FID
   AND ITSO.CUSTOMERS.URI = :history_file_uri
   AND ITSO.CUSTOMERS.NAME = :customer_name
   AND ITSO.P_TYPE.PLAN_NAME = :plan_name
```

16.4.2 Generating Web pages from SQL queries

Application Developer provides a wizard to help you create a set of Web pages and supporting Java classes starting from an existing or new SQL query. The wizard generates the required HTML pages, JSPs, and Java classes to quickly create a working skeleton application without you having to write any code. You can then expand and modify the generated code to create the finished application.

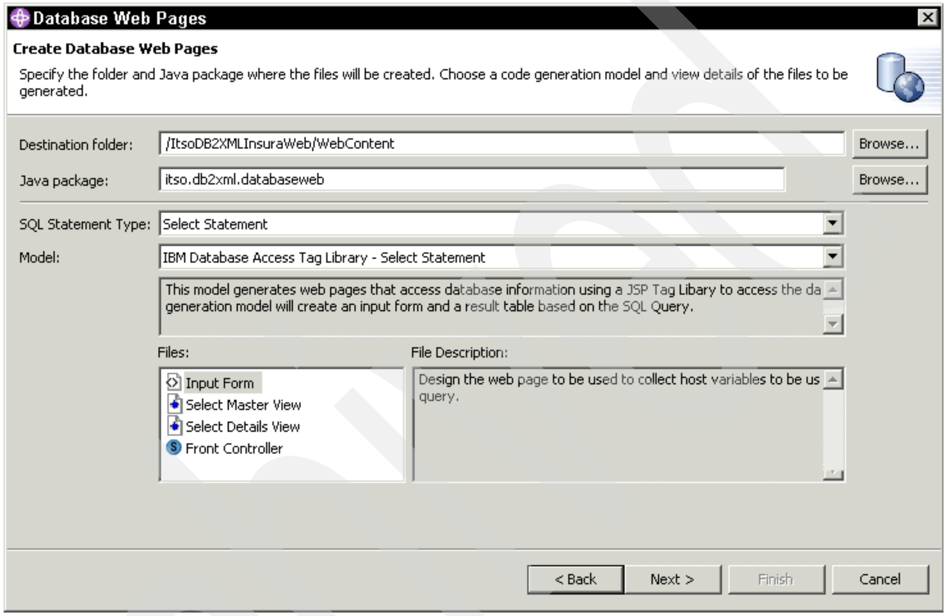
To start generating Web pages from the proposedPremium query, proceed with these steps:

1. Select the target folder **ItsoDB2XMLInsuraWeb/WebContent**.
2. Start the Database Web Pages wizard by selecting **File -> New -> Other -> Web -> Database Web Pages** and click **Next**.
3. The first page of the wizard is displayed. Complete the dialog as shown in Figure 16-26 on page 514:
   - Destination folder: folder where the generated Web pages are stored. Make sure that the folder is /ItsoDB2XMLInsuraWeb/WebContent.

– Java package: package where Java classes are generated. Enter `itso.db2xml.databaseweb` as Java package name.

– SQL Statement Type: the type of action to perform on the database. Ensure that **Select Statement** is selected.

– Model: select **IBM Database Access Tag Library- Select Statement**.



Database Web Pages

Create Database Web Pages

Specify the folder and Java package where the files will be created. Choose a code generation model and view details of the files to be generated.

Destination folder: /ItsoDB2XMLInsuraWeb/WebContent

Java package: itso.db2xml.databaseweb

SQL Statement Type: Select Statement

Model: IBM Database Access Tag Library - Select Statement

This model generates web pages that access database information using a JSP Tag Libary to access the da generation model will create an input form and a result table based on the SQL Query.

Files:

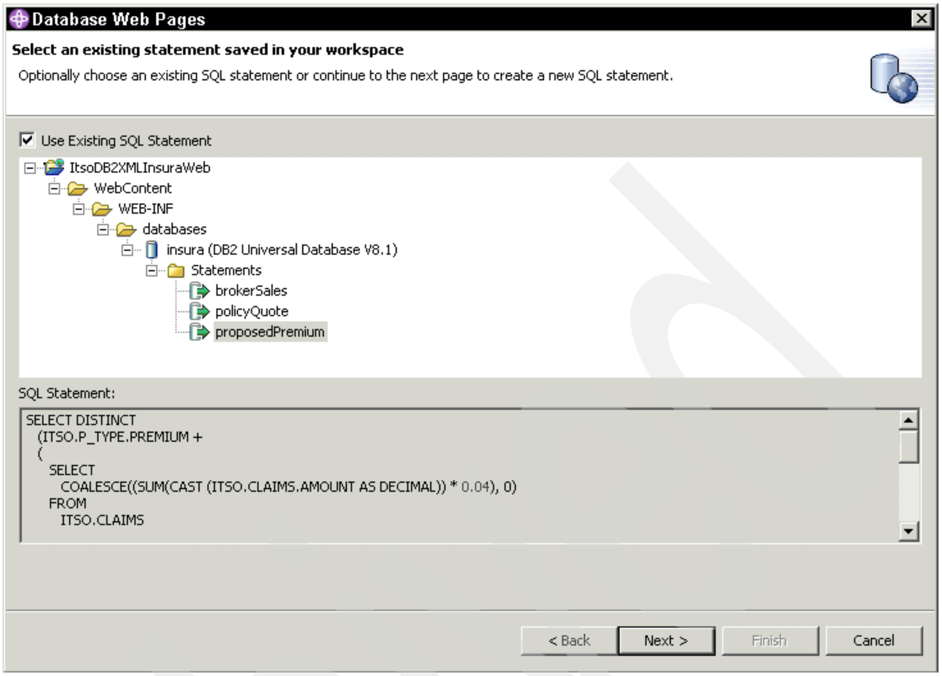
- Input Form
- Select Master View
- Select Details View
- Front Controller

File Description:

Design the web page to be used to collect host variables to be us query.

Figure 16-26 DB Web Pages wizard: select target, statement type, and model

4. Clicking **Next** brings up the second wizard page where you can select an existing SQL statement. Make sure to select **Use Existing SQL statement**. Keep expanding the ItsoDB2XMLInsuraWeb project until you find and select the `proposedPremium` query in the Statements folder (Figure 16-27 on page 515). Click **Next**.



Select an existing statement saved in your workspace

Optionally choose an existing SQL statement or continue to the next page to create a new SQL statement.

☑ Use Existing SQL Statement

- ItsoDB2XMLInsuraWeb
  - WebContent
    - WEB-INF
      - databases
        - insura (DB2 Universal Database V8.1)
          - Statements
            - brokerSales
            - policyQuote
            - proposedPremium

SQL Statement:

```
SELECT DISTINCT
  (ITSO.P_TYPE.PREMIUM +
  (
    SELECT
      COALESCE((SUM(CAST (ITSO.CLAIMS.AMOUNT AS DECIMAL)) * 0.04), 0)
    FROM
      ITSO.CLAIMS
```

Figure 16-27 Database Web Pages wizard: select SQL statement

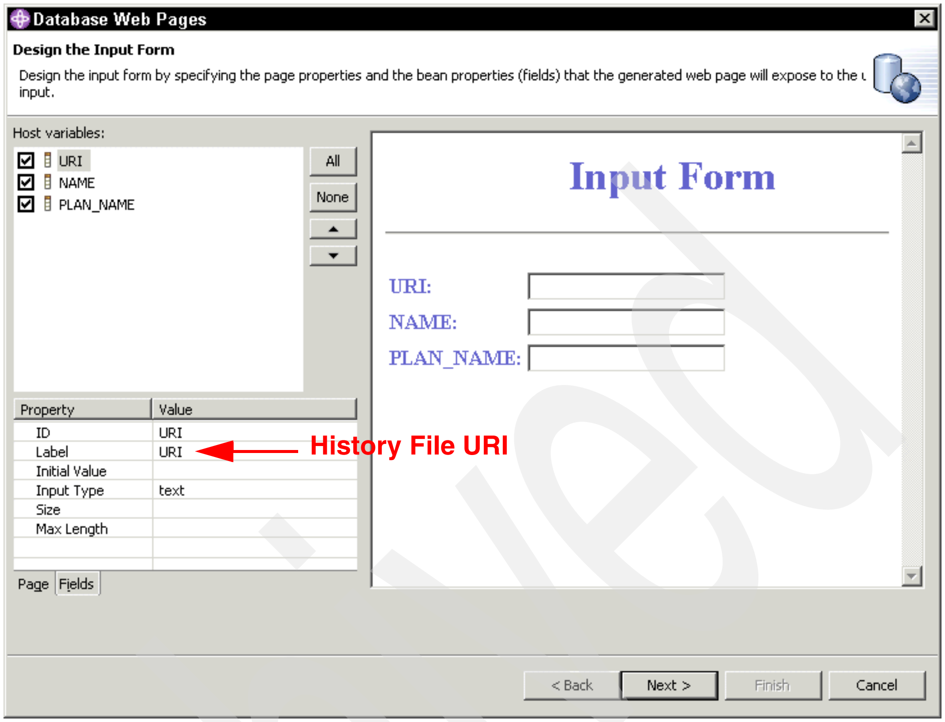
5. The Runtime Connection Page of the wizard allows you to specify the database connection you would like to use at runtime. For our scenario, we are using a simple JDBC connection. Ensure that **Use driver manager connection** is selected; in the fields Driver name and URL, you should have `COM.ibm.db2.jdbc.app.DB2Driver` and `jdbc.db2.insura` respectively.

   Alternatively, if you created the data source in the server configuration (see 16.3.4, “Using a data source for DB2 Web services” on page 509) then use the same data source name `jdbc/insura` here.

6. Click **Next**.

7. In the Controller Page, click **Next**, accepting the default values (store results in *Request, Do not use a front controller*).

8. On the Design the Input From page, the HTML input form is displayed (Figure 16-28 on page 516). Here you can make changes to page and field properties. Notice that the three input fields are automatically generated. This is where the `:history_file_uri`, `:customer_name` and `:plan_name` host variable values will come from.



Database Web Pages

Design the Input Form

Design the input form by specifying the page properties and the bean properties (fields) that the generated web page will expose to the u input.

Host variables:

- ☑ URI
- ☑ NAME
- ☑ PLAN_NAME

Property	Value
ID	URI
Label	URI
Initial Value	
Input Type	text
Size	
Max Length	

History File URI

Figure 16-28   Database Web Pages wizard: input form

Select each host variable and change the label to the desired text, for example, `History File URI`, `Customer Full Name`, and `Plan Name`. Once this is done and the following pages have been generated, you can make further changes using the Page Designer.

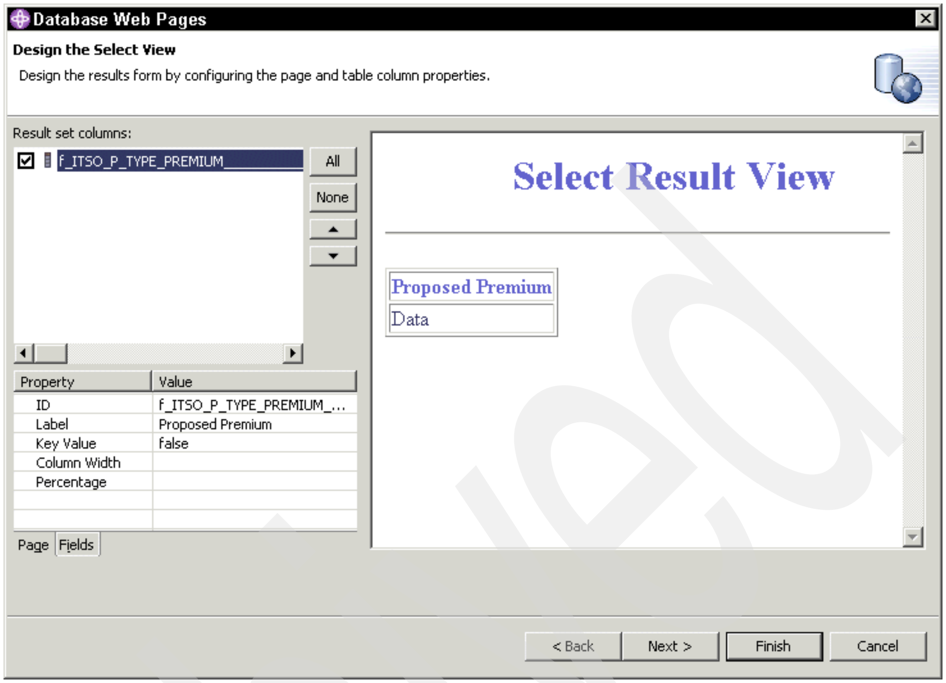
On the Page tab, you could change the heading (`Input Form`).

Click **Next**.

9. In Design the Select View page of the wizard, select the entry in *Result set columns* and change the Label field to `Proposed Premium` (Figure 16-29 on page 517).

   On the Page tab, you could change the heading (`Select Result View`).

   Click **Next**.



Database Web Pages

Design the Select View

Design the results form by configuring the page and table column properties.

Result set columns:

☑ f_ITSO_P_TYPE_PREMIUM

Property	Value
ID	f_ITSO_P_TYPE_PREMIUM_...
Label	Proposed Premium
Key Value	false
Column Width	
Percentage	

Select Result View

Proposed Premium
Data

Figure 16-29 Create Database Web Pages wizard: Design the Select View

10. In the Specify Prefix page, you may change the suggested prefix for the generated HTML files, for example, `propPremium`. Click **Finish**.

It takes some time to generate the output files:

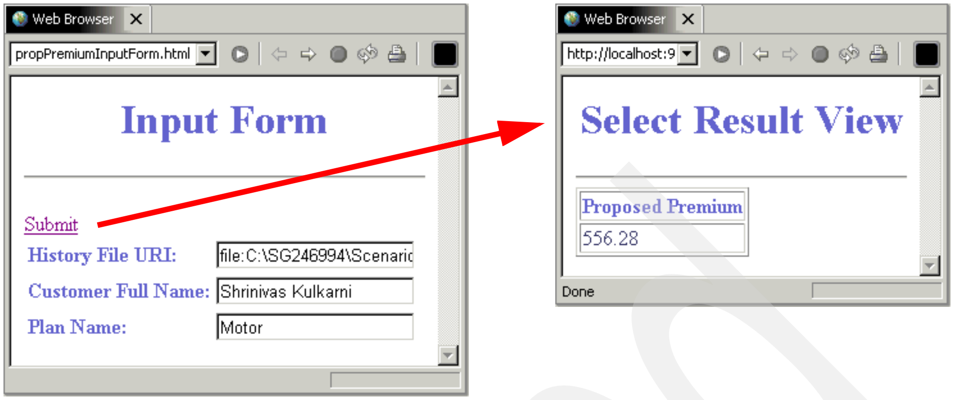
- Input HTML page: propPremiumInputForm.html
- Output JSP: propPremiumSelectView.jsp

Study the generated code. All the database access is in the JSP using the DB Beans tag library. If we had chosen another model (Figure 16-26 on page 514), we could have generated a servlet with JavaBeans for the database access.

16.4.3 Testing the Insura Web application

To test the generated database application, select the generated HTML input form, **propPremiumInputForm.html**, and **Run on Server** (context).

A sample run is shown in Figure 16-30 on page 518.



Input Form

Submit

History File URI: file:C:\SG246994\Scenaric

Customer Full Name: Shrinivas Kulkarni

Plan Name: Motor

Select Result View

Proposed Premium
556.28

Figure 16-30 Sample database application run

16.4.4 Summary

In this section, we showed how to build an SQL statement and then create a skeleton Web application from the SQL statement.

16.5 Insurance application as a Web service requestor

This section of the scenario deals with business case #3, that is, the Insurance application acting as a Web service requestor. This is also called a Web service consumer. In our example, we are creating a scalar Web service UDF returning the stock quote for a selected stock and a table Web service UDF returning headline information for the selected company.

Normally, you use the Application Developer tools to create DB2 Web service consumer UDFs by following these steps:

1. Enable DB2 Web service requestor capabilities.
2. Create the Web project that will contain the source files for the application.
3. Create Web service UDFs.
4. Build and run the generated Web service UDFs.
5. Use created Web service UDFs in SQL statements.

Because our INSURA database has been already enabled for XML Extender and SOAP UDFs have been registered, and the Web project ItsoDB2XMLInsuraWeb has been created as well, we can start our path by creating the UDFs.

16.5.1 Creating a scalar Web service UDF

First, we are creating a scalar Web service UDF returning a stock quote for the selected stock by using a freely available DelayedStockQuote Web service from the Web site:

```
http://www.xmethods.com
```

This Web service provides 20 minute delayed stock quotes. You can find the detailed description and the WSDL file of the Web service at:

```
http://services.xmethods.net/ve2/ViewListing.po?key=uuid:889A05A5-5C03-AD9B
-D456-0E54A527EDEE
```

To generate the scalar SOAP UDF in Application Developer:

1. Select **File -> New -> Other -> Data -> Web Service User-Defined Function** and click **Next** to start the Web service UDF wizard.
2. On the first page of the wizard, specify the WSDL file that will be used to generate the UDF. For the scenario, copy and paste the WSDL URL from Web site listed above, or enter the URL manually:

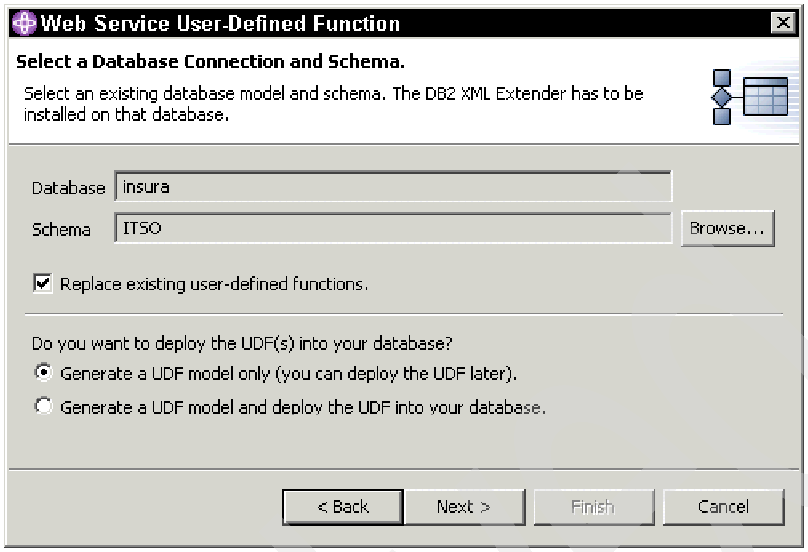
   ```
   http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl
   ```

   Click **Next**.
3. The next page of the wizard shows the database connection and a schema for which the UDF will be created (Figure 16-31 on page 520). Click **Browse** to locate the database schema. Click **ItsoDB2XMLInsuraWeb -> WebContent -> WEB-INF -> databases -> insura -> ITSO**. Make sure that the `insura` database also contains a `DB2XML` schema.

   > **Note:** The absence of the `DB2XML` schema in a database folder means that your DB2 XML Extender was not enabled by running the command:
   >
   > ```
   > dxxadm enable_db insura
   > ```

   Select the **ITSO** schema and click **OK**. If no connection to the specified database is available, a pop-up dialog will ask you for connection information.

   You can choose to deploy the generated UDF into the database immediately or generate a UDF model in the Application Developer workspace only. For this example, we are going to deploy the generated UDF later. Therefore, do not change the default options; click **Next**.



Web Service User-Defined Function

Select a Database Connection and Schema.

Select an existing database model and schema. The DB2 XML Extender has to be installed on that database.

Database: insura

Schema: ITSO

☑ Replace existing user-defined functions.

Do you want to deploy the UDF(s) into your database?

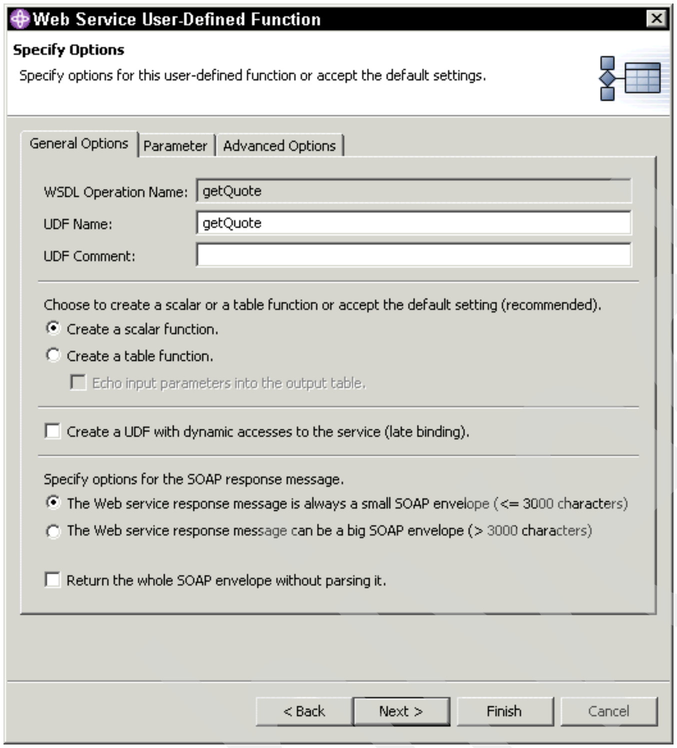
● Generate a UDF model only (you can deploy the UDF later).

○ Generate a UDF model and deploy the UDF into your database.

Figure 16-31 Create Web service UDF: database connection and schema

The *Select the user-defined functions (UDFs) that you want to create* page of the wizard shows operations described in the WSDL document. The wizard will generate one UDF for every operation that is selected. This Web service provides only one operation, `getQuote`, and it is automatically selected. Click **Next**.

4. The Specify Options page appears once for every operation selected in the previous page. It allows you to specify options for the UDF, such as changing the function name and providing a comment. This page also allows you to choose whether to build a scalar or a table function. Make sure that **Create a scalar function** option is selected on the General Options tab (Figure 16-32 on page 521).



Specify Options

Specify options for this user-defined function or accept the default settings.

WSDL Operation Name: getQuote

UDF Name: getQuote

UDF Comment:

Choose to create a scalar or a table function or accept the default setting (recommended).

● Create a scalar function.

○ Create a table function.

□ Echo input parameters into the output table.

□ Create a UDF with dynamic accesses to the service (late binding).

Specify options for the SOAP response message.

● The Web service response message is always a small SOAP envelope (<= 3000 characters)

○ The Web service response message can be a big SOAP envelope (> 3000 characters)

□ Return the whole SOAP envelope without parsing it.

Figure 16-32 Create Web service UDF: general options

Scalar or table function: By default, a scalar function is generated when a simple XML type is returned by the Web service. A table function is generated when a complex XML type is returned. The table function automatically maps the complex XML type into one or more columns.

Switching from a table function to a scalar function makes sense when the returned types should not be automatically mapped, but returned as an XML fragment.

On the other hand, switching from a scalar to a table function allows us to use the UDF in a FROM clause. It also makes it possible to include the input parameters as columns in the output table by selecting **Echo input parameters**.

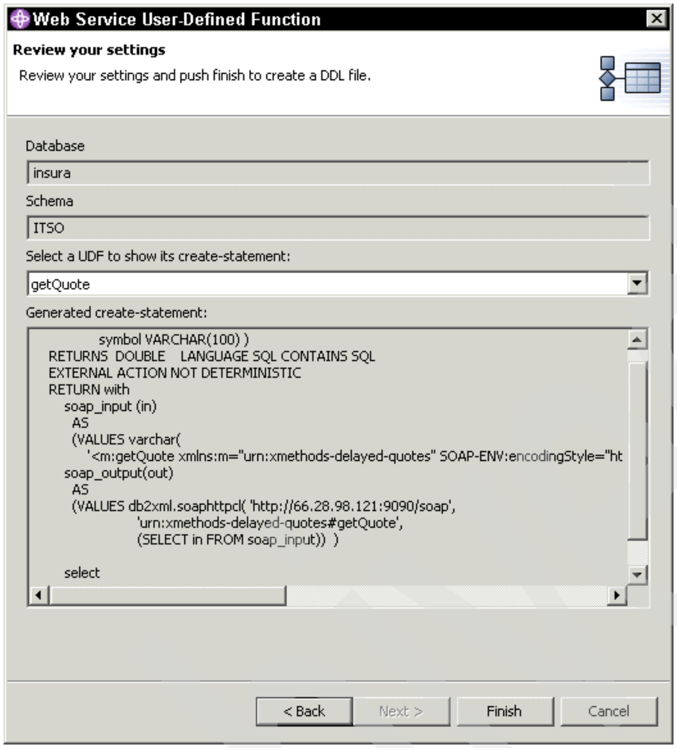
Additionally, you can choose to generate a UDF with dynamic access to the Web service. When the service location (the `location` attribute of the `soap:address` element) is not specified in the WSDL document, a dynamic function must be generated. You can select dynamic access even when the service location is specified, to make use of late binding. When a dynamic function is generated, you need to specify the service location at runtime as a parameter of the UDF.

When using a Web service that can return responses of more than 3000 characters, you have to select **big SOAP envelop**. By default, **small SOAP envelop** is selected because it results in better performance for most Web services. If we select **small SOAP envelop** and the Web service returns a SOAP message that is too big, the UDF will return a descriptive error message. It is also possible to not parse the SOAP envelope. This is useful for debugging purposes.

5. Select the **Parameter** tab to review the parameter and result mapping from WSDL types to DB2 SQL types.
6. Select the **Advanced Options** tab to specify the name for the UDF. If the name is not specified, a unique name is automatically generated by the database when the UDF is deployed. Leave the input text field blank and click **Next**.
7. The Review Your Settings summary page of the wizard shows the database and schema and it displays the create statement that will be issued against the database (Figure 16-33 on page 523):

```
CREATE FUNCTION ITSO.getQuote (
                 symbol VARCHAR(100) )
    RETURNS  DOUBLE    LANGUAGE SQL CONTAINS SQL
    EXTERNAL ACTION NOT DETERMINISTIC
    RETURN with
        soap_input (in)
          AS
          (VALUES varchar(
              '<m:getQuote xmlns:m="urn:xmethods-delayed-quotes"
                   SOAP-ENV:encodingStyle=
                          "http://schemas.xmlsoap.org/soap/encoding/">' ||
                '<symbol xsi:type="xsd:string">' || symbol|| '</symbol>' ||
              '</m:getQuote>') ),
        soap_output(out)
          AS
          (VALUES db2xml.soaphttpv( 'http://66.28.98.121:9090/soap',
                           'urn:xmethods-delayed-quotes#getQuote',
                           (SELECT in FROM soap_input))  )
        select
               db2xml.extractDouble(db2xml.xmlclob(x.out), '/*/*')
        from soap_output x
```

By clicking **Finish**, the UDF is generated. If you selected **Generate and deploy** on the Select Database Connection and Schema page of the wizard (Figure 16-31 on page 520), clicking **Finish** would also deploy the UDF.



Web Service User-Defined Function
Review your settings
Review your settings and push finish to create a DDL file.

Database: insura

Schema: ITSO

Select a UDF to show its create-statement: getQuote

Generated create-statement:

```
            symbol VARCHAR(100) )
    RETURNS  DOUBLE    LANGUAGE SQL CONTAINS SQL
    EXTERNAL ACTION NOT DETERMINISTIC
    RETURN with
       soap_input (in)
         AS
         (VALUES varchar(
           '<m:getQuote xmlns:m="urn:xmethods-delayed-quotes" SOAP-ENV:encodingStyle="ht
       soap_output(out)
         AS
         (VALUES db2xml.soaphttpcl( 'http://66.28.98.121:9090/soap',
                    'urn:xmethods-delayed-quotes#getQuote',
                    (SELECT in FROM soap_input))  )

       select
```

Figure 16-33   Create Web service UDF: summary page

The generated `getQuote` UDF is placed in the User-Defined Functions folder of the `insura` database under the `ITSO` schema in the Data Definition view.

Building and testing the scalar UDF

You can run the Web Service UDF directly from the workspace. If the UDF was generated but not deployed to the database, you first have to build the UDF:

1. In the Data Definition view of the Data perspective, expand the ItsoDB2XMLInsuraWeb project, select the **getQuote** UDF and **Build** from the context menu (Figure 16-34 on page 524).

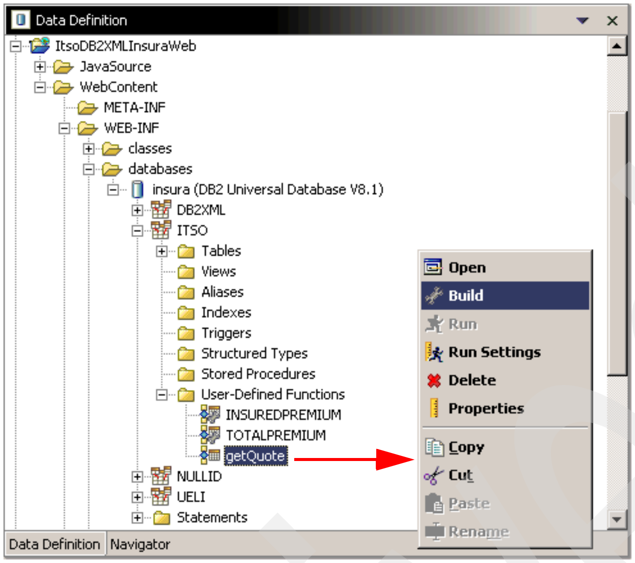
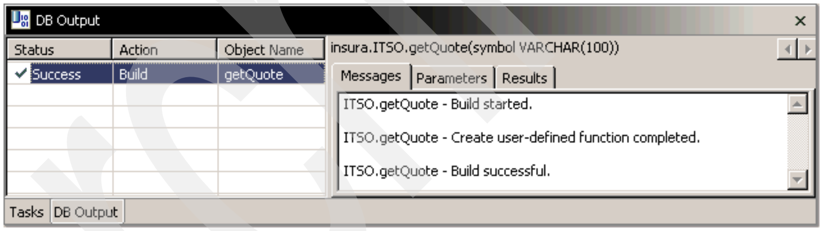


Figure 16-34 Building Web service DB2 UDF

2. Building a UDF takes some time. Once you get the `Success` status in the DB Output view, the UDF is built and deployed to the database (Figure 16-35).



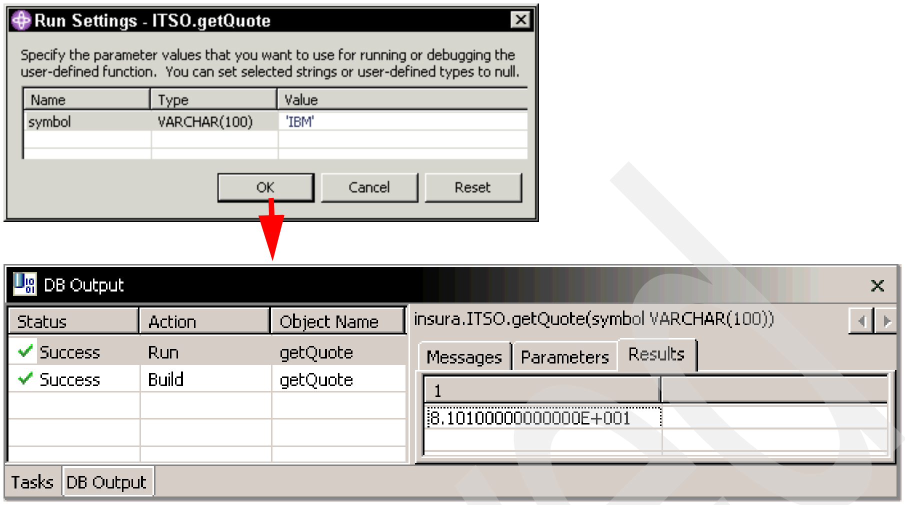
Status	Action	Object Name
✓ Success	Build	getQuote

insura.ITSO.getQuote(symbol VARCHAR(100))

```
ITSO.getQuote - Build started.

ITSO.getQuote - Create user-defined function completed.

ITSO.getQuote - Build successful.
```

Figure 16-35 DB2 Web service UDF: successful build

3. To test the deployed UDF, right-click the **getQuote** UDF in the tree and select **Run**. The *Run Settings* dialog is displayed. Enter a single-quoted value, for example, `'IBM'`, as an input parameter and click **OK**. The output is displayed in the DB Output view (Figure 16-36 on page 525).



Run Settings - ITSO.getQuote

Specify the parameter values that you want to use for running or debugging the user-defined function. You can set selected strings or user-defined types to null.

Name	Type	Value
symbol	VARCHAR(100)	'IBM'

DB Output

Status	Action	Object Name
✓ Success	Run	getQuote
✓ Success	Build	getQuote

insura.ITSO.getQuote(symbol VARCHAR(100))

1
8.10100000000000E+001

Figure 16-36 DB2 Web service UDF: testing

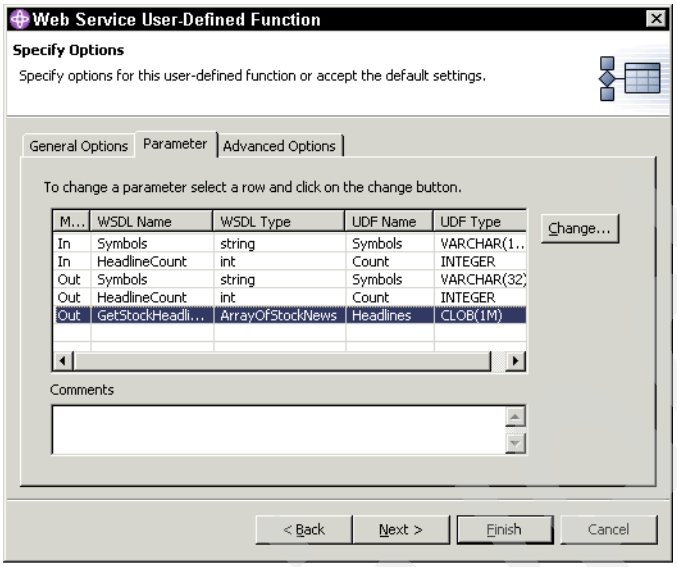
16.5.2 Creating a table Web service UDF

For the second UDF, we use another public function to query the headlines of a stock. The WSDL file is available at:

```
http://www.xignite.com/xnews.asmx?WSDL
```

Create the UDF in the same way as in “Building and testing the scalar UDF” on page 523, with these changes:

1. Select only the **GetStockHeadlines** function.
2. Set the name as `GetHeadlines` (we already have a `GetStockheadlines` UDF from 10.3.7, “Creating a wrapper UDF to Web services consumer function” on page 301.
3. Select **Create a table function** and **Echo input parameters** in the output table (Figure 16-32 on page 521).
4. On the Parameters page, change the two IN parameter names to `Symbols` and `Count`, and the output result to `Headlines` (Figure 16-37 on page 526).



Specify Options

Specify options for this user-defined function or accept the default settings.

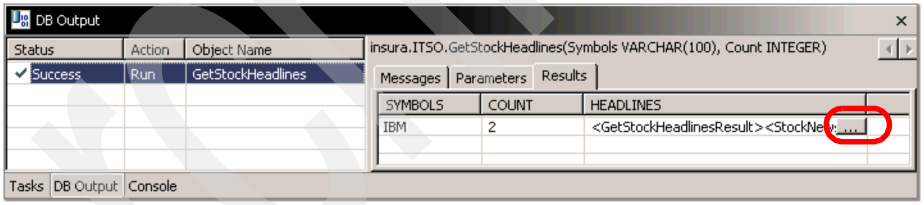
To change a parameter select a row and click on the change button.

M...	WSDL Name	WSDL Type	UDF Name	UDF Type
In	Symbols	string	Symbols	VARCHAR(1..
In	HeadlineCount	int	Count	INTEGER
Out	Symbols	string	Symbols	VARCHAR(32)
Out	HeadlineCount	int	Count	INTEGER
Out	GetStockHeadli...	ArrayOfStockNews	Headlines	CLOB(1M)

Comments

Figure 16-37 UDF table function parameters

5. Build and test the UDF. The result is a table of three columns, with the stock symbol, the number of headlines, and an XML document with the headlines (Figure 16-38).



Status	Action	Object Name
✓ Success	Run	GetStockHeadlines

insura.ITSO.GetStockHeadlines(Symbols VARCHAR(100), Count INTEGER)

SYMBOLS	COUNT	HEADLINES
IBM	2	<GetStockHeadlinesResult><StockNe ...

Figure 16-38 UDF table function result

6. In the HEADLINES column, click **...** to see the full result:

```
<GetStockHeadlinesResult>
<StockNews>
   <Outcome>Success</Outcome>
   <Headline>IBM Continues to Advance Linux Leadership, ...</Headline>
   <Ticker>IBM</Ticker><Date>8/13/2003</Date><Time>4:03 pm</Time>
   <Source>Market Wire</Source>
   <Url>http://biz.yahoo.com/iw/030813/056429.html</Url>
</StockNews>
<StockNews>......</StockNews>
</GetStockHeadlinesResult>
```

16.5.3 Creating a Web client that uses the UDFs

The objective of the Web client is to use the two scalar functions to investigate a stock. This can be performed in one SQL statement (the parameters are shown in bold):

```
select itso.getquote('IBM') as quote, t.symbols, t.count,
       substr(t.headlines,1,1000) as headlines
  from table(itso.getheadlines('IBM',2)) as t
```

Note. The headlines column returns a `CLOB(1M)`, not very suitable to display. The `substr` function can be used to limit the output to a given number of characters.

The Web service requestor client application consists of these parts:

- `StockWatch.java`—a servlet that executes the SQL statement and prepares an XML string for the output JSP.
- `StockWatch.html`—an HTML input page to enter a stock symbol and a headline count, then the servlet is invoked.
- `StockWatch.jsp`—an output JSP that uses XSL to translate XML into HTML.
- `StockWatch.xsl`—an XSL stylesheet for the transformation.

The four parts are provided in c:\SG246994\Scenario\wsad.

Import the `StockWatch` servlet into an `itso.stock` package (under `JavaSource`). Open the Web deployment descriptor (`web.xml`) and on the Servlets page click **Add** to add the servlet. Select the new servlet and click **Add** for URL mappings to define a mapping of `/StockWatch`.

Import the other three files (HTML, JSP, and XSL) into the WebContent folder.

Open the four parts and study the code:

1. The HTML file passes the two input fields as `symbol` and `count` to the servlet.
2. The servlet accepts the parameters (`symbol`, `count`) from the HTML, then prepares the XML output string, calls a method to execute the SQL statement, and completes the XML string:

```
<xsd1:Stockwatch ........>
   <Symbol>IBM</Symbol>
   <Count>2</Count>
   <Quote>81.1999969482421875</Quote>          <=== output from scalar UDF
   <GetStockHeadlinesResult>                   <=== output from table UDF
      <StockNews> ........ </StockNews>        <=== output from table UDF
   </GetStockHeadlinesResult>                  <=== output from table UDF
</xsd1:Stockwatch>
```

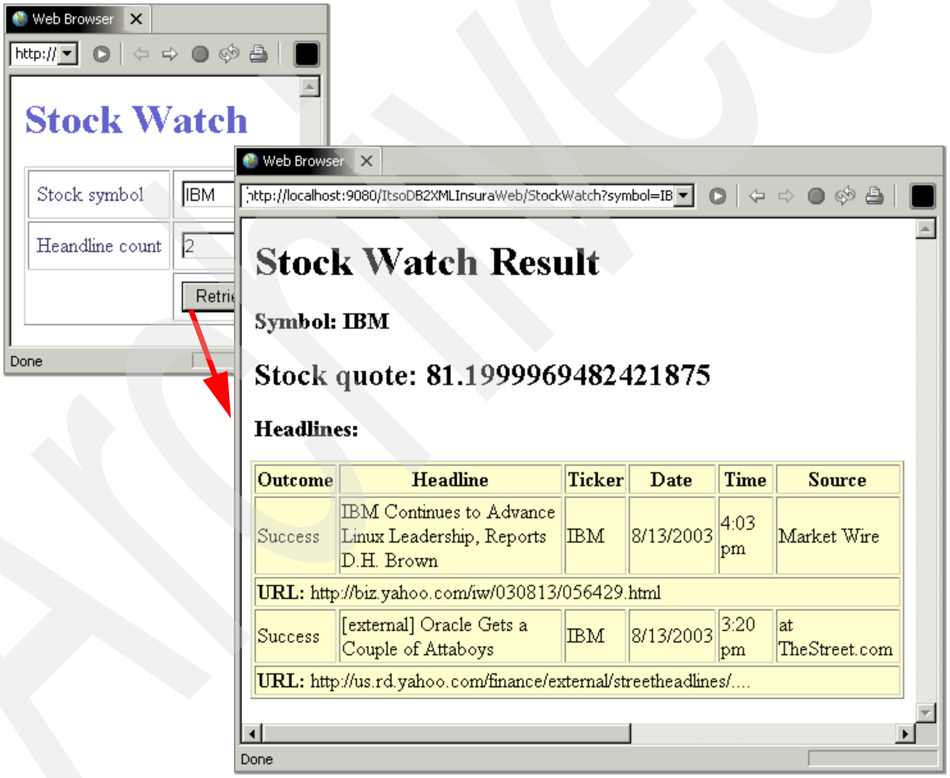
3. The JSP displays the XML output using XSL:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="StockWatch.xsl"?>
<%= request.getAttribute("result") %>
```

4. The XSL file was created in a similar way to that depicted in 16.3.3, “Creating a real client using XSL transformation” on page 505:
   - The parameters and the stock quote are displayed as headings.
   - The headlines are displayed in a table. Because of the size of the URL column it is formatted as a separate line.

Running the client application

A sample run of the client application is shown in Figure 16-39.


Web Browser
http://
Done
Web Browser
http://localhost:9080/ItsoDB2XMLInsuraWeb/StockWatch?symbol=IB
Done

Stock Watch

Stock symbol IBM

Headline count 2

Retrie

Stock Watch Result

Symbol: IBM

Stock quote: 81.1999969482421875

Headlines:

Outcome	Headline	Ticker	Date	Time	Source
Success	IBM Continues to Advance Linux Leadership, Reports D.H. Brown	IBM	8/13/2003	4:03 pm	Market Wire
URL: http://biz.yahoo.com/iw/030813/056429.html					
Success	[external] Oracle Gets a Couple of Attaboys	IBM	8/13/2003	3:20 pm	at TheStreet.com
URL: http://us.rd.yahoo.com/finance/external/streetheadlines/....					

Figure 16-39 Web client invoking Web service UDFs

Note: The servlet uses a data source (`jdbc/insura`). If the data source is not defined then a JDBC driver must be used.

16.5.4 Summary

In this section, we showed how to generate Web service UDFs with Application Developer and use them to access Web services from database application. With the Web services UDF wizard, you can quickly generate and test Web service UDFs. The generated UDFs facilitate access to dynamic data sources and allow them to be integrated into DB2 applications.

Restriction: It is currently not possible in Application Developer to build and use SQL statements that use UDFs. It is possible to use such SQL statements in Java code.

17

Deployment of DB2 Web services

This chapter explains how to deploy an enterprise application with Web services to a WebSphere Application Server V5.0.

In this chapter, we tell you how to:

- Export an enterprise application to an EAR file
- Set up a J2C authentication alias
- Create a data source using the authentication alias
- Deploy the application to the default server and start it
- Test the deployed Web applications

For detailed information about WebSphere Application Server, see the IBM Redbook *IBM WebSphere Application Server V5.0 Handbook*, SG24-6195.

17.1 Preparing the WebSphere Application Server

Running DB2 Web services in a WebSphere Application Server requires the Web Services Object Runtime Framework (WORF) runtime. See 10.2.2, “Installing WORF on IBM WebSphere Application Server” on page 279 for details.

In short:

- Locate dxxworf.zip in your <db2home>\samples\java\Websphere directory, where <db2home> is the directory where DB2 is installed.
- Unzip the dxxworf.zip to any directory.
- Copy worf.jar from the unzipped lib subdirectory to <washome>\lib, where <washome> is the directory where WebSphere Application Server is installed.

17.2 Deployment without Application Developer

If you manually create a Web application with DB2 Web services and you want to deploy the application to a WebSphere server, follow the instructions in 10.2.4, “Deploying the Web application” on page 287.

17.3 Deployment with Application Developer

In this section, we describe in detail how to deploy the `ItsoDB2XMLInsura` and `ItsoDB2XMLBroker` enterprise applications developed in Chapter 16, “DB2 Web services and XML with Application Developer” on page 475.

The major steps in this process are:

1. Export the enterprise applications from Application Developer.
2. Configure the application server.
3. Install the enterprise applications in the server.
4. Start the enterprise applications.

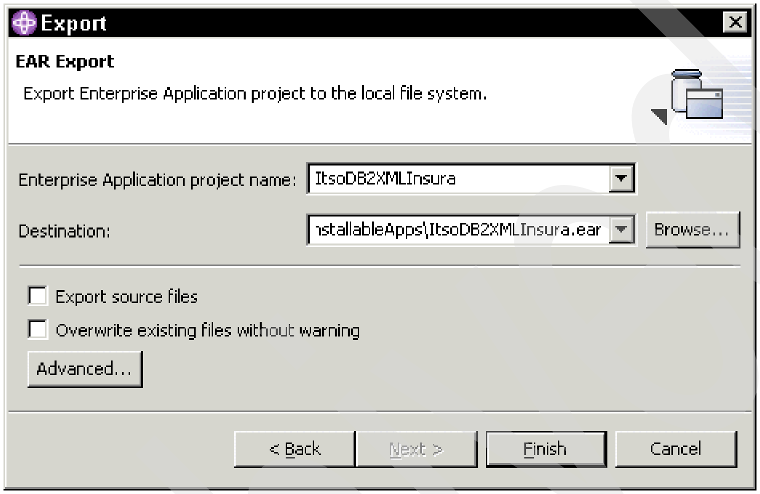
17.3.1 Exporting the enterprise applications

After having tested the enterprise applications in Application Developer, we can install them on a real WebSphere Application Server.

Make sure that the internal WebSphere servers are stopped in Application Developer, if you are installing the applications on the same machine. By default, the external and internal servers use the same ports.

To export an enterprise application, select the application, for example **ItsoDB2XMLInsura**, in the Project Navigator or J2EE Hierarchy view, and click **File -> Export** (or **Export** from the context menu). The Export dialog opens:

1. Select the **EAR** file and click **Next**.
2. Set the destination file on your local system (Figure 17-1).



EAR Export

Export Enterprise Application project to the local file system.

Enterprise Application project name: ItsoDB2XMLInsura

Destination: ıstallableApps\ItsoDB2XMLInsura.ear

□ Export source files

□ Overwrite existing files without warning

Figure 17-1 Exporting an enterprise application

   – If you deploy to the same machine, you can export directly into the <washome>\installableApps directory.
   – If you deploy to a remote machine, copy the exported EAR file into the <washome>\installableApps directory on the remote machine.
   – To install an enterprise application, the source files are not required.

3. Click **Finish**.

Repeat the export for any other enterprise applications (`ItsoDB2XMLBroker`) that you want to deploy.

17.3.2 Configuring the application server

Configuration of the application server is required if your enterprise applications use data sources or other facilities (JMS) that you had to configure on the test server.

To configure the application server, follow these steps:

1. Start the WebSphere server.

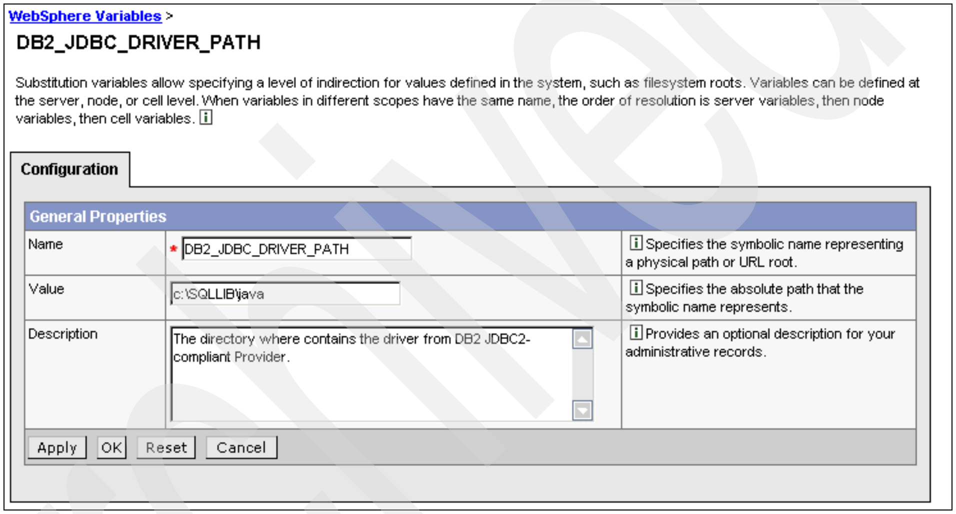
2. Start the WebSphere Administrative Console from the WebSphere program group or by typing the following URL in the Web browser's address bar:

   ```
   http://localhost:9090/admin
   ```

3. Enter your user ID and the Administrative Console opens (Figure 10-4 on page 287).

JDBC driver path

1. Select **Environment** (left side) and **Manage WebSphere Variables**.
2. Select **DB2 JDBC DRIVER PATH**.
3. Verify or enter the directory of the db2java.zip file (`c:\SQLLIB\java`).
4. Click **OK** (Figure 17-2).



WebSphere Variables >

DB2_JDBC_DRIVER_PATH

Substitution variables allow specifying a level of indirection for values defined in the system, such as filesystem roots. Variables can be defined at the server, node, or cell level. When variables in different scopes have the same name, the order of resolution is server variables, then node variables, then cell variables.

Configuration

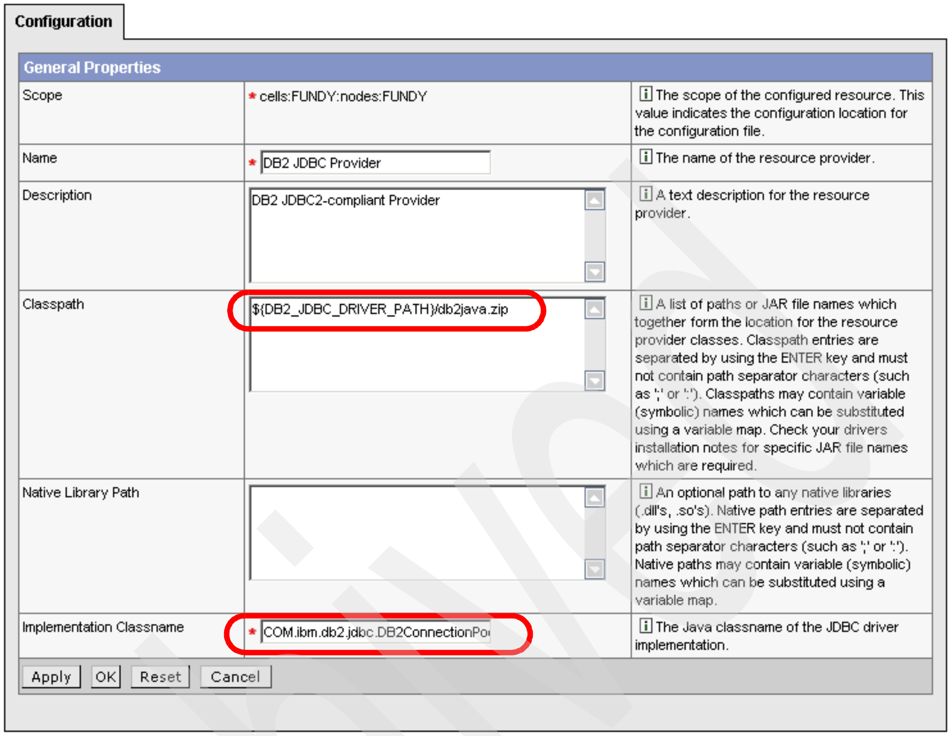
General Properties

Name	* DB2_JDBC_DRIVER_PATH	Specifies the symbolic name representing a physical path or URL root.
Value	c:\SQLLIB\java	Specifies the absolute path that the symbolic name represents.
Description	The directory where contains the driver from DB2 JDBC2-compliant Provider.	Provides an optional description for your administrative records.

Figure 17-2 Verifying the JDBC driver path

JDBC driver

1. Select **Resources** (left side) and select **JDBC Providers**.
2. Select **Node** (should be preselected)
3. Click **New** (under JDBC Providers):
   a. Select **DB2 JDBC Provider** from the pull-down and click **Apply**.
   b. All the defaults should be fine. Check that the implementation class is `COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource`. Note that the classpath entry points to `${DB2_JDBC_DRIVER_PATH}/db2java.zip` (Figure 17-3 on page 535).
   c. Click **Apply**.



Configuration

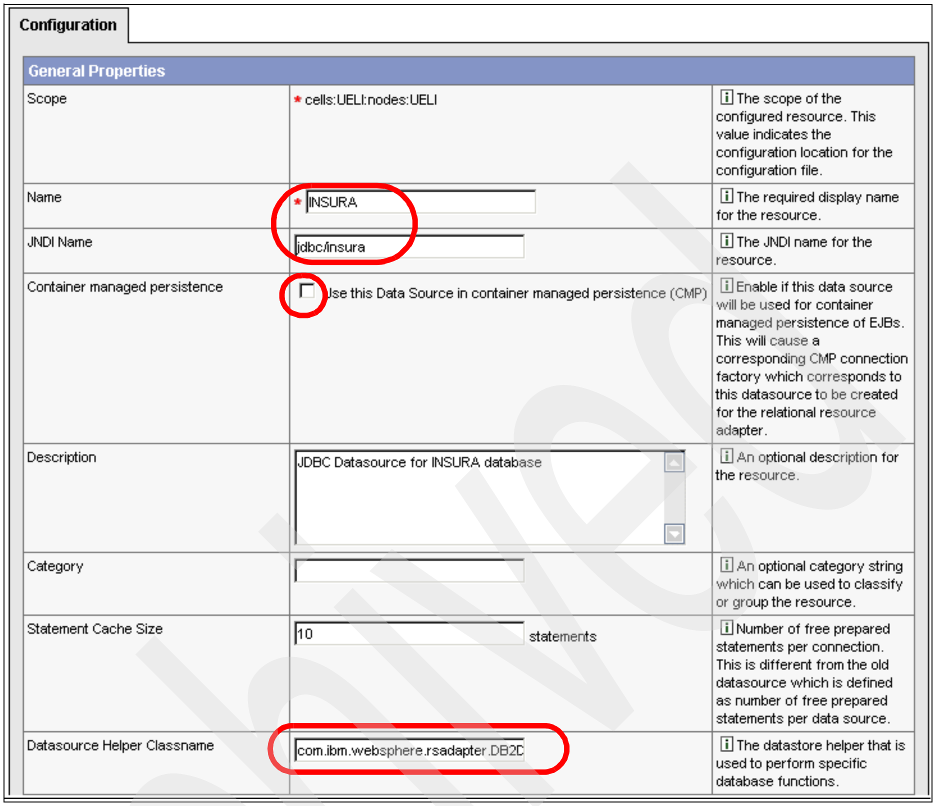
General Properties

Scope	* cells:FUNDY:nodes:FUNDY	The scope of the configured resource. This value indicates the configuration location for the configuration file.
Name	* DB2 JDBC Provider	The name of the resource provider.
Description	DB2 JDBC2-compliant Provider	A text description for the resource provider.
Classpath	${DB2_JDBC_DRIVER_PATH}/db2java.zip	A list of paths or JAR file names which together form the location for the resource provider classes. Classpath entries are separated by using the ENTER key and must not contain path separator characters (such as ';' or ':'). Classpaths may contain variable (symbolic) names which can be substituted using a variable map. Check your drivers installation notes for specific JAR file names which are required.
Native Library Path		An optional path to any native libraries (.dll's, .so's). Native path entries are separated by using the ENTER key and must not contain path separator characters (such as ';' or ':'). Native paths may contain variable (symbolic) names which can be substituted using a variable map.
Implementation Classname	* COM.ibm.db2.jdbc.DB2ConnectionPo	The Java classname of the JDBC driver implementation.

Figure 17-3 Defining a JDBC provider

Data source

1. Under Additional Properties select **Data Sources**.
   a. Click **New** (Figure 17-4 on page 536).
   b. Enter `INSURA` a the name and `jdbc/insura` as the JNDI name (to match what we used in the application).
   c. Deselect **Use this Data Source in container-managed persistence (CMP)**. This is for enterprise JavaBeans (EJB) only.
   d. The data source helper class should be filled in as `com.ibm.websphere.rsadapter.DB2DataStoreHelper`.
   e. Click **Apply**.

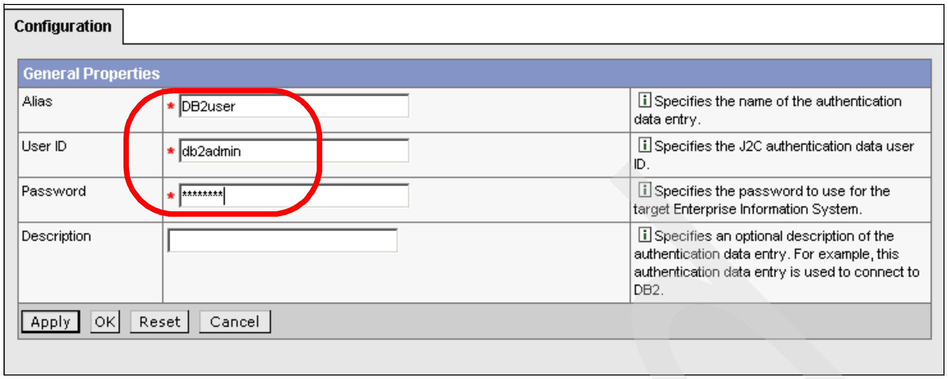


General Properties

Scope	* cells:UELI:nodes:UELI	The scope of the configured resource. This value indicates the configuration location for the configuration file.
Name	* INSURA	The required display name for the resource.
JNDI Name	jdbc/insura	The JNDI name for the resource.
Container managed persistence	□ Use this Data Source in container managed persistence (CMP)	Enable if this data source will be used for container managed persistence of EJBs. This will cause a corresponding CMP connection factory which corresponds to this datasource to be created for the relational resource adapter.
Description	JDBC Datasource for INSURA database	An optional description for the resource.
Category		An optional category string which can be used to classify or group the resource.
Statement Cache Size	10 statements	Number of free prepared statements per connection. This is different from the old datasource which is defined as number of free prepared statements per data source.
Datasource Helper Classname	com.ibm.websphere.rsadapter.DB2D	The datastore helper that is used to perform specific database functions.

Figure 17-4 Defining the data source for the EJBBANK database

2. Under Additional Properties, select **Custom Properties**, then select **databaseName**, enter `INSURA` as the value and click **OK**.
3. Go back to the data source (select **INSURA** at the top of the page). Under Related Items, select **J2C Authentication Data Entries**, then click **New** (Figure 17-5 on page 537):
   a. Enter `DB2user` as the alias.
   b. Enter a valid user ID and password, for example what was used to install DB2.
   c. Click **OK**. The new alias appears in the list, prefixed with the node name.



Configuration

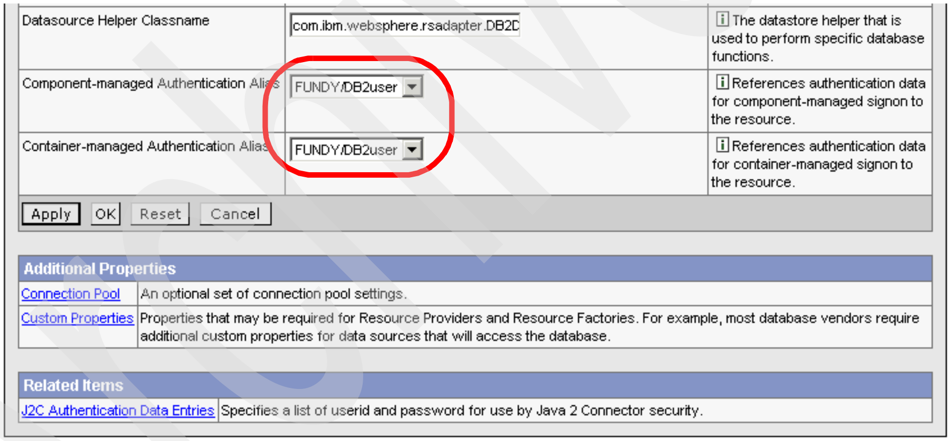
General Properties

Alias	* DB2user	Specifies the name of the authentication data entry.
User ID	* db2admin	Specifies the J2C authentication data user ID.
Password	* ********	Specifies the password to use for the target Enterprise Information System.
Description		Specifies an optional description of the authentication data entry. For example, this authentication data entry is used to connect to DB2.

Figure 17-5 Defining an authentication alias

4. Go back to the data source (select **INSURA** at the top of the page):

   For the component- and container-managed authentication alias, select the **{node}/DB2user** from the pull-down and click **OK** (Figure 17-6).



Datasource Helper Classname	com.ibm.websphere.rsadapter.DB2D	The datastore helper that is used to perform specific database functions.
Component-managed Authentication Alias	FUNDY/DB2user	References authentication data for component-managed signon to the resource.
Container-managed Authentication Alias	FUNDY/DB2user	References authentication data for container-managed signon to the resource.

Additional Properties

Connection Pool	An optional set of connection pool settings.
Custom Properties	Properties that may be required for Resource Providers and Resource Factories. For example, most database vendors require additional custom properties for data sources that will access the database.

Related Items

J2C Authentication Data Entries	Specifies a list of userid and password for use by Java 2 Connector security.

Figure 17-6 Defining container authentication for the data source

> **Note:** Due to Web browser caching, the alias may not appear in the pull-down list. If this happens, refresh your Web browser by pressing **F5** and then navigate to this entry again.

5. If you select **Security** (left side) and **JAAS Configuration**, you can find the new alias by selecting **J2C Authentication Data**.

Saving the configuration and restarting the server

1. Save the configuration (select **Save** in the toolbar). There should be no problems.
2. Close the administrative console (select **Logout** in the toolbar).
3. Stop the server, then start it again.

17.3.3 Installing the enterprise applications in the server

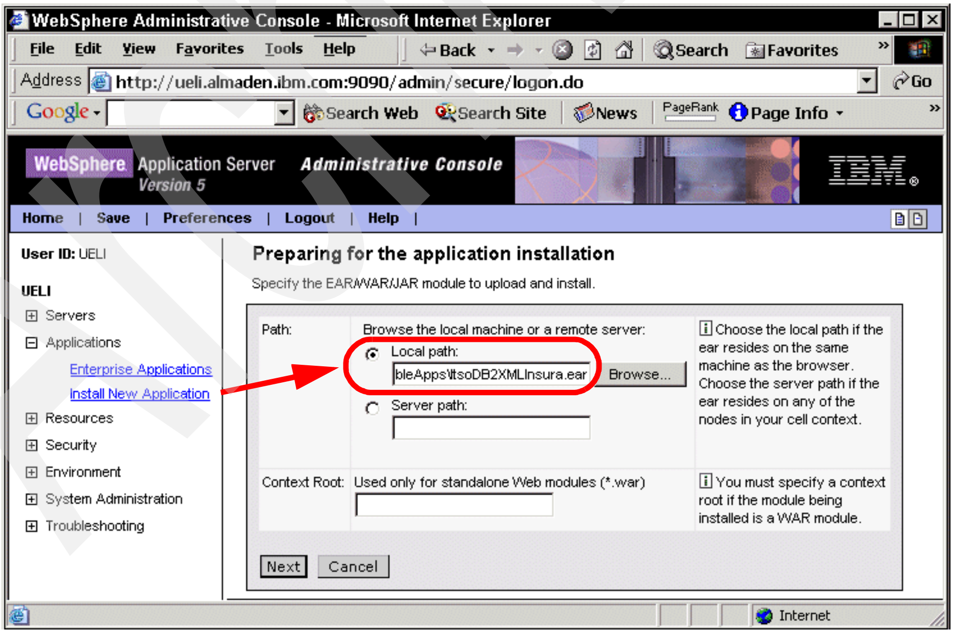
The server is now ready for installation of the enterprise applications. You have two options:

- Install the EAR files using the administrative console.
- Install the EAR files using commands.

Installing an EAR file using the administrative console

This process was described in detail in 10.2.4, “Deploying the Web application” on page 287. The only difference is that we point to the exported EAR file:

1. Start the administrative console.
2. Select **Install New Applications** under Applications. Provide the path of the exported EAR file (Figure 17-7).


WebSphere Administrative Console - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://ueli.almaden.ibm.com:9090/admin/secure/logon.do
Go
Google Search Web Search Site News PageRank Page Info
WebSphere Application Server Version 5 Administrative Console
IBM
Home | Save | Preferences | Logout | Help |
User ID: UELI
UELI
Servers
Applications
Enterprise Applications
Install New Application
Resources
Security
Environment
System Administration
Troubleshooting
Internet

Preparing for the application installation

Specify the EAR/WAR/JAR module to upload and install.

Path: Browse the local machine or a remote server:

● Local path: bleApps\ItsoDB2XMLInsura.ear Browse...

○ Server path:

Choose the local path if the ear resides on the same machine as the browser. Choose the server path if the ear resides on any of the nodes in your cell context.

Context Root: Used only for standalone Web modules (*.war)

You must specify a context root if the module being installed is a WAR module.

Next Cancel

Figure 17-7 Install an enterprise application

3. Click **Next** and keep accepting the default values for the next panels. Click **Finish** on the summary panel.
4. Click **Save to master configuration**. Click **Save** on the confirmation panel.

Installing an EAR file using commands

WebSphere Application Server provides the wsadmin facility which can be used with commands to perform all kinds of configuration tasks. For example, we can install an enterprise application using commands:

1. Open a command window and navigate to the <washome>\bin directory.
2. Enter `wsadmin`.
3. To install the `ItsoDB2XMLBroker` enterprise application, enter:

```
$AdminApp install <washome>/installableApps/ItsoDB2XMLBroker.ear
$AdminConfig save
quit
```

   Be sure to use forward slashes! Enter the fully qualified name for `<washome>`, for example, `c:/WebSphere/AppServer`.

17.3.4 Starting the enterprise application

You can start the enterprise application using the administrative console as described in 10.2.5, “Starting the enterprise application” on page 290:

1. Select **Enterprise Applications** under Applications.
2. In the Enterprise Applications panel, select the **ItsoDB2XMLInsura** application and click **Start**.

> **Note:** You have to log out and log in to see the application that you installed using commands.

An alternative way is to install an enterprise application using the wsadmin facility:

1. Start wsadmin in a command window.
2. To start the `ItsDB2XMLBrokers` enterprise application, enter (on one line), where `mycell` and `mynode` are your machine name (you can find the names under <washome>\config\cells):

```
set appManager [$AdminControl queryNames
        cell=mycell,node=mynode,type=ApplicationManager,process=server1,*]
```

Then enter:

```
$AdminControl invoke $appManager startApplication ItsoDB2XMLBroker
quit
```

17.4 Running the Web services applications

Open a browser to test the installed applications. You can use these URLs to test the various applications:

- Web services from SQL statements:

  ```
  http://localhost:9080/ItsoDB2XMLInsuraWebClient/sample/insura
                                                              /TestClient.jsp
  ```

  Enter `Health` for *policyQuote* and `456` for *brokerSales*.

- Proposed premium:

  ```
  http://localhost:9080/ItsoDB2XMLInsuraWeb/propPremiumInputForm.html
  ```

  Enter these parameter values:

  ```
  file:C:\SG246994\Scenario\InsuranceHistory.xml
  Shrinivas Kulkarni
  Motor
  ```

- Stock watch:

  ```
  http://localhost:9080/ItsoDB2XMLInsuraWeb/StockWatch.html
  ```

  Enter a stock symbol (`IBM`) and number of headlines (**3**).

- Broker test client and real client:

  ```
  http://localhost:9080/ItsoDB2XMLBrokerWebClient/sample/insuraPolicyQuote
                                                              /TestClient.jsp
  http://localhost:9080/ItsoDB2XMLBrokerWebClient/client/ClientInput.html
  ```

  Enter `Health` as parameter.

17.5 Using a real HTTP server

In the tests, we used `http://localhost:9080` to run the applications. This uses a WebSphere internal HTTP server.

To use a real HTTP server on port 80, you have to configure the HTTP server with a WebSphere plugin:

1. Start the administrative console.
2. Expand Environment and select **Update Web Server Plugin**.

3. Click **OK** to update the plugin. You will see a confirmation message at the top.
4. Stop and restart the HTTP server to activate the plugin.

Part 5

Appendixes

A

Installation

This appendix describes the installation prerequisites for:

- DB2 UDB V8
- DB2 Information Integrator V8
- WebSphere Application Server V5
- WebSphere Studio Application Developer V5.1

as well as how to install these software products.

DB2 V8.1 with FixPak 2

Installing DB2 UDB V8.1 for Linux, UNIX and Windows, is straightforward. Just follow the installation panels. You do not have to install all the components, such as data warehousing and performance tools.

After the base installation, you have to install Fixpack 2 or later.

Suggested installation directory: **`C:\SQLLIB`**

Starting with DB2 UDB for Linux, Unix, and Windows V8, XML Extender installs with the base DB2 installation. No extra seps are required to install the XML Extender code.

XML Extender is also a free, separately installable component, of DB2 for z/OS V7 and later. For installation instructions for DB2 XML Extender for z/OS and OS/390®, see *DB2 UDB for z/OS V8 XML Extender Administration and Programming*,SC18-743, as well as the following document with installtion hints and tips on the Web:

http://www.ibm.com/software/data/db2/extenders/xmlext/docs/v71390/xmlehints.pdf

Enabling a database for XML Extender

However, even with V8, you still have to enable a database for the usage of XML Extender.

To enable a database for XML Extender run this command in a DB2 command window:

```
dxxadm enable_db database_name
dxxadm enable_db insura                                <=== scenario sample
```

Enabling a database with Web service consumer UDFs

The database you want to enable for DB2 Web services consumer must be first enabled for DB2 XML Extender.

To enable the Web service consumer UDFs, issue the following command from the DB2 command line to register five user-defined functions:

```
db2enable_soap_udf -n dbName [-u uID] [-p password] [-force]
db2enable_soap_udf -n insura -u db2admin -p db2admin   <=== scenario sample
```

The parameters have the following definitions:

- *dbName*—The database name to be enabled
- *uID* user—D for accessing the database
- *password*—The password associated with the user ID
- *-force*—Attempts to drop any existing functions.

The **enable** command copies the shared library with the implementation of the functions to your `sqllib/function` directory, and then it enables your database to use the functions.

To make sure that you enabled the database correctly ensure that you have the following UDFs defined.

```
db2xml.soaphttpv (VARCHAR(256), VARCHAR(256), VARCHAR(3072))
db2xml.soaphttpv (VARCHAR(256), VARCHAR(256), CLOB(1M))
db2xml.soaphttpc (VARCHAR(256), VARCHAR(256), varchar(3072))
db2xml.soaphttpc (VARCHAR(256), VARCHAR(256), CLOB(1M))
db2xml.soaphttpcl ( VARCHAR(256), VARCHAR(256), varchar(3072))
```

And also ensure that `db2soapudf.dll` is copied to `sqllib/function` directory of your DB2 installation.

You can disable the UDFs by executing the following command from DB2 command line:

```
db2disable_soap_udf  -n dbName -u uID -p password
```

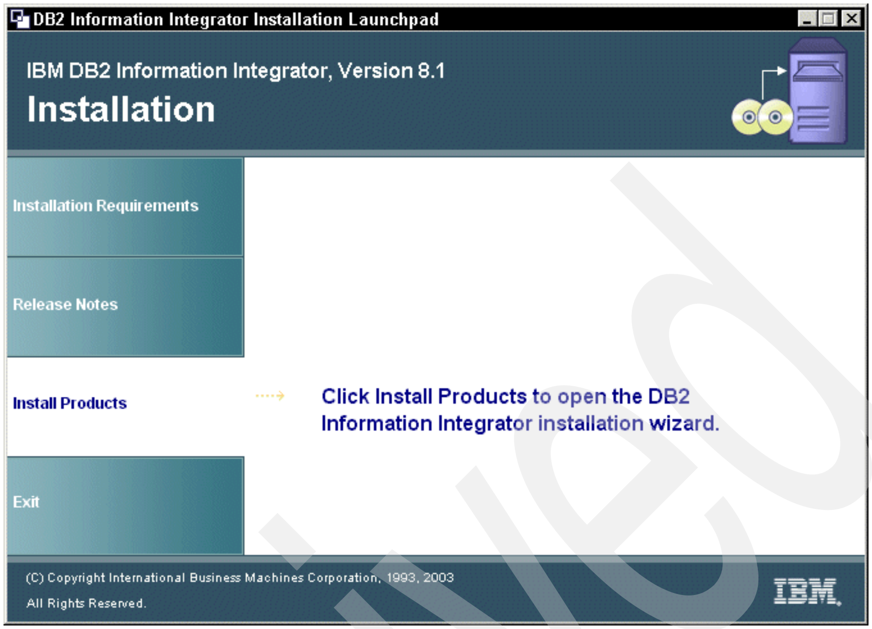
The meanings of the parameters are the same as above.

Refer to DB2 Web services consumer documentation available in *DB2 Information Integrator V8 Developer's Guide,* SC18-7359.

DB2 Information Integrator

DB2 Information Integrator installs on top of DB2 V8.1. It is required if you want to use the XML Wrapper.

Start the process by running the `iiSetup.exe`, and the is installation panel is displayed (Figure A-1 on page 548).


DB2 Information Integrator Installation Launchpad
IBM DB2 Information Integrator, Version 8.1
Installation
Installation Requirements
Release Notes
Install Products
Exit
IBM

Click Install Products to open the DB2 Information Integrator installation wizard.

(C) Copyright International Business Machines Corporation, 1993, 2003
All Rights Reserved.

Figure A-1 Installation of Information Integrator

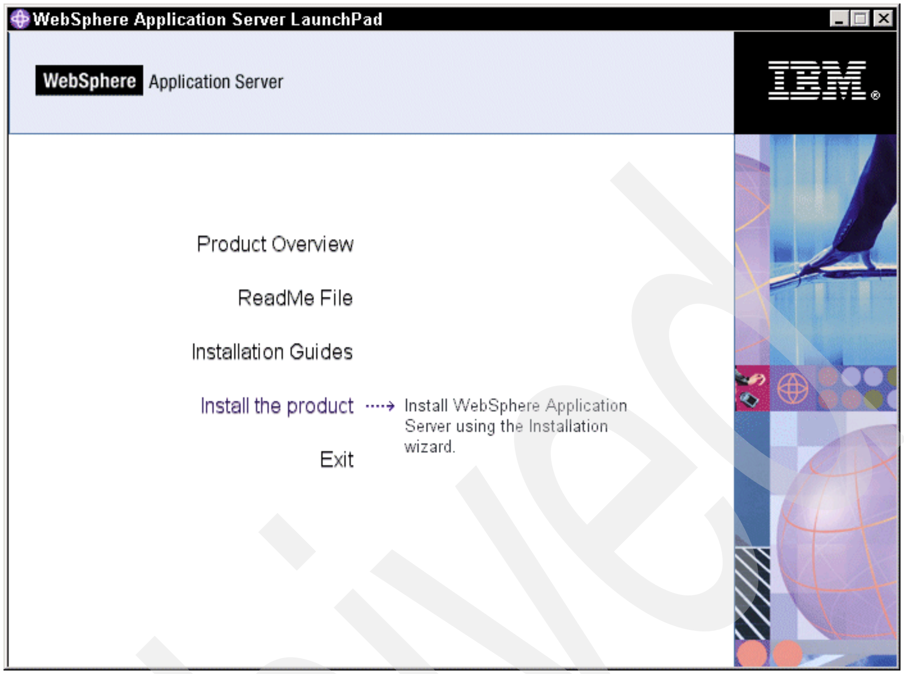
Click **Install Products** and the installation starts:

- You have a choice to install *relational wrappers* or *non-relational wrappers*. For the XML Wrapper, it is enough to select non-relational wrappers only.
- For each selection, a new installation wizard starts to guide you through the installation of the wrapper.

WebSphere Application Server V5

First, start the LaunchPad (launchpad.bat) to access the product overview, the ReadMe file, and installation guides.

1. Select **Install the product** to launch the installation wizard (Figure A-2 on page 549).



WebSphere Application Server LaunchPad

WebSphere Application Server

IBM

Product Overview

ReadMe File

Installation Guides

Install the product → Install WebSphere Application Server using the Installation wizard.

Exit

Figure A-2 WebSphere Application Server LaunchPad

2. In the first window panel, select the language and click **OK**.
3. Click **Next** in the Welcome panel.
4. After confirming that we agree with the license agreement, we have to decide between two installation choices: **Typical** and **Custom**. Typical installs the entire product, whereas the Custom installation option allows you to deselect components you do not want to install. We chose Typical installation.
5. The installation directories for the selected components are entered in the next panel. Suggested installation directories are:

```
C:\WebSphere\AppServer
C:\IBMHttpServer
C:\WebSphere MQ
```

6. In the following panel, enter a node name and host name or IP address. In addition, select to install both WebSphere Application Server and IBM HTTP Server as a service on Windows NT® or 2000.
7. After the Summary window, the installation starts.
8. The First Steps window is started automatically at the end of the installation.

Verifying the installation

Installation verification can be started from the menu. In Windows 2000, select **Start -> IBM WebSphere -> Application Server v5.0 -> First Steps**. Then select **Verify Installation**. We can also start the command **`ivc localhost`**.

If the install was successful, you should see messages similar to the following:

```
OS: Windows 2000
locale: en_US
hostname: NODENAME
cmd.exe /c "C:\WebSphere\AppServer\bin\ivt" NODENAME 9080
C:\WebSphere\AppServer
IVT0006I: Connecting to the WebSphere Server NODENAME on Port:9080
IVT0007I:WebSphere Server NODENAME is running on Port:9080
IVT0060I: Servlet Engine Verification: Passed
IVT0065I: JSP Verification: Passed
IVT0066I: EJB Verification: Passed
IVT00100I: IVT Verification Succeeded
IVT0015I: Scanning the file
D:\WebSphere\AppServer\logs\server1\SystemOut.log for
errors and warnings
IVT0020I: 0 Errors/Warnings were detected in the file
D:\WebSphere\AppServer\logs\server1\SystemOut.log
IVT0110I: Installation Verification is complete.
```

Installing FixPak 2

To install FixPak 2, follow these steps:

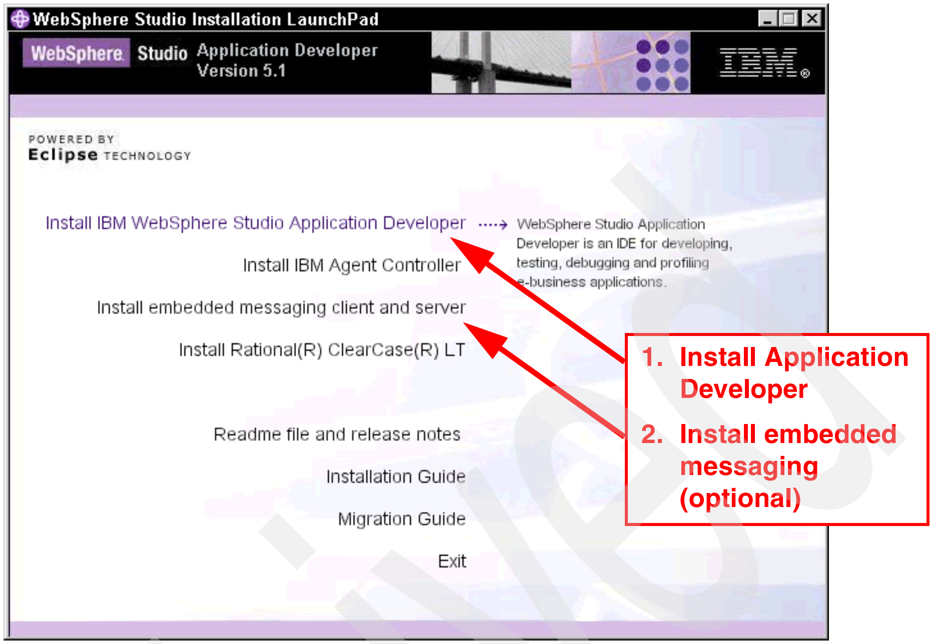
1. Open a command window.
2. Run the command **`c:\WebSphere\AppServer\bin\setupcmdline.bat`** (from the directory where WebSphere is installed)
3. Run the updateWizard.bat file (from the directory of the FixPak 2) to install the Fix Pack. Follow the instructions in the prompt.

WebSphere Studio Application Developer V5.1

We are using V5.1 of Application Developer.

The installation of the Application Developer is a very straightforward process. Perform the following steps:

1. Double-click **setup.exe** and the Installation Launcher window appears (Figure A-3 on page 551).



WebSphere Studio Installation LaunchPad

WebSphere Studio Application Developer Version 5.1

POWERED BY Eclipse TECHNOLOGY

Install IBM WebSphere Studio Application Developer ····→ WebSphere Studio Application Developer is an IDE for developing, testing, debugging and profiling e-business applications.

Install IBM Agent Controller

Install embedded messaging client and server

Install Rational(R) ClearCase(R) LT

Readme file and release notes

Installation Guide

Migration Guide

Exit

1. **Install Application Developer**
2. **Install embedded messaging (optional)**

Figure A-3 Application Developer Installation window

3. Select **Install IBM WebSphere Studio Application Developer**.
4. In the Welcome page, click **Next**.
5. In the License Agreement page, accept the agreement and click **Next**.
6. In the Destination Folder page, browse to a folder of your choice and then click **Next**. The suggested installation directory is:

   ```
   C:\WSAD51
   ```

7. In the Custom Setup page, accept the defaults and click **Next**.
8. In the next page, click **Install.**
9. After a rather long time period, the next page tells you about the success of the installation. Click **Finish**.
10. Start Application Developer. When asked to specify the location of the workspace, enter `c:\WSAD51sg246994`.

    Optionally, restart the installation launcher window and select **Install embedded messaging client and server**. This component is optional for JMS messaging to feed a message-driven bean using WebSphere MQ. This is not required for the examples in this publication.

11. If you installed WebSphere Application Server, installing the embedded messaging locates the existing WebSphere MQ installation and only tailors the Application Developer to use the existing code.
12. If WebSphere MQ is not installed on this machine, then it will be installed now.

B

Introduction to XML

This appendix provides an introduction to XML. We give a brief overview of XML, explain its business and technological benefits, and discuss basic XML concepts. At the end of this appendix we look at some examples of using XML in applications.

What is XML?

XML stands for eXtensible Markup Language. XML is a meta-markup language and is used for creating your own markup languages. Using XML, you can define the tags for your markup language. XML tags are used to describe the contents of the document. This means that any type of data can be easily defined using XML.

XML is universal not only by its range of applications but also by its ease of use: Its text-based nature makes it easy to create tools, and it is also an open, license-free, cross-platform standard, which means anyone can create, develop, and use tools for XML. What also makes XML universal is its power. Data is transmitted and stored in computers in many different ways: Originally it was stored in flat-files, with fixed-length or delimited formats, and then it moved into databases, and often into complex binary formats. XML is a structured data format, which allows it to store complex data, whether it is originally textual, binary, or object-oriented. To this day, very few data-driven technologies have managed to address all these different aspects in one package — except XML.

A brief comparison of XML and HTML

XML and HTML both descend from common roots. However, because XML is still fairly new, most data on the Web is still stored in HTML format. The amount of data currently stored on the Web is hard to imagine — for the latest survey (by OCLC Research) see the following Web site:

http://wcp.oclc.org

This survey indicates that there are 1.4 billion public pages on the Web (2002). In fact, the amount of data available may be much higher, because counting pages ignores the fact that one dynamically generated page can act as a gateway to a large database.

The problem with data available in HTML format is that it is formatted for people to view, and not for computers to use. HTML consists of a pre-defined set of tags, the purpose of which are known. This makes it a language that is easy to learn and accessible, but makes it hard to re-use the data.

This is where XML enters the picture. As its name indicates, XML is *extensible*, which means that you can define your own set of tags and make it possible for others (people or programs) to know and understand these tags. This makes XML much more flexible than HTML. In fact, because XML tags represent the logical structure of the data, they can be interpreted and used in various ways by different applications.

Much of the value of the Web comes from re-using data. For example, one of the great success stories of the Web are the search engines. They work on the basis of a universal communications method (HTTP), and a universal markup language (HTML), to catalog Web pages. However, search engines work on very limited information, because only a tiny part of an HTML document is designed to be used by a search engine. Imagine how much more powerful search engines could be if the data that they search was stored in a simple, structured, re-usable format.

XML business benefits

To date, XML has three main applications:

- Sharing of information between computer systems and between businesses
- Storage and transmission of information within a single application
- Content delivery — delivering information to users

The early usage of XML has been in the first two of these areas, where the benefits of XML are easiest to achieve.

Information sharing

The benefits of using XML to share information between computer systems and businesses are probably the greatest and easiest to achieve: XML allows businesses to define data formats in XML, and to easily build tools which read data, write data, and transform data between XML and other formats. This has allowed a number of businesses and industry consortiums to build standard XML data formats. Areas such as *Electronic Data Interchange* (EDI), inter-bank payments, supply-chain, trading, and document management are all the subject of ongoing XML-based standardization by industry consortiums.

By using XML, the standard can be published, extended, and easily used by new applications. Data transformation tools exist that can convert between existing formats and XML, and new tools are emerging. The ability to link enterprise applications, known as Enterprise Application Integration (EAI), is a key focus for many companies, and has produced cost savings and increased revenue for many enterprise customers. In particular, many businesses aim to improve Customer Relationship Management (CRM) by creating a single logical view of each customer across multiple existing systems. XML is an important technology to create this single customer view.

Furthermore, because XML makes it easy to relate structure to content, XML subsets can be defined with specific industries or applications in mind. For example, XML has been used to define standard data formats for the banking industry. In the same manner, a standard could be developed specifically for flight booking systems, thereby allowing airlines to easily exchange information.

XML inside a single application

The business benefits of using XML within a single application are less compelling, but they are very easy to achieve, and so we have seen a number of applications that use XML internally. The benefits of this approach are speed of development and future extensibility.

XML is a very powerful and flexible language for describing the complexities of the real world. Because XML can describe complex data, it means that it can be a powerful tool to creating new applications. The extensibility of XML means that the application can grow and develop without major changes. Finally, when the application needs to connect to other applications, XML makes an excellent way of linking the application with others.

Content delivery

XML has a number of key benefits for content delivery. The main benefits are the ability to support different users and *channels*, and to build more efficient applications. Channels are information delivery mechanisms — for example, Digital TV, phone, Web, and multimedia kiosk. Supporting different channels is an important step in delivering e-business applications to customers through their chosen medium. XML is a key technology for this.

For example, a customer and a supplier both need to access the same on-line product catalogue. Although the information is the same, the visual emphasis will differ, depending on who the user is: The customer will be more interested in looking for information on functionality, pricing, and availability, while the supplier will want to have easy access to catalog maintenance and inventory information. All this information might be stored in a single XML document and be displayed differently by the application.

Using XML for content delivery requires XML-enabled browsers. Microsoft Internet Explorer 5.0 was the first browser to support XML directly. As people become more familiar with this technology, this particular strength of XML is likely to be exploited more often.

Technological benefits of XML

In order to see the technological benefits of XML, let us consider an example.

An example of using XML

Many libraries offer their catalogs over the Web. Usually, there is a simple Web form where you enter a title, name, or subject, and you are presented with some search results. If you want to search several libraries, you need to go to each of their Web sites. It would be useful and convenient if there could be a single Web page which could search many libraries. To build that today would require extracting the data (title, author, ISBN, and so forth) from each search results page. However, each search results page is formatted differently, and the data is mixed in with presentation information. To collate the results of many searches would require complex programming for each libraries Web site, separating the data from the presentation.

Suppose, instead, that there were a single format for returning search results from libraries: Let us call it *A Book Catalog Markup Language* or ABCML. ABCML would define tags for the author, title, and so on, thus making it easy for a computer to extract the data. Building the meta-catalog suddenly becomes easier.

ABCML would also help the libraries, because they could re-use each other's software. Also, when a new book comes in, the publisher can provide the book's catalog information in ABCML, and save the librarian the effort of typing it into the catalog.

Major benefits

So ABCML would help us build a meta-catalog, and also help the libraries re-use existing data. Those arguments may not be enough to persuade a particular library's IT director to rewrite his online catalog. However, there are a number of further technical benefits to using XML. As well as re-using data, these benefits include: separating data from display, extensibility, and adding semantic content to the data.

Re-use of data

We have seen the benefit of re-using data: The librarian could re-use the publisher's data, because it was in a common format, and we could re-use the data when we built our meta-catalog. Of course there are many common file formats in the world of computing that have allowed data re-use. However, these have usually been proprietary and application specific. XML is neither of those.

Separation of data and display

What are the benefits of separating data and presentation? First, without this separation, we could not achieve the simple re-use of the data. Second, the presentation changes. If you look at the Web sites of 5 years ago, and the Web sites of today, they are radically different. If you look at any successful Web site, you will probably see at least one redesign a year. That is not simply because it is trendy — successful Web sites analyze and react to feedback from users, and redesign the site to be more productive and intuitive. Let us return to the library Web site — the Web site gets a redesign, but the underlying data remains in place — so it makes sense to separate the data output from the Web site design.

There is still another even more compelling reason to separate data and display: the rise of *pervasive computing*. Pervasive computing means that computing devices become integrated into common everyday appliances: mobile phones, televisions, printers, and palm computers. Each of these appliances may have a different display technology, and require different instructions on how to display the data. The same search of the library catalog should be viewable on a mobile phone or a high-resolution PC.

Extensibility

HTML has been a constantly evolving standard since it emerged, and one of the problems it has faced is that it has often been extended by companies wishing to go beyond HTML. Browser suppliers regularly add non-standard extensions to HTML. Similarly, Web server manufacturers build “server-side” extensions to HTML: These include NCSA includes, Microsoft Active Server Pages, Java Server Pages, and many others. This has led to many confusing variants of the HTML standard, causing difficulties for Web developers, tool vendors, and ultimately for end-users.

As the name implies, eXtensible Markup Language was designed from the beginning to allow extensions. If we go back to our example of the library, when they first indexed books, the Web did not exist. Probably the library catalog has no references to Web sites in it. Nowadays, many books have a companion Web site, and the librarian may wish to reference it. If XML were used to develop the catalog, then this could easily be accomplished. Importantly, with XML, old software is not disrupted by the addition of new information.

Semantic information

The final major benefit of XML is that it builds semantic information into the document. Semantic information (or meaning) is what allows readers to decide whether the book is about the color Brown, or written by Brown. An HTML-based Web search engine cannot do that, because it cannot distinguish between the title and author in the document — there isn’t enough semantic information in the HTML page. With XML, the document includes self-descriptive terms that show the meaning of the document.

Other benefits

The other main benefits of XML are that it is human-readable, tree-based, and easy to program. As time goes on, a large number of XML tools are emerging from both existing software vendors and XML startup companies.
It is human-readable, because it is based on text and simple tagging. This means that it can be understood by humans, and it can be written using nothing more than a text-editor or word-processor. This is important in the sense that programmers can interpret the data faster when writing new applications, but once they start running, no one reads the data (only the applications). The tree-based structure of XML is much more powerful than fixed-length data formats. Because objects are tree structures as well, XML is ideally suited to working with object-oriented programming. In particular, many people believe that there is an excellent affinity between Java and XML.

Finally, XML is easy to program, because there are already standards for *XML parsers. XML parsers are the subsystems that read the XML and allow programmers to work with XML. Because XML parsers are available to be re-used in new computer systems, many programmers are starting to use XML, even if they do not need any of the previously mentioned benefits.*

XML concepts

In this section we try to cover a few basic concepts of XML, however this is not intended to be an XML reference manual. Since most people are familiar with HTML, we often make comparisons between HTML and XML. While people’s familiarity with HTML will hopefully simplify their task in understanding XML, it is also important to emphasize the differences between the two languages.

Document validity and well-formedness

XML is a metalanguage, which means that it is a language for describing markup languages. This is done by defining a set of tags for each markup language. XML does not predefine any tags. It allows you to create your own tags. However the process of defining tags and creating documents using the tag is not arbitrary. Therefore, there are a few rules that XML tags and documents should adhere to, in order to ensure that they are usable by any XML application.

XML has tighter constraints as compared to HTML, which tolerate minor structural irregularities in the documents during parsing, such as unclosed tags. A well formed XML document should start with an XML declaration and should have a root element which contains all other elements. XML parsers will not accept documents that contain start tags, such as `<AUTHOR>`, without their corresponding end tags, in this example `</AUTHOR>`. This differs from HTML, which can be parsed even without any explicit end tags. On the other hand, XML does accept *empty* tags such as `<AUTHOR />`. Well-formedness constraints also deal with attribute names, which should be unique within an element, and attribute values, which must not contain the character “<“. A document is said to be *well-formed* when it conforms to these constraints, which are referred to as the well-formedness constraints (WFC) defined in the XML 1.0 Recommendation (refer to http://www.w3.org/XML/ for more information).

The notion of *validity* applies to XML documents which have a Document Type Definition (DTD) associated with them. A Document Type Definition specifies the structure of the XML document by providing with a list of elements, attributes, notations and entities that a document can contain. When an XML document has a DTD associated with it, a validating parser will read the DTD and checks whether the document adheres to the rules specified in the DTD. For example, for a document to be valid, all tags and attributes appearing in the document must have corresponding declarations in the DTD, and all elements appearing within other elements must respect the content model defined in the DTD.

It is worth noting that validity and well-formedness are two different aspects of an XML document. While well-formedness insures that XML parsers will be able to read the document, validity determines whether an XML document adheres to a DTD or schema. An XML application will check for and reject documents that are not well-formed before checking whether these documents comply with its validity constraints (VC). After a document is tested, validity checking can be turned off to improve performance.

Document type definition

In this section we briefly describe what a document type definition (DTD) is.

What is a DTD?

Since it is not the intent of this book to serve as an XML reference manual, we will not describe all the syntax elements of a DTD here. However, it is essential to understand the purpose and use of a DTD, and that is what we will focus on in this section. DTDs are extensively used by DB2 XML Extender, so we will look at them in a bit more detail.

The 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 which 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, it also constrains the element types that can be included in the document and determines its conformance: An XML document which conforms to its DTD is said to be valid.

A DTD can either be stored in a separate file or embedded within the same XML file. XML documents referencing a DTD will contain the `<!DOCTYPE>` declaration which either contains the entire DTD declaration, or specifies the location of an external DTD, as shown in the following example:

```
<!DOCTYPE LibraryCatalogue SYSTEM “library.dtd”>
```

An XML document is not required to have a DTD. However, with most applications, it will prove beneficial or even necessary to build a DTD which conveys efficiently the meaning behind the XML file’s contents. DTDs provide parsers with clear instructions on what to check for when they are determining the validity of an XML document.

Having the logical definition of an XML file stored separately allows for the resulting DTD to be shared across organizations, industries, or the Web. When building XML applications, it is probably a good idea to look for existing DTDs that might suit your purpose.

For more information on the latest emerging XML standards, the following sites may prove a good starting point:

```
http://www.w3.org
http://www.oasis-open.org
```

DTD example

The DTD has its own syntax, but is similar to XML in that it also uses markup tags. The following sample shows a simple *internal* DTD:

```
<?xml version = “1.0”?>
<!DOCTYPE authors [
<!ELEMENT authors(author)+>
<!ELEMENT author(firstname, lastname, title)>
<!ELEMENT firstname(#PCDATA)>
<!ELEMENT lastname(#PCDATA)>
<!ELEMENT title(#PCDATA)>
]>
...
[ insert XML data here]
...
```

In the above example, the **DOCTYPE** statement represents the *Document Type Declaration* and the statements included within the square brackets make up the *Document Type Definition*. Both terms share the same acronym (DTD), which can be confusing, but it is usually clear from the context which of the two meanings is being referred to.

A well-formed XML document must contain a *root element*. The `DOCTYPE` name specified in the declaration must match that root element, in this case `authors`:

```
<!DOCTYPE authors [
<!ELEMENT authors(author)+>
```

The second line constitutes an *element declaration*, and the “+” indicates that the `authors` element can contain one or more `author` elements, which in turn are declared like this:

```
<!ELEMENT author(firstname, lastname, title)>
```

Similarly, the `author` element contains several elements: `firstname`, `lastname`, `title`. However, only one instance of each is allowed in this case:

```
<!ELEMENT firstname(#PCDATA)>
<!ELEMENT lastname(#PCDATA)>
<!ELEMENT title(#PCDATA)>
```

These last three elements contain only text and are therefore defined as *parse character data* or *PCDATA*. The adjunction of the # character marks the PCDATA type as a reserved word, and it cannot be used for names created by the author.

As mentioned earlier in this chapter, the DTD can either be stored within the XML document which it validates, or in an external file. The following is an example of a Document Type Declaration specifying an *external* DTD:

```
<?xml version = “1.0”?>
<!DOCTYPE authors SYSTEM “authors.dtd”>
```

The use of the SYSTEM keyword indicates to the parser that this is an external declaration and that the set of rules for this XML document can be found in the specified file.

What’s in a DTD?

A Document Type Definition can contain different types of declarations. A list of these different types follows:

- ***Elements*** constitute the basic building blocks of an XML file and are declared like this:

  ```
  <!ELEMENT elementName(allowed element contents)>
  ```

  Example:

  ```
  <!ELEMENT greeting (#PCDATA)>
  <greeting> Hello, World! </greeting>
  ```

  Table B-1 on page 563 lists all the declaration attributes allowed inside an element declaration.

Table B-1 DTD symbols

Symbol	Meaning	Example	Description
, (comma)	Means “and” in specified order	TITLE, AUTHOR	TITLE and AUTHOR in that order
|	Means “or”	TITLE | AUTHOR	TITLE or AUTHOR
?	Means “optional”, but no more than one is allowed	ISBN?	ISBN does not have to be present, but if it is, there can be no more than one.
*	Means 0 or more Elements.	(TITLE | AUTHOR) *	Any number of TITLE or AUTHOR elements can be present
+	Means 1 or more Elements	AUTHOR+	At least one or more AUTHOR elements must be present
( )	Used to group elements	<!ELEMENT BOOK (AUTHOR | TITLE, YEAR-PUBLISHED, ISBN)>	An AUTHOR or a TITLE element must be present and must precede the YEAR-PUBLISHED and ISBN elements.

- ***Attributes***, as their name indicates, are attributes of an element which must be declared in the same DTD:

```
<!ATTLIST elementName attributeName attributeType attributeDefault>
```

Here is an example:

```
<!ELEMENT BOOK(#PCDATA)>
<!ATTLIST BOOK
    ID              ID              #REQUIRED
    TYPE    (Harcover | Paperback)  "Hardcover"
    STORELOC        CDATA           #FIXED    "5th Avenue"
    COMMENT         CDATA           #IMPLIED
```

Table B-2 on page 564 provides a description of the various attribute types which can be used in attribute declarations, and lists these attribute types.

Table B-2 Attribute types

Attribute type	Description
CDATA	Can contain any kind of character data.

ID	Must have unique values within the element. In the example below, TYPEID is of the ID type, and so requires unique values within the range of BOOK elements:
	`<BOOK TYPEID="ch1">See Spot Run</BOOK>`
	`<BOOK TYPEID="ch2">Jack and Jill</BOOK>`

IDREF	The value of an ID type attribute of an element in the document.
IDREFS	Multiple ID's of elements separated by whitespace.
(enumerated)	Attributes can have a specified list of acceptable values.
ENTITY	The name of an entity declared in the DTD.
ENTITIES	The attribute is optional.
NMTOKEN	The attribute is fixed (the syntax is of the type "#FIXED *Value*").
NOTATION	The name of a notation declared in the DTD.
NMTOKENS	Multiple XML names separated by whitespace.

Table B-3 lists all the default attribute values.

Table B-3 Default value for attributes

Attribute value	Description
#REQUIRED	The attribute is required.
#IMPLIED	The attribute is optional.
#FIXED	The attribute is fixed (the syntax is of the type "#FIXED *Value*").

- ***Entities*** are used to represent one or more characters in an XML document and act as constants, the value of which can be changed without the need to edit corresponding XML documents:

  ```
  <!ENTITY entityName "character string represented">
  ```

  Example:

  ```
  <!ENTITY prodname "ACME Calendar">

  (XML file:)
  Thank you for choosing &prodname; as your primary scheduling program
  ```

```
(rendered:)
Thank you for choosing ACME Calendar as your primary scheduling program
```

- ***Parameter entities*** are entities which are used within the DTD itself. Their declaration differs by the inclusion of the % character:

  ```
  <!ENTITY % entityName “character string represented”>
  ```

  Example:

  ```
  <!ENTITY % commonAtts
          "ID     ID      #REQUIRED
          MAKE    CDATS   #IMPLIED
          MODEL   CDATA #IMPLIED">

  <!ELEMENT CAR (#PCDATA)>
  <!ATTLIST CAR %commonAtts>

  <!ELEMENT COMPUTER (#PCDATA)>
  <!ATTLIST COMPUTER %commonAtts>
  ```

- ***Notations*** are used to refer to data from an outside (non-XML) source. They provide a basic means by which non-textual information can be handled within a document:

  ```
  <!NOTATION name ExternalID>
  ```

  Example:

  ```
  <!NOTATION jpeg SYSTEM "jpeg.exe">
  <!NOTATION gif SYSTEM "gif.exe">

  <!ELEMENT person (#PCDATA)>
  <!ATTLIST person
          picformat NOTATION (jpeg | gif) #REQUIRED>

  (XML file:)
  <person picformat=”jpeg”>Kelly Brown</person>
  ```

- ***Comments*** can be inserted inside the DTD by using the following notation:

  ```
  <!-- insert comment text here -->
  ```

  Example:

  ```
  <!--    XML Comments Example         -->
  ```

Namespaces

Namespaces are useful when there is a need for elements and attributes of the same name to take on a different meaning depending on the context in which they are used.

For instance, a tag called <TITLE> takes on a different meaning, depending on whether it is applied to a person or a book. If both entities (a person and a book) need to be defined in the same document, for example, in a library entry which associates a book with its author, we need some mechanism to distinguish between the two and apply the correct semantic description to the <TITLE> tag whenever it is used in the document. Namespaces provide a mechanism that allows us to write XML documents which contain information relevant to many software modules. Consider this example:

```
<?xml version=”1.0”?>
<library-entry xmlns:authr=”authors.dtd”
               xmlns:bk=”books.dtd”>
 <bk:book>
   <bk:title>XML Sample</bk:title>
   <bk:pages>210</bk:pages>
   <bk:isbn>1-868640-34-2</bk:isbn>
   <authr:author>
      <authr:firstname>Joe</authr:firstname>
      <authr:lastname>Bloggs</authr:lastname>
      <authr:title>Mr</authr:title>
   </authr:author>
 </bk:book>
</library-entry>
```

As we can see, the <TITLE> tag is used twice, but in a different context, once within the <AUTHOR> element and once within the <BOOK> element. Note the use of the xmlns keyword in the namespace declaration. Interestingly, the XML recommendation does not specify whether a namespace declaration should point to a valid URI (Uniform Resource Identifier), only that it should be unique and persistent.

In the previous example, in order to illustrate the relationship of each element to a given namespace, we chose to specify the relevant namespace prefix before each element. However, it is assumed that once a prefix is applied to an element name, it applies to all descendants of that element unless it is over-ridden by another prefix. The extent to which a namespace prefix applies to elements in a document is defined as the namespace *scope*. If we were to use scoping, the above example would then look like this:

```
<?xml version”1.0”?>
<library-entry xmlns:authr=”authors.dtd”
               xmlns:bk=”books.dtd”>
```

```
 <bk:book>
    <title>XML & WebSphere</title>
    <pages>210</pages>
    <isbn>1-868640-34-2</isbn>
    <authr:author>
       <firstname>Joe</firstname>
       <lastname>Bloggs</lastname>
       <title>Mr</title>
    </authr:author>
 </bk:book>
</library-entry>
```

In this example, it is clear that all elements within the `<BOOK>` element are associated with the **`bk`** namespace, except for the elements within the `<AUTHOR>` element which belong to the **`authr`** namespace.

DTD versus XML Schemas

The DTD provides a relatively easy-to-use way to describe the syntax and semantics of XML documents. However, to achieve this simplicity, a compromise was made when porting DTD support over from SGML to XML, which resulted in the expected simplification, but also in limitations that prevented the DTD from performing a high degree of semantic checking.

For example, a DTD allows for limited conditional checking by specifying allowed values, but there is no support for more complex semantic rules. For instance, it is impossible to check that an element which *should* contain a date actually *does* contain a date. There are also limitations when it comes to defining complex relationships between data elements and their usage, especially when XML documents also use namespaces which might define elements conflicting with DTD declarations.

Therefore, there is a need for a way to specify more complex semantic rules and provide type-checking within an XML document. XML Schemas, aim to provide such functionality and also introduce new semantic capabilities such as support for namespaces and type-checking.

For more information on XML Schemas, refer to the specification documents from the W3C:

XML Schema Part 1: Structures

`http://www.w3.org/TR/xmlschema-1/`

XML Schema Part 2: Datatypes

`http://www.w3.org/TR/xmlschema-2/`

XPath

The name *XPath* comes from its use as notations in URIs for navigating through XML documents. The aim of an XPath is to address parts of an XML document. XPath uses a compact syntax, and it operates on the logical structure underlying XML to facilitate usage of XPath within URIs and XML attribute values. Xpath supports XML namespaces because XPath models an XML document as a tree of nodes (root nodes, element nodes, attribute nodes, text nodes, namespace nodes, processing instruction nodes, and comment nodes). The basic syntactic construct in XPath is the *expression*. An object is obtained by evaluating an expression, which has one of the following four basic types:

- Node-set (an unordered collection of nodes without duplicates)
- Boolean
- Number
- String

XPath uses path notation to define locations within a document. A path starting with a “/” signifies an absolute path. A simple example of this follows.

Let us consider an XML document (Library.xml) that describes a library system. This document will be used for XPath examples.

```
<? xml version=”1.0”?>
<!DOCTYPE LIBRARY SYSTEM “library.dtd”>
<LIBRARY>
  <BOOK ID=”B1.1”>
    <TITLE>xml</TITLE>
    <COPIES>5</COPIES>
  </BOOK>
  <BOOK ID=”B2.1”>
    <TITLE>WebSphere</TITLE>
    <COPIES>10</COPIES>
  </BOOK>
  <BOOK ID=”B3.2”>
    <TITLE>great novel</TITLE>
    <COPIES>10</COPIES>
  </BOOK>
  <BOOK ID=”B5.5”>
    <TITLE>good story</TITLE>
    <COPIES>10</COPIES>
  </BOOK>
</LIBRARY>
```

The path /child::book/child::copies selects all **copies** element children of **book** which are defined under the document's root. The above path can also be written as **/book/copies**.

The XPath location step makes the selection of a document part based on the basis and the predicate. The basis performs a selection based on Axis Name and Node Test. Then the predicate performs additional selections based on the outcome of the selection from the basis. A simple example of this is as follows:

The path /child::book[position()-1] selects the first **book** element under root. This location step can also be written as **/book[1]** if you wish.

For example, the path /book/author/@* would have selected all the **author** elements' attributes.

The path /book/author[@type='old'] would have selected all the **author** elements with type attribute equal to **"old"**.

eXtensible Stylesheet Language (XSL)

XSL is the language defined by the W3C to add formatting information to XML data. Stylesheets allow data to be formatted based on the structure of the document, so one stylesheet can be used with many similar XML documents.

XSL is based on two existing standards, *Cascading Style Sheets* (CSS) and *Document Style Semantics and Specification Language* (DSSSL).

CSS is the stylesheet language for HTML 4.0, and as such, is well supported in Web design tools, such as IBM WebSphere Studio. XSL is mainly based on CSS, and so a short description of CSS is provided below. CSS can also be used as a formatting language for XML, but it is less powerful than XSL. Because CSS was designed for the Web, it is excellent for defining the presentation of data for Web browsers.

XML aims to support any possible display, and printed output has a number of challenges that browsers do not face. DSSSL is the stylesheet language of SGML and has mainly been used for printed output. Therefore, elements of DSSSL that go beyond CSS have been incorporated into XSL. More information on DSSSL can be found at the following Web address:

http://www.jclark.com/dsssl/

Cascading Stylesheets

Cascading Stylesheets were designed to help separate presentation from data with HTML. The reason that they are called *Cascading Stylesheets* is because HTML, like XML, has a tree structure, and styles which are applied to the root of a tree cascade down to the branches and leaves. CSS allows the Web developer to define styles that apply:

- To any given type of element (for example, all paragraphs)
- To a class of elements (for example, all paragraphs which contain code samples)
- To a particular element (for example, the third paragraph)

This is achieved by specifying *class*es and *id*s in the HTML, and applying styles to them.

A very simple stylesheet is presented in Example B-1 on page 571. This stylesheet defines a standard font and colors for all text in the BODY of the HTML file. It defines a specific class of text which is twice the normal size, bold and capitalized (**`.largeClass`**), and finally it specifies that a particular element labelled **`THISID`** should be displayed in fuschia-colored cursive text.

The benefits of CSS are well-understood: Web developers can easily change the layout and presentation of a whole site by editing a single stylesheet.
CSS can be used with XML if the display engine supports it, for example Microsoft Internet Explorer.

Example: B-1 A simple CSS stylesheet

```
BODY{
   font-family : Verdana,sans-serif;
   font-weight : normal;
   color : black;
   background-color : white;
   text-decoration : none;
}
.largeClass{
   font-size : 200%;
   font-weight : bolder;
   text-transform : capitalize;
}
#THISID{
   font-family : cursive;
   color : fuchsia;
}
```

CSS can be used within a document, or referenced in a separate stylesheet, which is the more common approach. For more information on CSS, see:

`http://www.w3.org/Style/CSS`

XSL = fo: + XSLT

Although XSL has derived much from CSS, the approach of XSL is much more powerful, and has major differences from CSS. XSL is W3C Recommendation. More information can be found at the following URL:

`http://www.w3.org/TR/xsl/`

XSL actually consists of two different standards, the *transformation language*, and the *formatting objects*.

The transformation language is called XSLT, and is defined as a W3C Recommendation at the following URL:

`http://www.w3.org/TR/xslt/`

XSLT defines a common language for transforming one XML document into another. It defines how to create a *result tree* from a *source tree*.

The formatting objects (FO) define how to display the result tree. This is the part of XSL which is most closely related to CSS. Formatting objects are referred to as `fo:` in the XSL code. The main difference between CSS and XSL FO is that XSL is based on an XML format, properly defined with a DTD, and CSS is not.

Because XSL defines an extra step in presenting data, it can do much more powerful presentation tasks than CSS can. CSS always retains the order of the source tree, whereas XSL can re-order the data. A simple example might be where two stylesheets can be used to display the same data, one ordered by name, and the other ordered by location.

XSL transformations

XSLT works on two principles: *pattern matching* and *templates*. Remember that XSLT transforms a source tree into a result tree. The XSLT processor takes two inputs; the XSL stylesheet and the source tree; and produces one output — the result tree. The stylesheet associate patterns with templates. The patterns identify parts of the source tree. When a match is made, the associated template is applied.

XSL stylesheets can also *filter* the input data, based on logical tests, and *re-order* the data. The templates can also contain arbitrary tags, formatting instructions or data. The result of this is that an XSL stylesheet can perform very powerful transformations on the input data. For example, a stylesheet can be used to create an HTML display of a list of bank accounts, sorted by balance, with overdrawn accounts colored red, and large balances colored green. The same data can be used with another stylesheet which graphically represented the data by transforming it into *Structured Vector Graphics* (SVG), which is an XML format for drawing graphics.

Processing XML using Java

In a perfect world, computer applications would just exchange XML documents. In real life, applications often have to be able to support multiple client types, all with different capabilities. The dominant client type for Web application servers is currently a browser (usually Netscape or Internet Explorer), but it will not be like that forever. We might have cellular phones and other front-end devices, all with different XML capabilities.

We also do not want to send the same XML document to every client, because some users of the application might be authorized to see more data than others. We must have the ability to process XML documents and generate the kind of response to the client that is adequate for the client type.

On the server side, the Web application server usually connects to a back-end data store like a relational database that does not natively support data interchange using XML. We need to be able to extract the necessary information from an XML document and pass that information to the database, as well as transform the information coming from the database to XML. To fulfill both the client and the server requirements, we need an XML processor.

While the XML document format is the most natural form of data exchange in the Internet, Java is the most natural language to be used in Internet applications and application servers. This is because of Java’s nature: object-oriented and distributed.

One technical advantage of Java over other languages is its built-in support for Unicode. With other languages, XML processing has to be done using tricks or by developing additional libraries to support Unicode in that language environment. IBM does have a C++ implementation of an XML parser, as well as supporting Unicode libraries, but, to summarize, Java is an excellent language in implementing XML processors and other XML related tools and applications.

XML applications

At the heart of every XML application is an XML processor that parses the well-formed XML document, so that the document elements can be retrieved and transformed into data that can be understood by the application and task in hand. The other responsibility of the parser is to check the syntax and structure (validity and well-formedness) of the document.

Anyone has the freedom to implement a parser that can read and print an XML document. The XML 1.0 Recommendation defines how an XML processor should behave when reading and printing a document, but the API to be used is not defined. However, there are standards that define how XML documents should be accessed and manipulated. Currently, the following two APIs used are widely used:

- Simple API for XML
- Document Object Model

SAX

Simple API for XML (SAX) was developed by David Megginson and a number of people on the xml-dev mailing list on the Web, because a need was recognized for simple, common way of processing XML documents. As such, SAX 1.0 is not a W3C recommendation, but it is the de-facto standard for interfacing with an XML parser, with many commonly available Java parsers supporting it.

SAX is an event-driven lightweight API for accessing XML documents and extracting information from them. It cannot be used to manipulate the internal

structures of XML documents. As the document is parsed, the application using SAX receives information about the various parsing events. The logical structure of an application using SAX API with the parser is shown in
Figure B-1.

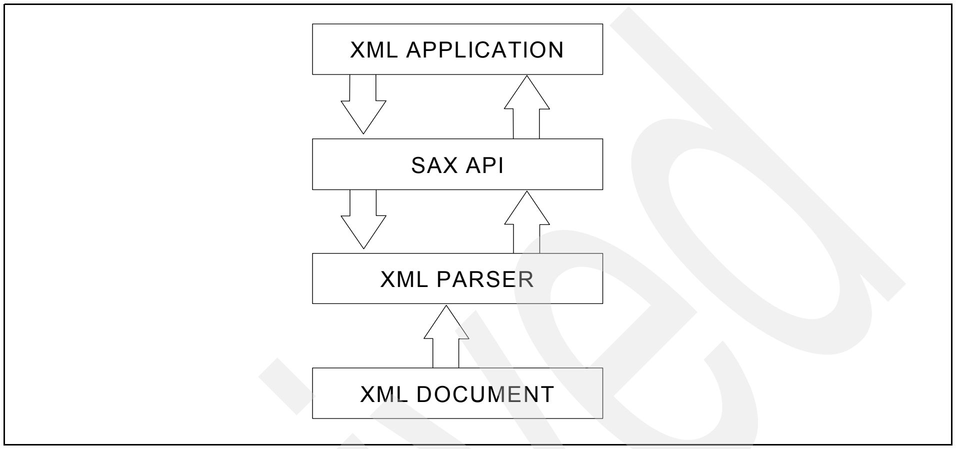


Figure B-1 SAX application components

The SAX driver can be implemented by the XML parser vendor, or as an add-on to the parser. That makes the application using the parser via SAX independent of the parser.

SAX classes and interfaces

The SAX 1.0 API defines two sets of interfaces, one of which is meant to be implemented by XML parsers, and one by XML applications. The interfaces that parsers have to implement are:

- Parser
- AttributeList
- Locator (optional)

The first thing an XML application has to do is to register SAX event handlers to a parser object that implements the **Parser** interface. As the XML document is processed by a parser, SAX notifies the application whenever an event occurs. The events that can be captured depend on the registered event handlers, the interfaces of which are:

- DocumentHandler
- DTDHandler
- ErrorHandler

The most important and commonly used interface is DocumentHandler, because it can be used to track basic document-related events like the start and end of elements and character data. The events occur in the order that is directly related to the order of elements that are found in the tree-formed XML document that is being parsed.

DTDHandler notifies the application about unparsed external entity declarations or when a notation declaration is encountered. ErrorHandler notifies the application whenever an error occurs while parsing the XML document.

The SAX specification also provides a HandlerBase class, which implements all interfaces and provides default behavior. Instead of implementing the appropriate interfaces, an XML application can extend the HandlerBase class and override just the methods that need to be customized.

The Java implementation of SAX is organized in two Java packages:

- org.xml.sax
- org.xml.sax.helpers

The first of the above-mentioned packages contains the SAX core implementation classes, interfaces and exceptions. The second one contains convenience classes and a Java-specific class (**ParserFactory**) for dynamically loading SAX parsers.

The implementation can be downloaded from `http://www.megginson.com/SAX/`. The same location also contains full descriptions (in JavaDoc format) of all classes and interfaces defined in SAX 1.0.

SAX example

For a Java application to be able to use SAX, we need a class that implements an interface most suitable for the job. The following code fragment shows the relevant methods of DocumentHandler that are implemented to track start and end of elements and the whole document. It also prints out the actual data within the elements (Example B-2):

Example: B-2 Sax example

```
public class MyDocHandler implements org.xml.sax.DocumentHandler
...
public void characters(char[] arg1, int start, int length) throws
org.xml.sax.SAXException {
   System.out.println(new String(arg1, arg2, arg3));
}
public void startDocument() throws org.xml.sax.SAXException {
   System.out.println("Start of document");
```

```
}
public void endDocument() throws org.xml.sax.SAXException {
   System.out.println("End of document");
}
public void startElement(String name, org.xml.sax.AttributeList arg2)
throws org.xml.sax.SAXException {
   System.out.println("Start of element " + name);
}public void endElement(String name) throws org.xml.sax.SAXException {
   System.out.println("End of element " + name);
```

The application that uses the DocumentHandler implementation above is simple. IBM’s XML for Java implements the Parser interface in SAXParser class, which the following example uses:

```
...
Parser parser = ParserFactory.makeParser("com.ibm.xml.parsers.SAXParser");
SampleDocumentHandler hndlr = new SampleDocumentHandler();
parser.setDocumentHandler(hndlr);
parser.parse(anXMLFileURL);
...
```

Given the following XML document as input:

```
<?xml version="1.0"?>
   <personnel>
      <person id="jedi1">
      <name>
         <lastname>Skywalker</lastname>
         <firstname>Luke</firstname>
      </name>
   </person>
</personnel>
```

The output of the SAX application looks like this:

```
Start of document
Start of element personnel
Start of element person
Start of element name
Start of element lastname
Skywalker
End of element lastname
Start of element firstname
Luke
End of element firstname
End of element name
End of element person
End of element personnel
End of document
```

DOM

While XML is a language to describe tree-structured data, the Document Object Model (DOM) defines a set of interfaces to access tree-structured XML documents. DOM specifies how XML and HTML documents can be represented as objects. Unlike SAX, DOM also allows creating and manipulating the contents of XML documents. Basically, the DOM interfaces are platform and language neutral.

DOM originated from the need to dynamically render HTML content (DHTML). The current DOM Level 1 Recommendation has two parts: Core and HTML. Core contains fourteen interfaces, seven of which are applicable to both HTML and XML documents. Six remaining interfaces are specific to XML. DOM HTML defines additional convenience methods that are useful for client side scripting.

DOM hierarchy

The DOM API is a set of interfaces that must be implemented by a DOM implementation such as IBM's XML for Java. The interfaces, being originally described in IDL, form a hierarchy (see Figure B-2 on page 578).

The root of the inheritance tree is Node, that defines the necessary methods to navigate and manipulate the tree-structure of XML documents. The methods include getting, deleting, and modifying the children of a node, as well as inserting new children to it. Document represents the whole documents, and the interface defines methods for creating elements, attributes, comments, and so on. Attributes of a Node are manipulated using the methods of the Element interface. DocumentFragment allows extracting parts of a document.

Note that while a DOM application reads an XML document and an object representation if formed, that representation remains only in memory. Changing a DOM object in memory does not automatically modify the original file. That is something an application program has to do for itself.

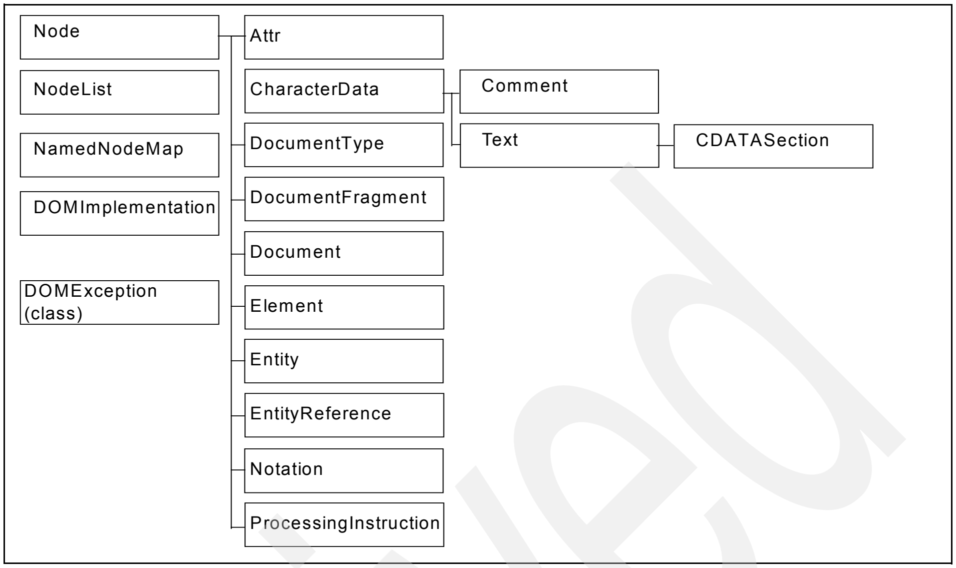


Figure B-2 DOM interface hierarchy

The W3C DOM Level 1 Recommendation can be found at:

http://www.w3.org/DOM/

DOM example

When the simple XML document we used in our SAX example (see listing on page 576) is processed using DOM, the resulting object tree will look like the one in Figure B-3. The shaded rectangles represent character data, and the others represent elements.

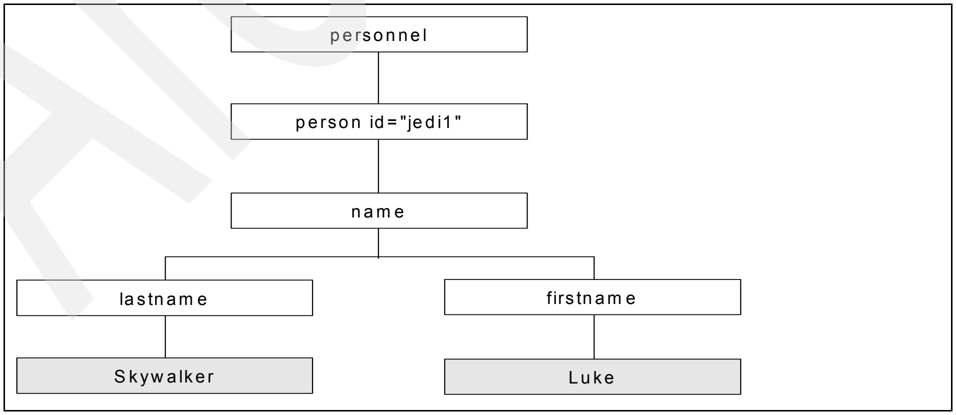


Figure B-3 Sample DOM tree

Reading an XML document using DOM is relatively easy, provided that a good parser is available. Among other things, IBM's XML Parser for Java provides a robust and very complete implementation of the W3C DOM API. The following code fragment shows a simplified example of how to read and manipulate an XML document using the DOMParser class:

```
DOMParser parser = new DOMParser();
parser.parse(uri);
Document document = parser.getDocument();
print(document); // implemented in our own code
Node n = document.getLastChild().getFirstChild();
n.setNodeValue("ZAP! You're history!");
print(document);
```

SAX or DOM?

There are certainly applications that could use either SAX or DOM to get the necessary functionality needed when processing XML documents. However, these two approaches to XML processing each have their strengths and weaknesses.

SAX advantages and disadvantages

SAX provides a standardized and commonly used interface to XML parsers. It is ideal for processing large documents whose content and structure does not need to be changed. Because the parser only tells about the events that the application is interested in, the application is typically small, and has a small memory footprint. This also means that SAX is fast and efficient, and a good choice for application areas such as filtering and searching, where only certain elements are extracted from a possibly very large document.

Because the events must be handled as they occur, it is impossible for a SAX application, for example, to traverse backwards in the document that is under processing. It is also beyond SAX's capabilities to create or modify the contents and internal structure of an XML document.

DOM advantages and disadvantages

Because every element of an XML document is represented as a DOM object to the application using the DOM API, it is possible to make modifications to the original XML document. Deleting a DOM node means deleting the corresponding XML element and so on. This makes DOM a good choice for XML applications that want to manipulate XML documents, or to create new ones.

DOM is not originally an event driven API like SAX, even though DOM Level 2 specifies events. To extract even a small piece of data from an XML document, the whole DOM tree has to be created. There is no way of creating lightweight applications using DOM. If the original XML document is large, the DOM application that manipulates the document requires a lot of memory. In practice, DOM is mostly used only when creating or manipulating XML documents is a requirement.

C

Table-based and object-relational mappings

When using an XML-enabled database, it is necessary to map the database schema to the XML schema (or vice versa). There are a number of different types of mappings. This chapter describes two of the most common ones:

- The table-based mapping
- The object-relational mapping

Common to both of these mappings (and not discussed in the following sections) is that the mappings allow both names and data types to be changed during the mapping. That is, names in the XML schema are not required to match names in the database schema. Similarly, data types in the XML schema are not required to match data types in the database schema, although it must be possible to convert between the data types in each schema and the values in instance documents must be convertible to the data type in the database schema.

Table-based mapping

Table-based mapping requires the XML document to have a structure corresponding to a single table or a set of tables. For example, an XML document corresponding to a sales order table may look like Example C-1.

Example: C-1 Sales order XML document

```
<SalesOrders>
   <SalesOrder>
      <Number>123</Number>
      <OrderDate>2003-7-28</OrderDate>
      <CustomerNumber>456</CustomerNumber>
   </SalesOrder>
   ...
</SalesOrders>
```

And an XML document corresponding to a sales order table and the related rows from a line item table might look like Example C-2.

Example: C-2 Sales order document with line items

```
<Database>
   <SalesOrders>
      <SalesOrder>
         <Number>123</Number>
         <OrderDate>2003-7-28</OrderDate>
         <CustomerNumber>456</CustomerNumber>
      </SalesOrder>
      ...
   </SalesOrders>
   <Items>
      <Item>
         <SONumber>123</SONumber>
         <Number>1</Number>
         <PartNumber>XY-47</PartNumber>
         <Quantity>14</Quantity>
         <Price>16.80</Price>
      </Item>
      <Item>
         <SONumber>123</SONumber>
         <Number>2</Number>
         <PartNumber>B-987</PartNumber>
         <Quantity>6</Quantity>
         <Price>2.34</Price>
      </Item>
      ...
   </Items>
</Database>
```

There are three important things to notice about the second document.

- First, the data from each table is listed separately, rather than having the line item data nested inside the corresponding sales order data as might be expected in an XML document. This is the major limitation of the table-based mapping.
- Second, the sales order number, which is used to link the two tables, is listed twice; once in the data for the sales order table, and once in the data for the line item table. This is because the sales order number appears in both tables in the database.
- Third, the document has a wrapper element around all the other elements. This element does not correspond to any structure in the database, but is needed because XML requires a single root element.

The table-based mapping can use child elements, attributes, or a mixture of the two to represent data. For example, both of the documents in Example C-3 represent the same data as the first document, even though these are technically different XML documents.

Example: C-3 Attributes - elements mixture

```
<SalesOrders>
   <SalesOrder Number="123" OrderDate="2003-7-28" CustomerNumber="456" />
   ...
</SalesOrders>


<SalesOrders>
   <SalesOrder Number="123">
      <OrderDate>2003-7-28</OrderDate>
      <CustomerNumber>456</CustomerNumber>
   </SalesOrder>
   ...
</SalesOrders>
```

Although the preceding examples show how rows in the database can be represented in an XML document, the mapping itself is actually done at the schema level. That is, the database schema is mapped to an XML schema. For example, the sales order and line item tables might be mapped to a DTD (Example C-4).

Example: C-4 Tables and schemas

```
Tables:
Orders (Number, OrderDate, CustomerNumber)
Items  (SONumber, Number, PartNumber, Quantity, Price)

Schema:
   <!ELEMENT Database (SalesOrders, Items)>

   <!ELEMENT SalesOrders (SalesOrder*)>
   <!ELEMENT SalesOrder (Number, OrderDate, CustomerNumber)>
   <!ELEMENT Number (#PCDATA)>
   <!ELEMENT OrderDate (#PCDATA)>
   <!ELEMENT CustomerNumber (#PCDATA)>

   <!ELEMENT Items (Item*)>
   <!ELEMENT Item (SONumber, Number, PartNumber, Quantity, Price)>
   <!ELEMENT PartNumber (#PCDATA)>
   <!ELEMENT Quantity (#PCDATA)>
   <!ELEMENT Price (#PCDATA)>
```

The table-based mapping is important because of its simplicity. It is very easy to write the software that transfers data between XML documents and the database according to this mapping. Furthermore, the table-like structure of the XML document means that inserts can be performed efficiently; documents can be processed linearly (limiting memory usage), and the data can be inserted with bulk inserts (improving performance).

The table-based mapping is most commonly used to serialize result sets, update existing data, and transfer large amounts of relational data, such as during database replication. It is also the basis for implementing XQuery over a relational database, with each table viewed as a single XML document.

In spite of the fact that the table-based mapping requires data from different tables to be listed separately, it can be used with more deeply nested documents, as shown in Example C-5.

Example: C-5 Using nesting with table-based mapping

```
<SalesOrder Number="123">
   <OrderDate>2003-7-28</OrderDate>
   <CustomerNumber>456</CustomerNumber>
   <Item Number="1">
      <PartNumber>XY-47</PartNumber>
      <Quantity>14</Quantity>
      <Price>16.80</Price>
   </Item>
   <Item Number="2">
      <PartNumber>B-987</PartNumber>
      <Quantity>6</Quantity>
      <Price>2.34</Price>
   </Item>
</SalesOrder>
```

To do this, the application uses XSLT to transform this document into the table-based document and vice versa, as is shown in Figure C-1. Thus, when retrieving data from the database, a table-based document is constructed first, and then XSLT is used to create a more deeply nested document. (This transformation requires keys to determine how the data should be nested.) Similarly, when inserting data into the database, XSLT can be used to transform a deeply-nested document into a table-based document, and the data then transferred according to a table-based mapping.

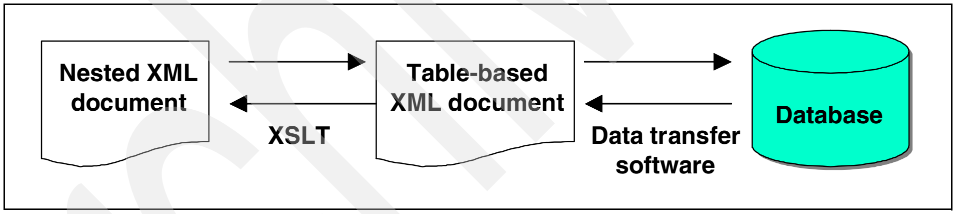


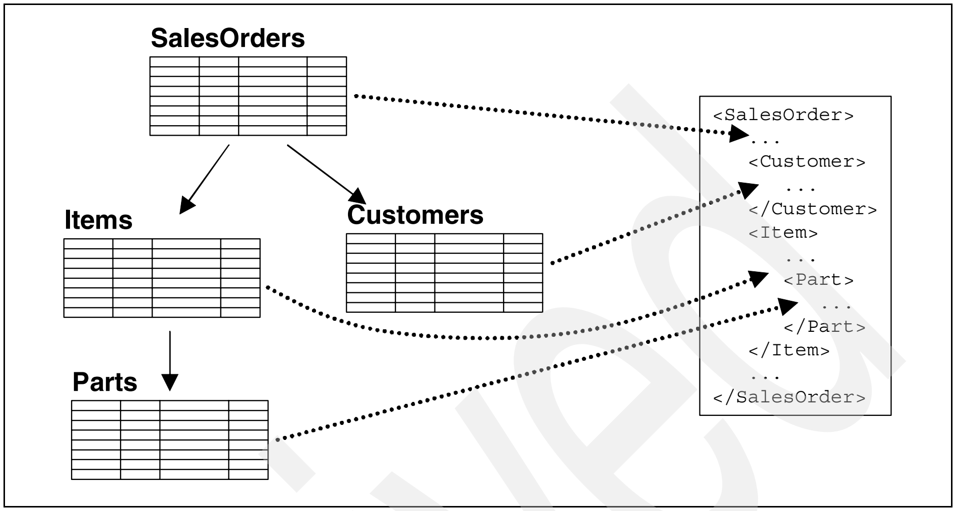
Figure C-1 Transforming a nested document into a table-based document

For more information, see Chapter 7, “Bulk processing of XML documents” on page 191.

Object-relational mapping

Unlike the table-based mapping, the object-relational mapping handles deeply nested XML documents directly. The object-relational mapping can be viewed in one of two ways.

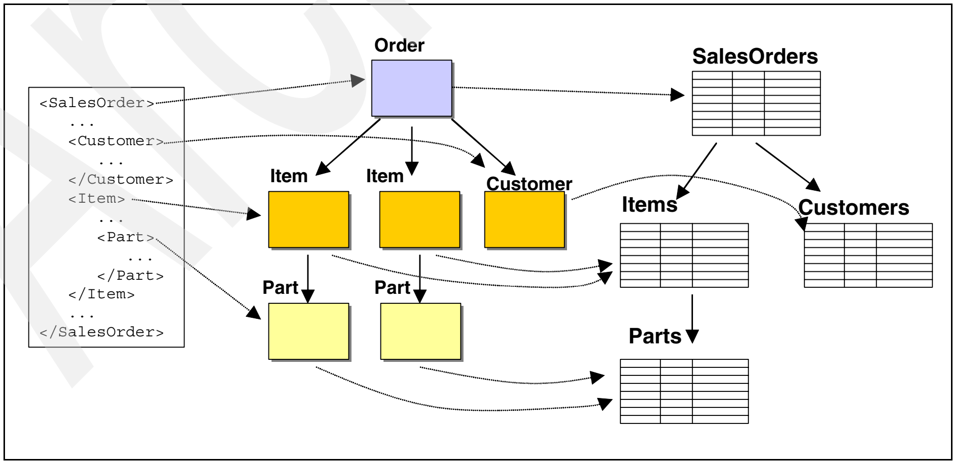
From a *database perspective*, individual rows are mapped with a table-based mapping, but primary key/foreign key relationships in the database determine how the rows will be nested in the XML document. Thus, a tree of tables in the database becomes a tree of elements in the XML documents shown in Figure C-2.



```
<SalesOrder>
   ...
   <Customer>
      ...
   </Customer>
   <Item>
      ...
      <Part>
         ...
      </Part>
   </Item>
   ...
</SalesOrder>
```

Figure C-2 Mapping a tree of tables to a nested XML document

From an *XML perspective*, an XML document is viewed as a serialized tree of objects with nesting indicating the relationships between objects and properties in the obvious way. The objects are then mapped to the database using traditional object-relational mapping techniques. That is, objects are mapped to tables, properties are mapped to columns, and inter-object relationships are mapped to primary key/foreign key relationships, as sown in Figure C-3.



```
<SalesOrder>
   ...
   <Customer>
      ...
   </Customer>
   <Item>
      ...
      <Part>
         ...
      </Part>
   </Item>
   ...
</SalesOrder>
```

Figure C-3 Mapping an XML document to objects, then to tables

Although the preceding diagram shows the result of mapping an XML document to an object tree, and then to rows in a table, the mapping itself is actually done at the schema level. It is also important to note that the objects used in this mapping are specific to each XML schema and are not DOM objects. In other words, they model the data found in the document, not the document itself. To see the difference, Figure C-4 shows the tree of objects used in the preceding example.

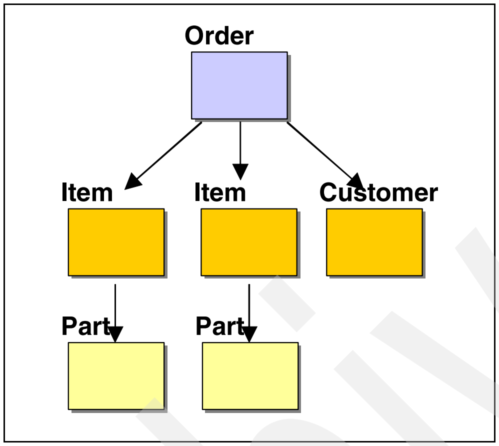


Figure C-4 Tree of sales order objects

Figure C-5 on page 588 shows a DOM tree that models the same document.

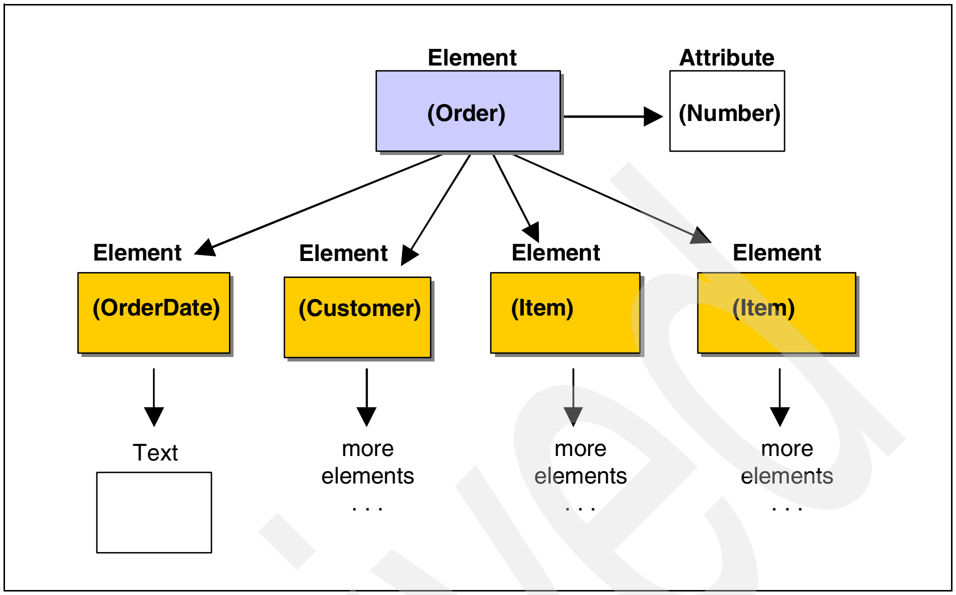


Figure C-5 DOM tree for the sales order document

Furthermore, the objects are only used only to visualize the mapping. That is, they are *not* instantiated when data is transferred between the XML document and the database.

In the following sections, we briefly describe how to map XML schemas to relational schemas, using DTD notation for simplicity. Mapping relational schemas to XML schemas is just the reverse of this process, and is somewhat simpler. For a more complete description, see "Mapping DTDs to Databases" on XML.com:

http://www.xml.com/pub/a/2001/05/09/dtdtodbs.html

Simple and complex types

Before we describe the actual mapping, we need to define the terms *simple type* and *complex type*. Although these come from XML Schemas, they are equally useful in describing types found in DTDs.

- A *simple type* is a scalar type, such as a string or an integer.

  In DTDs, element types support a single scalar type: PCDATA (a string). Attributes can have a number of different scalar types, including CDATA, ID, IDREF, NMTOKEN, and enumerated types. However, since databases do not directly support these types, they are most easily thought of as strings.

XML Schemas support a larger variety of simple types, including strings, integers, floats, dates, and times. In addition, users can define their own simple types. The simple types in XML Schemas can be used to assign types to attributes and to element types that contain only PCDATA.

Note: When using simple types in XML Schemas, it is important to remember that the data in an XML document is always a string; the simple type specifies the format of a particular data value.

- A *complex type* is a structure. This structure may be expressed using attributes and/or a content model —that is, a set of child elements.

DTDs do not directly support the concept of a complex type as something separate from an element type. That is, if you want two different element types to have the same content, you must repeat the content model and/or attributes. For example:

```
<!ELEMENT ShipToAddress (Street, City, State, PostCode, Country)>
<!ELEMENT BillToAddress (Street, City, State, PostCode, Country)>
```

Parameter entities appear to be similar to complex types. For example:

```
<!ENTITY % Address "Street, City, State, PostCode, Country">
<!ELEMENT ShipToAddress (%Address;)>
<!ELEMENT BillToAddress (%Address;)>
```

However, there is one important difference: parameter entities are a physical construct used for string substitution, while complex types are a logical construct like an element type.

In XML Schemas, you can define complex types separately and declare that an element type uses a complex type, as shown in Example C-6.

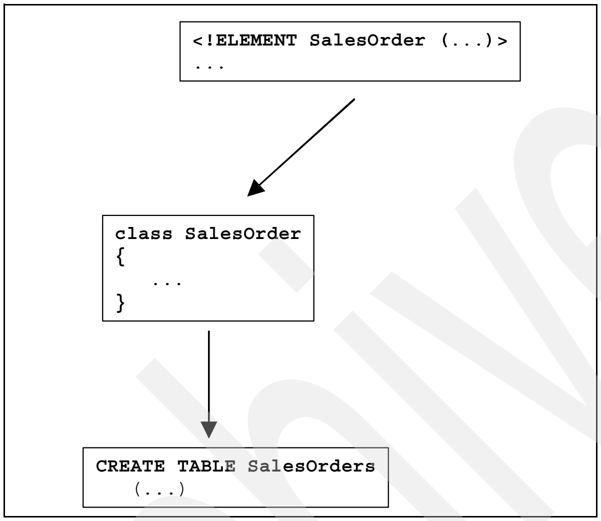
Example: C-6 Using complex types in XML Schemas

```
<xsd:complexType name="Address">
   <xsd:sequence>
      <xsd:element name="Street"   type="xsd:string"/>
      <xsd:element name="City"     type="xsd:string"/>
      <xsd:element name="State"    type="xsd:string"/>
      <xsd:element name="PostCode" type="xsd:string"/>
      <xsd:element name="Country"  type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:element name="ShipToAddress" type="Address">
<xsd:element name="BillToAddress" type="Address">
```

For the purposes of mapping XML schemas to database schemas, it is only important to understand that simple types are scalar types and complex types are structures.

Mapping complex element types

Element types that have a complex type are known as *complex element types*. These are mapped to classes, then to tables (Figure C-6).



```
<!ELEMENT SalesOrder (...)>
...
```

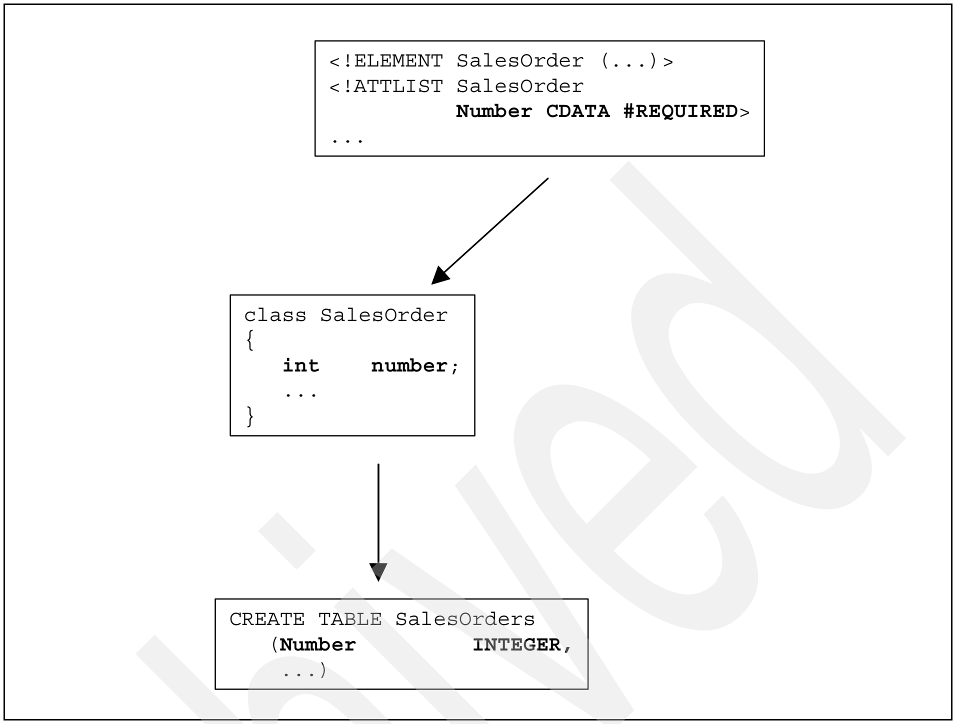
```
class SalesOrder
{
   ...
}
```

```
CREATE TABLE SalesOrders
   (...)
```

Figure C-6 Mapping complex element types to classes, then to tables

Mapping attributes

Attributes are mapped to properties in the class of their parent element type, then to columns in the parent's table (Figure C-7 on page 591).



```
<!ELEMENT SalesOrder (...)>
<!ATTLIST SalesOrder
          Number CDATA #REQUIRED>
...
```

```
class SalesOrder
{
   int    number;
   ...
}
```

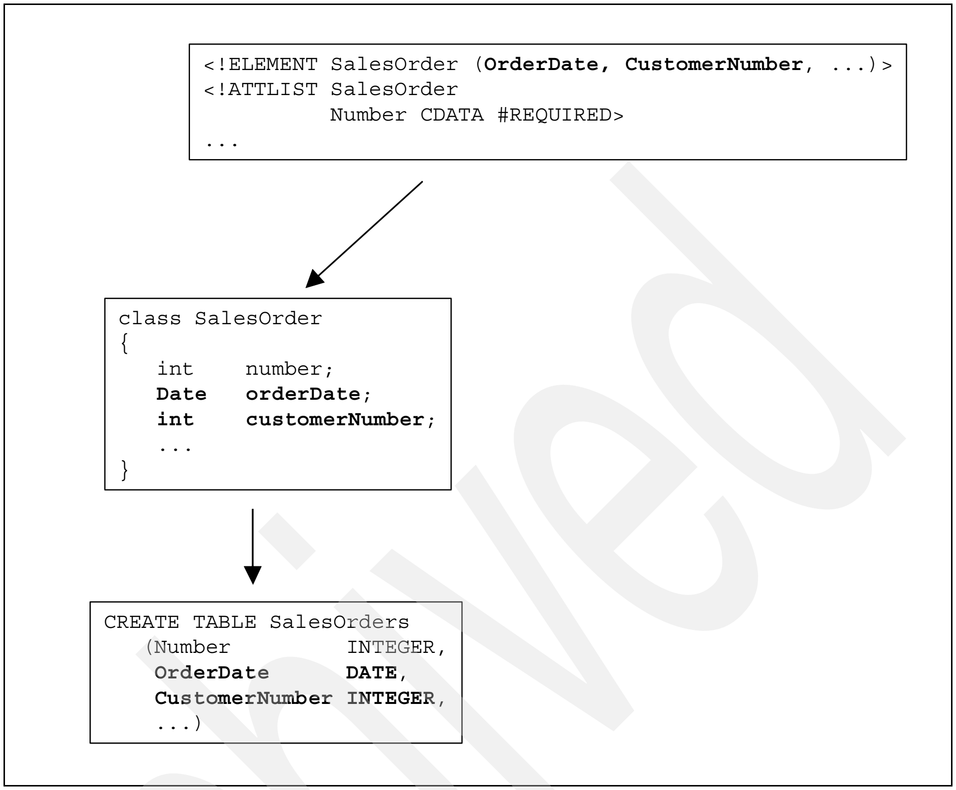
```
CREATE TABLE SalesOrders
   (Number         INTEGER,
    ...)
```

Figure C-7   Mapping attributes to properties, then to columns

The data type of the attribute must be convertible to the data type of the column. For example, it is an error to map an attribute with a “date” data type to a column with a “boolean” data type.

Mapping references to simple element types

Like attributes, references in the content model to simple element types are mapped to properties in the class of their parent element type, then to columns in the parent's table (Figure C-8 on page 592).



```
<!ELEMENT SalesOrder (OrderDate, CustomerNumber, ...)>
<!ATTLIST SalesOrder
          Number CDATA #REQUIRED>
...
```

```
class SalesOrder
{
   int    number;
   Date   orderDate;
   int    customerNumber;
   ...
}
```

```
CREATE TABLE SalesOrders
   (Number         INTEGER,
    OrderDate      DATE,
    CustomerNumber INTEGER,
    ...)
```

Figure C-8 Mapping references to simple element types to properties to columns

As with attributes, the data type of the element type must be convertible to the data type of the column.

Note that we say we are mapping a *reference* to an element type, rather than mapping the element type itself. The distinction is important when the same element type occurs in more than one content model. For example, suppose we have an XML schema that describes books and that both the Chapter and the Appendix element types refer to the Title element type in their content models:

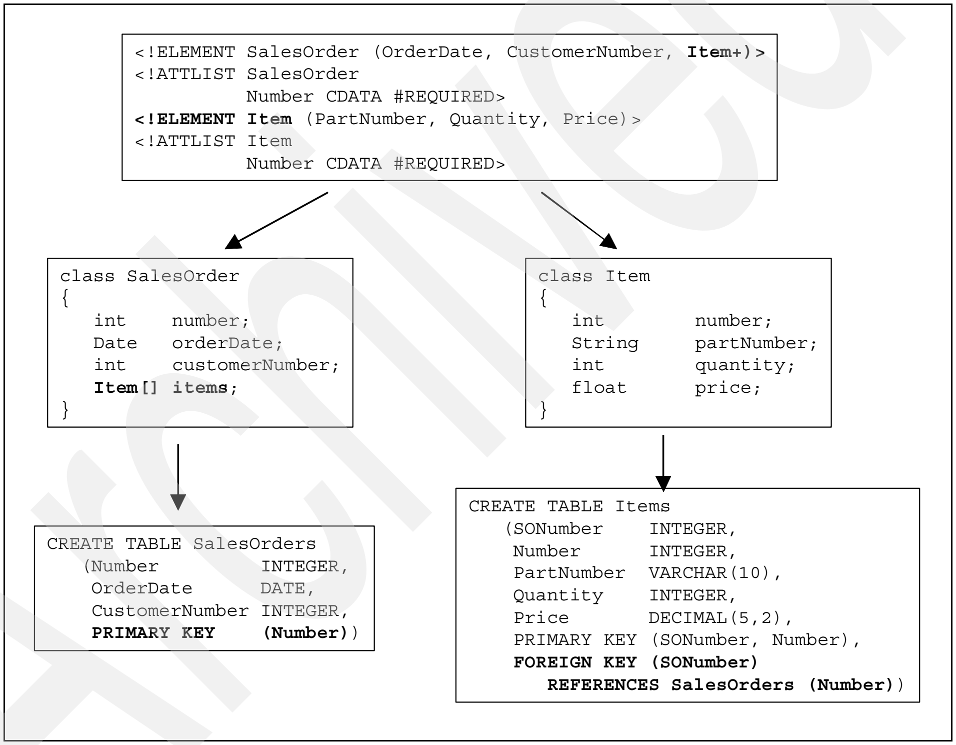
```
<!ELEMENT Chapter (Number, Title, Section+)>
<!ELEMENT Appendix (Letter, Title, Section+)>
```

Each of these references must be mapped separately. That is, even though both content models refer to the Title element type, each reference results in a different column in the database. When the Title element type appears in the Chapter element type, it is mapped to a column in the Chapters table, and when it appears in the Appendix element type, it is mapped to a column in the Appendixes table:

```
CREATE TABLE Chapters  (Number INTEGER, Title VARCHAR(50))
CREATE TABLE Appendixes (Letter CHAR(1), Title VARCHAR(50))
```

Mapping references to complex element types

Like references to simple element types, references to complex element types are mapped to properties in the class of their parent element type. However, there are two differences. First, the data type of the property is the class to which the referenced element type is mapped. Second, the property is mapped to a primary key/foreign key relationship between the tables of the parent element type and the child element type. For example, Figure C-9 shows how the reference to the Item element type is mapped:



```
<!ELEMENT SalesOrder (OrderDate, CustomerNumber, Item+)>
<!ATTLIST SalesOrder
          Number CDATA #REQUIRED>
<!ELEMENT Item (PartNumber, Quantity, Price)>
<!ATTLIST Item
          Number CDATA #REQUIRED>
```

```
class SalesOrder
{
   int    number;
   Date   orderDate;
   int    customerNumber;
   Item[] items;
}
```

```
class Item
{
   int        number;
   String     partNumber;
   int        quantity;
   float      price;
}
```

```
CREATE TABLE SalesOrders
   (Number         INTEGER,
    OrderDate      DATE,
    CustomerNumber INTEGER,
    PRIMARY KEY    (Number))
```

```
CREATE TABLE Items
   (SONumber    INTEGER,
    Number      INTEGER,
    PartNumber  VARCHAR(10),
    Quantity    INTEGER,
    Price       DECIMAL(5,2),
    PRIMARY KEY (SONumber, Number),
    FOREIGN KEY (SONumber)
       REFERENCES SalesOrders (Number))
```

Figure C-9 Map references to complex element types to pointers then to keys

It is important to note that the primary key can be in the table of either the parent element type or the child element type. For example, consider the relationship between sales orders and customers: A customer can have many sales orders, so there is a foreign key in the sales order table that points to a primary key in the customer table. To demonstrate that the primary key can be in the table of the parent or child element type, Example C-7 on page 594 shows an XML document where the primary key is in the table of the parent element type (the Customer element type),

Example: C-7 XML document with primary key from parent element

```
<Customer>

   <!-- The following element types map to
           columns in the customer table. -->

   <Number>456</Number>
   <Street>123 Main St.</Street>
   <City>Chicago</City>
   <State>IL</State>
   <PostCode>60609</PostCode>
   <Country>US</Country>

   <!-- SalesOrder maps to the sales order table. -->

   <SalesOrder Number="123">
         <OrderDate>2003-7-28</OrderDate>
   </SalesOrder>
   <SalesOrder Number="124">
      <OrderDate>2003-7-30</OrderDate>
   </SalesOrder>
   <SalesOrder Number="125">
         <OrderDate>2003-8-1</OrderDate>
   </SalesOrder>
</Customer>
```

and Example C-8 shows an XML document where the primary key is in table of the child element type (again the Customer element type):

Example: C-8 XML document with primary key from child element

```
<SalesOrder Number="123">

   <!-- OrderDate and the Number attribute map
           to columns in the sales order table. -->

   <OrderDate>2003-7-28</OrderDate>

   <!-- Customer maps to the customer table. -->

   <Customer>
      <Number>456</Number>
```

```
    <Street>123 Main St.</Street>
    <City>Chicago</City>
    <State>IL</State>
    <PostCode>60609</PostCode>
    <Country>US</Country>
  </Customer>
</SalesOrder>
```

Mapping complicated content models

Although the details are beyond the scope of this book, here is a quick summary of how more complicated content models are mapped. (For simplicity, the term *child element* is used instead of the more correct *reference to an element type*.)

- Optional child elements (? operator) are mapped to nullable columns.
- Child elements that occur in choice groups (as opposed to sequences) are mapped to nullable columns.
- Child elements that can be repeated (* and + operators) must be mapped to separate tables.
- Subgroups can largely be ignored. That is, child elements are mapped according to the rules in the previous sections, regardless of whether they are in a subgroup.
- Subgroups are important in determining whether a child element is optional or may be repeated. This affects how the child element is mapped.

For more information, see the previously mentioned article on XML.com.

(Not) mapping wrapper element types

Many XML schemas contain element types that exist only to help organize the data in the XML document. For example, in the following XML schema, the Address element type is not strictly necessary, as the parts of the address still belong to the customer:

```
<!ELEMENT Customer (Number, Name, Address)>
<!ELEMENT Address (Street, City, State, PostCode, Country)>
```

Element types like Address are called *wrapper element types* because they wrap a set of child elements. They generally have no corresponding structure in the database.

> **Note:** Wrapper element types are completely unrelated to wrappers—such as the XML wrapper—in DB2 Information Integrator. A wrapper element type is an element type whose sole purpose is to "wrap" (surround) other element types. A wrapper in DB2 Information Integrator is a piece of software whose job is to "wrap" a data source. That is, transfer data between the data source and DB2 Information Integrator.

For example, the Address element type could be mapped to a separate address table as shown in Example C-9.

Example: C-9 Wrapper element mapped to separate table

```
CREATE TABLE Customers (Number    VARCHAR(10),
                        Name      VARCHAR(30),
                        AddressID INTEGER,
                        FOREIGN KEY AddressID REFERENCES Addresses (ID))
CREATE TABLE Addresses (ID       INTEGER,
                        Street   VARCHAR(50),
                        City     VARCHAR(30),
                        State    CHAR(2),
                        PostCode VARCHAR(9),
                        Country  CHAR(2),
                        PRIMARY KEY (ID))
```

but it is more likely that it will be ignored in the mapping and its children stored directly in the customer table (Example C-10).

Example: C-10 Wrapper element mapped to same table

```
CREATE TABLE Customers (Number   VARCHAR(10),
                        Name     VARCHAR(30),
                        Street   VARCHAR(50),
                        City     VARCHAR(30),
                        State    CHAR(2),
                        PostCode VARCHAR(9),
                        Country  CHAR(2))
```

Note: It is possible to ignore wrappers —and map their child elements as if they belonged to the parent element type— only if the wrapper occurs at most once in the parent. If the wrapper can occur more than once, it must be mapped to a separate table.

Wrappers can also contain other wrappers. For example, the Address element type itself is wrapped inside a ContactInformation element type. As long as each wrapper occurs only once, its attributes and child elements can be mapped as if they belonged to the parent of the highest level wrapper.

```
<!ELEMENT Customer (Number, ContactInformation)>
<!ELEMENT ContactInformation (Address, Phone)>
<!ELEMENT Address (Street, City, State, PostCode, Country)>
```

Summary of the object-relational mapping

The basic object-relational mapping can be summarized as follows:

- Complex element types are mapped to tables
- Attributes are mapped to columns
- References in the content model to simple element types are mapped to columns
- References in the content model to complex element types are mapped to primary key/foreign key relationships

The following points are also important:

- Names in the XML schema can be different from names in the database schema
- Data types in the XML schema can be different from data types in the database schema
- The primary key can be in the table of the parent element type or the child element type
- Wrapper element types may be ignored during the mapping process

D

DAD DTD reference

This appendix describes the DTD used by DAD documents. The DAD DTD actually describes three different document types: XML column DAD documents, SQL mapping DAD documents, and RDB node mapping DAD documents. Because of this, it is somewhat general and cannot show all of the constraints that actually exist for each document type.

To make the structure of each type of DAD document more understandable, this appendix describes four DTDs — the actual DAD DTD and the DTDs that describe each document type. Note that only the actual DAD DTD is shipped with the XML Extender. The other DTDs are described here for illustrative purposes only.

DTD for DAD documents

This is the actual DTD for DAD documents.

```
<!-- DAD is the root element type -->
<!ELEMENT DAD ((schemabindings |dtdid)?,validation,(Xcolumn|Xcollection))>
<!ELEMENT schemabindings (nonamespacelocation)>
<!ELEMENT nonamespacelocation EMPTY>
<!ATTLIST nonamespacelocation location CDATA #REQUIRED>
<!-- this is where to specify absolute path (assessible from db2 server)
     to xml schema file to be used for validation -->
<!ELEMENT dtdid (#PCDATA)>
<!ELEMENT validation (#PCDATA)>
<!-- Xcolumn elements describe an XML column and side tables -->
<!ELEMENT Xcolumn (table*)>
<!ELEMENT table (column*)>
<!ATTLIST table
          name    CDATA #REQUIRED
          key     CDATA #IMPLIED
          orderBy CDATA #IMPLIED>
<!ELEMENT column EMPTY>
<!ATTLIST column
          name             CDATA #REQUIRED
          type             CDATA #IMPLIED
          path             CDATA #IMPLIED
          multi_occurrence CDATA #IMPLIED>
<!-- Xcollection elements describe an XML collection -->
<!ELEMENT Xcollection (SQL_stmt?,prolog, doctype?, stylesheet?, root_node)>
<!ELEMENT SQL_stmt (#PCDATA)>
<!ELEMENT prolog (#PCDATA)>
<!ELEMENT doctype (#PCDATA)>
<!ELEMENT root_node (element_node)>
<!ELEMENT element_node (RDB_node?,
                        attribute_node*,
                        text_node?,
                        element_node*,
                        namespace_node*,
                        process_instruction_node*,
                        comment_node*)>
<!ATTLIST element_node
          name             CDATA #REQUIRED
          ID               CDATA #IMPLIED
          multi_occurrence CDATA "NO"
          BASE_URI         CDATA #IMPLIED>
<!ELEMENT attribute_node (column | RDB_node)>
<!ATTLIST attribute_node
          name CDATA #REQUIRED>
<!ELEMENT text_node (column | RDB_node)>
<!ELEMENT RDB_node (table+, column?, condition?)>
```

```
<!ELEMENT condition (#PCDATA)>
<!ELEMENT comment_node (#PCDATA)>
<!ELEMENT namespace_node EMPTY>
<!ATTLIST namespace_node
          name  CDATA #IMPLIED
          value CDATA #IMPLIED>
<!ELEMENT process_instruction_node (#PCDATA)>
```

XML column DAD documents

This section describes XML column DAD documents

DTD

Here is the DTD for XML column DAD documents.

```
<!-- DAD is the root element type -->
<!ELEMENT DAD (dtdid?, validation, Xcolumn>)
<!ELEMENT dtdid (#PCDATA)>
<!ELEMENT validation (#PCDATA)>
<!-- Xcolumn elements describe an XML column and side tables -->
<!ELEMENT Xcolumn (table*)>
<!ELEMENT table (column+)>
<!ATTLIST table
          name CDATA #REQUIRED>
<!ELEMENT column EMPTY>
<!ATTLIST column
          name             CDATA #REQUIRED
          type             CDATA #REQUIRED
          path             CDATA #REQUIRED
          multi_occurrence (YES | NO | Yes | No) "NO">
```

Note that any combination of upper, lower, and mixed case of yes and now is usually allowed in the DAD.

Element type and attribute reference

This section describes the element types and attributes used in XML column DAD documents.

column

Syntax: `<!ELEMENT column EMPTY>`

Appears in: table

Description: The column element specifies the column in a side table in which the data returned by a location path expression will be stored.

column (attributes)

Syntax:

```
<!ATTLIST column
      name CDATA #REQUIRED
      type CDATA #REQUIRED
      path CDATA #REQUIRED
       multi_occurrence (YES | NO | Yes | No) "NO">
```

Description: The name attribute specifies the name of the column.

The type attribute specifies the type of the column. This can be any valid SQL type name.

The path attribute contains a location path expression that specifies the data which is to be stored in the column. For information about the subset of XPath that is supported, see the “Working with an XML Extender location path” in the XML Extender Administration and Programming manual.

The multi_occurrence attribute specifies whether the location path expression in the path attribute can return multiple values. Its value must be YES or NO; by default it is NO. If the value is YES, then the DAD document cannot map any other columns to the side table in which this column occurs.

DAD

Syntax:

```
<!ELEMENT DAD (dtdid?, validation, Xcolumn)>
```

Appears in: --

Description: The DAD element is the root element type for DAD documents.

dtdid

Syntax:

```
<!ELEMENT dtdid (#PCDATA)>
```

Appears in: DAD

Description: The dtdid element contains the ID of the DTD against which the XML document is to be validated. This value is used to retrieve the DTD from the DTD_REF table, so its value must be in the DTDID column of that table.

table

Syntax: `<!ELEMENT table (column+)>`

Appears in: Xcolumn

Description: The table element describes a side table. It contains a list of column elements, each of which describes a column in which data from the XML document will be stored.

table (attributes)

Syntax:

```
<!ATTLIST table
                 name CDATA #REQUIRED>
```

Description: The name attribute specifies the name of the side table.

validation

Syntax: `<!ELEMENT validation (#PCDATA)>`

Appears in: DAD

Description: The validation element specifies whether the XML document is to be validated before it is stored in the XML column. The value of the validation element must be YES or NO (mixed case is also allowed).

Xcolumn

Syntax: `<!ELEMENT Xcolumn (table*)>`

Appears in: DAD

Description: The Xcolumn element specifies that the DAD document is an XML column DAD document. The Xcolumn element contains a list a side tables (if any) in which data from the XML document is to be stored.

SQL mapping DAD documents

This section describes SQL mapping DAD documents

DTD

Here is the DTD for SQL mapping DAD documents.

```
<!-- DAD is the root element type -->
<!ELEMENT DAD (dtdid?, validation, Xcollection>)
<!ELEMENT dtdid (#PCDATA)>
<!ELEMENT validation (#PCDATA)>
```

```
<!-- Xcollection elements describe an XML collection -->
<!ELEMENT Xcollection (SQL_stmt, prolog, doctype?, root_node)>
<!ELEMENT SQL_stmt (#PCDATA)>
<!ELEMENT prolog (#PCDATA)>
<!ELEMENT doctype (#PCDATA)>
<!ELEMENT root_node (element_node)>
<!ELEMENT element_node (attribute_node*,
                        (text_node? | element_node*),
                        namespace_node*,
                        process_instruction_node*,
                        comment_node*)>
<!ATTLIST element_node
          name             CDATA #REQUIRED
          ID               CDATA #IMPLIED
          multi_occurrence (YES | NO) "NO"
          BASE_URI         CDATA #IMPLIED>
<!ELEMENT attribute_node (column)>
<!ATTLIST attribute_node
          name CDATA #REQUIRED>
<!ELEMENT text_node (column)>
<!ELEMENT comment_node (#PCDATA)>
<!ELEMENT namespace_node EMPTY>
<!ATTLIST namespace_node
          name  CDATA #IMPLIED
          value CDATA #IMPLIED>
<!ELEMENT process_instruction_node (#PCDATA)>
<!ELEMENT column EMPTY>
<!ATTLIST column
          name             CDATA #REQUIRED
          type             CDATA #IMPLIED>
<!ELEMENT attribute_node (column)>
<!ATTLIST attribute_node
          name CDATA #REQUIRED>
```

Element type and attribute reference

This section describes the element types and attributes used in *SQL mapping DAD documents*.

attribute_node

Syntax: `<!ELEMENT attribute_node (column)>`

Appears in: element_node

Description: The attribute_node element maps a column selected by the SQL statement to an attribute.

attribute_node (attributes)

Syntax:

```
<!ATTLIST attribute_node
                  name CDATA #REQUIRED>
```

Description: The name attribute specifies the name of the attribute.

column

Syntax:

```
<!ELEMENT column EMPTY>
```

Appears in: attribute_node, text_node

Description: The column element specifies the column selected by the SQL statement to which an element or attribute is mapped. (Because a text_node can only appear in an element_node, a column element in a text_node element effectively maps an element to a column.)

column (attributes)

Syntax:

```
<!ATTLIST column
                   name CDATA #REQUIRED
                  type CDATA #IMPLIED>
```

Description: The name attribute specifies the name of the column. The type attribute is ignored in SQL composition.

DAD

Syntax:

```
<!ELEMENT DAD (dtdid?, validation, Xcollection)>
```

Appears in: --

Description: The DAD element is the root element type for DAD documents.

doctype

Syntax:

```
<!ELEMENT doctype (#PCDATA)>
```

Appears in: Xcollection

Description: The doctype element contains the DOCTYPE declaration to be used in the XML document, except for the starting less than sign (<) and the ending greater than sign (>). For example:

```
<doctype>!DOCTYPE Order SYSTEM
"dxx_install/samples/db2xml/dtd/getstart.dtd"</doctype>
```

If this includes an internal subset, the less than and greater than signs used in element type, attribute, entity, and notation declarations, as well as in processing instructions and comments, must be escaped with references to the lt and gt entities.It is not necessary to escape the percent sign (%) in references to parameter entities, although doing so does no harm. “%” does not have a predefined entity. Only “<“ and “&” need to be escaped with predefined entities.

dtdid

Syntax:

```
<!ELEMENT dtdid (#PCDATA)>
```

Appears in: DAD

Description: The dtdid element contains the ID of the DTD against which the XML document is to be validated. This value is used to retrieve the DTD from the DTD_REF table, so its value must be in the DTDID column of that table. If the value is not in the DTD_REF table, it must specify an absolute path by which the DTD can be accessed by DB2.

element_node

Syntax:

```
<!ELEMENT element_node (attribute_node*,
                        (text_node? | element_node*),
                        namespace_node*,
                        process_instruction_node*,
                        comment_node*)>
```

Appears in: root_node, element_node

Description: Along with attribute_node and text_node elements, element_node elements define the structure of the XML document to be published. (It is easiest to think of these elements as forming a template for the actual XML document.) An element_node element:

- Contains one attribute_node element for each attribute of the corresponding element in the XML document. Each attribute_node element specifies the column selected by the SQL statement to which the attribute is mapped.
- Contains a text_node element if the element is mapped to a column selected by the SQL statement. The text_node element specifies the column to which the element is mapped. If an element_node element contains a text_node element, it cannot contain any element_node elements.

- Contains one element_node element for each child of the corresponding element in the XML document. If an element_node element contains other element_node elements, it cannot contain a text_node element.
- namespace_node, process_instruction_node, or comment_node have not been implemented.

element_node (attributes)

Syntax:

```
<!ATTLIST element_node
        name             CDATA #REQUIRED
        ID               CDATA #IMPLIED
        multi_occurrence (YES | NO | Yes | No) "NO"
        BASE_URI         CDATA #IMPLIED>
```

Description: The name attribute specifies the name of the element.

The multi_occurrence attribute specifies whether the corresponding element may occur more than once in its parent. Its value must be YES or NO; by default, it is NO.

The ID and BASE_URI attributes have not been implemented.

namespace_node (attributes)

Syntax:

```
<!ATTLIST namespace_node
          name CDATA #IMPLIED
          value CDATA #IMPLIED>
```

Description: The name attribute specifies the namespace prefix. This node is currently not implemented. It is reserved for future use.

prolog

Syntax:

```
<!ELEMENT prolog (#PCDATA)>
```

Appears in: Xcollection

Description: The prolog element specifies the XML declaration to be used in the XML document. The value of the prolog is the XML declaration, except for the starting less than sign (<) and the ending greater than sign (>). For example:

```
<prolog>?xml version="1.0"?</prolog>
```

Note that:

- This “prolog” is different from the prolog specified in the XML 1.0 recommendation. In particular, this prolog can only contain an XML declaration. It cannot contain any processing instructions or comments, nor can it contain a DOCTYPE declaration. (The DOCTYPE declaration is specified with the doctype element.)

- The result document is not converted to the code page specified by the encoding declaration. It is just written into the resulting XML document, as part of the output of the XML declaration. XML Extender uses the code page of the server when creating the XML document. Thus, the generated document is encoded in the database code page.

root_node

Syntax: `<!ELEMENT root_node (element_node)>`

Appears in: Xcollection

Description: The root_node element is a container for the element_node that specify the structure of the XML document.

SQL_stmt

Syntax: `<!ELEMENT SQL_stmt (#PCDATA)>`

Appears in: Xcollection

Description: The SQL_stmt element specifies the SELECT statement that retrieves data to be returned as an XML document. The SELECT statement must structure the result set as follows:

- Columns in the select list must be grouped by the table they come from.
- The first column in each group must uniquely identify a row for the table consisting of columns in its group. It can either be a single column primary key retrieved from the table or a generated key.
- The groups must be ordered from left to right according to the nesting hierarchy in the XML document.

stylesheet

Syntax: `<!ELEMENT stylesheet (#PCDATA)>`

Appears in: Xcollection

Description: During composition, you can specify instructions for stylesheets, using the stylesheet tag excluding the beginning "<" and ending ">", in the form '?xml-stylesheet' (S PseudoAtt)* S? '?' as specified in `http://www.w3.org/TR/xml-stylesheet`. As in `<doctype>`, less than and amperstand signs must be escaped with their predefined entities.

text_node

Syntax:

```
<!ELEMENT text_node (column)>
```

Appears in: element_node

Description: The text_node element maps a column selected by the SQL statement to an element. (The mapped element is specified by the parent of the text_node element.)

validation

Syntax:

```
<!ELEMENT validation (#PCDATA)>
```

Appears in: DAD

Description: The validation element specifies whether the XML document is to be validated after it is created. The value of the validation element must be YES or NO. Mixed case is also allowed.

Xcollection

Syntax:

```
<!ELEMENT Xcollection (SQL_stmt, prolog, doctype?,
                          stylesheet?, root_node)>
```

Appears in: DAD

Description: The Xcollection element specifies that the DAD document is a SQL mapping DAD document or RDB node DAD document. The XML Extender determines the document type by checking if the Xcollection element contains a SQL_stmt element.

RDB node mapping DAD documents

This section describes RDB node mapping DAD documents

DTD

Here is the DTD for RDB node mapping DAD documents.

```
<!-- DAD is the root element type -->
<!ELEMENT DAD (dtdid?, validation, Xcollection)>
<!ELEMENT dtdid (#PCDATA)>
<!ELEMENT validation (#PCDATA)>
<!-- Xcollection elements describe an XML collection -->
<!ELEMENT Xcollection (prolog, doctype?, root_node)>
<!ELEMENT prolog (#PCDATA)>
<!ELEMENT doctype (#PCDATA)>
<!ELEMENT root_node (element_node)>
<!ELEMENT element_node (RDB_node?,
                        attribute_node*,
                        (text_node? | element_node*),
                        namespace_node*,
                        process_instruction_node*,
                        comment_node*)>
<!ATTLIST element_node
          name             CDATA #REQUIRED
          ID               CDATA #IMPLIED
          multi_occurrence CDATA "NO"
          BASE_URI         CDATA #IMPLIED>
<!ELEMENT attribute_node (RDB_node)>
<!ATTLIST attribute_node
          name CDATA #REQUIRED>
<!ELEMENT text_node (RDB_node)>
<!ELEMENT RDB_node (table+, column?, condition?)>
<!ELEMENT condition (#PCDATA)>
<!ELEMENT comment_node (#PCDATA)>
<!ELEMENT namespace_node EMPTY>
<!ATTLIST namespace_node
          name  CDATA #IMPLIED
          value CDATA #IMPLIED>
<!ELEMENT process_instruction_node (#PCDATA)>
<!ELEMENT table EMPTY>
<!ATTLIST table
          name    CDATA #REQUIRED
          key     CDATA #IMPLIED
          orderBy CDATA #IMPLIED>
<!ELEMENT column EMPTY>
<!ATTLIST column
          name CDATA #REQUIRED
          type CDATA #IMPLIED>
```

Element type and attribute reference

This section describes the element types and attributes used in RDB node mapping DAD documents.

attribute_node

Syntax:

```
<!ELEMENT attribute_node (RDB_node)>
```

Appears in: element_node

Description: The attribute_node element maps a column in a table to an attribute.

attribute_node (attributes)

Syntax:

```
<!ATTLIST attribute_node
          name CDATA #REQUIRED>
```

Description: The name attribute specifies the name of the attribute. This name must be different from the names of all other attributes and all elements that are mapped to columns (except during decomposition).

column

Syntax:

```
<!ELEMENT column EMPTY>
```

Appears in: RDB_node

Description: The column element specifies the column in a table to which an element or attribute is mapped. (Because a text_node can only appear in an element_node, a column element in a text_node element effectively maps an element to a column.) The column must be in the table specified by the sibling table element.

column (attributes)

Syntax:

```
<!ATTLIST column
          name CDATA #REQUIRED
          type CDATA #IMPLIED>
```

Description: The name attribute specifies the name of the column.

The type attribute specifies the type of the column. This can be any valid SQL type name, including the name of a user-defined type when doing decomposition. The type attribute is ignored during composition.

condition

Syntax: `<!ELEMENT condition (#PCDATA)>`

Appears in: RDB_node

DAD type: RDB node mapping

Description: The meaning of a condition element depends on where its parent RDB_node element appears.

If the RDB_node element is a child of the top-most element_node element, then the condition element specifies the columns by which 2 tables are related. The join conditions must be of the form *table-name.column-name* = *table-name.column-name*. The join conditions must be separated by whitespace, followed by the keyword AND, followed by whitespace.

Notes:

- It does NOT matter whether the name of the parent table and column appears on the left side of the join condition and/or the name of the child table and column appears on the right side of the join condition.
- Note that the ANDs are not significant; they are simply used to separate the join conditions; that is, all predicates of the join condition are NOT applied simultaneously (like in a “normal” SQL query)

If the RDB_node element occurs in an attribute_node or text_node element, then the condition element specifies a condition that restricts the data retrieved for the attribute or text node when publishing relational data as XML.In this case, the condition must be a valid SQL predicate.

DAD

Syntax: `<!ELEMENT DAD (dtdid?, validation, Xcollection)>`

Appears in: --

Description: The DAD element is the root element type for DAD documents.

doctype

Syntax: `<!ELEMENT doctype (#PCDATA)>`

Appears in: Xcollection

Description: The doctype element contains the DOCTYPE declaration to be used when publishing data in an XML document, except for the starting less than sign (<) and the ending greater than sign (>). For example:

```
<doctype>!DOCTYPE Order SYSTEM
"dxx_install/samples/db2xml/dtd/getstart.dtd"</doctype>
```

If this includes an internal subset, the less than and greater than signs used in element type, attribute, entity, and notation declarations, as well as in processing instructions and comments, must be escaped with references to the lt and gt entities. It is not necessary to escape the percent sign (%) in references to parameter entities, although doing so does no harm. “%” does not have a predefined entity. Only “<“ and “&” need to be escaped with predefined entities.

dtdid

Syntax:

```
<!ELEMENT dtdid (#PCDATA)>
```

Appears in: DAD

Description: The dtdid element contains the ID of the DTD against which the XML document is to be validated. This value is used to retrieve the DTD from the DTD_REF table, so its value must be in the DTDID column of that table. It must also be the value of the system identifier used in the DOCTYPE declaration, if one is specified by the doctype element. If the value is not in the DTD_REF table, it must specify an absolute path by which the DTD can be accessed by DB2.

element_node

Syntax:

```
<!ELEMENT element_node (RDB_node?,
                        attribute_node*,
                        (text_node? | element_node*),
                        namespace_node*,
                        process_instruction_node*,
                        comment_node*)>
```

Appears in: root_node, element_node

Description: Along with attribute_node and text_node elements, element_node elements define the structure of the XML document to be published. (It is easiest to think of these elements as forming a template for the actual XML document.) An element_node element:

- Must contain an RDB_node element if it is the top-most element_node element — that is, the element_node element in the root_node element. The RDB_node element specifies the tables to which the document is mapped and the conditions used to join those tables.
- Contains one attribute_node element for each attribute of the corresponding element in the XML document. Each attribute_node element specifies the table and column to which the attribute is mapped.
- Contains a text_node element if the element is mapped to a column. The text_node element specifies the table and column to which the element is mapped. If an element_node element contains a text_node element, it cannot contain any element_node elements.
- Contains one element_node element for each child of the corresponding element in the XML document. If an element_node element contains other element_node elements, it cannot contain a text_node element.
- namespace_node, process_instruction_node, and comment_node elements have not been implemented.

element_node (attributes)

Syntax:

```
<!ATTLIST element_node
          name              CDATA #REQUIRED
          ID                CDATA #IMPLIED
          multi_occurrence  (YES | NO) "NO"
          BASE_URI          CDATA #IMPLIED>
```

Description: The name attribute specifies the name of the element. If the element is mapped to a column (it contains a text_node element), then this name must be different from the names of all attributes and the names of all other elements that are mapped to columns (except during decomposition).

- The multi_occurrence attribute specifies whether the corresponding element may occur more than once in its parent. Its value must be YES or NO. Mixed case is also allowed. By default, multi_occurrence is NO.
- The ID and BASE_URI attributes are not implemented.

namespace_node (attributes)

Syntax:

```
<!ATTLIST namespace_node
                 name CDATA #IMPLIED
                value CDATA #IMPLIED>
```

Description: The node is currently not implemented, and is reserved for future use.

prolog

Syntax:

```
<!ELEMENT prolog (#PCDATA)>
```

Appears in: Xcollection

Description: The prolog element specifies the XML declaration to be used when publishing data to an XML document. The value of the prolog is the XML declaration, except for the starting less than sign (<) and the ending greater than sign (>). For example:

```
<prolog>?xml version="1.0"?</prolog>
```

Note that:

- This “prolog” is different from the prolog specified in the XML 1.0 recommendation. In particular, this prolog can only contain an XML declaration. It cannot contain any processing instructions or comments, nor can it contain a DOCTYPE declaration. (The DOCTYPE declaration is specified with the doctype element.)
- The result document is not converted to the code page specified by the encoding declaration. It is just written into the resulting XML document, as part of the output of the XML declaration. XML Extender uses the code page of the server when creating the XML document. Thus, the generated documented is encoded in the database code page.

RDB_node

Syntax:

```
<!ELEMENT RDB_node (table+, column?, condition?)>
```

Appears in: element_node, attribute_node, text_node

Description: RDB_nodes have two different functions and syntaxes, depending on where they appear.

In the top-most element_node element, the RDB_node element specifies the tables to which the XML document is mapped and the conditions used to join these tables. The syntax in this case is:

```
<!ELEMENT RDB_node (table+, condition)>
```

where each table element specifies a table and the condition element specifies the join conditions.

In an attribute_node or text_node element, the RDB_node element specifies the table and column to which an attribute or element is mapped. The syntax in this case is:

```
<!ELEMENT RDB_node (table, column, condition?)>
```

where the table element specifies the table, the column element specifies the column, and the condition element specifies a condition that restricts the data retrieved for the attribute or text node when publishing relational data as XML. The logical AND of all conditions for a table, is used as the filter for the retrieved data from a table.

root_node

Syntax: `<!ELEMENT root_node (element_node)>`

Appears in: Xcollection

Description: The root_node element is a container for the element_nodes that specify the structure of the XML document.

stylesheet

Syntax: `<!ELEMENT stylesheet (#PCDATA)>`

Appears in: Xcollection

Description: During composition, you can specify instructions for stylesheets, using the stylesheet tag excluding the beginning "<" and ending ">", in the form '?xml-stylesheet' (S PseudoAtt)* S? '?' as specified in `http://www.w3.org/TR/xml-stylesheet`. As in `<doctype>`, less than and amperstand signs must be escaped with their predefined entities.

table

Syntax: `<!ELEMENT table EMPTY>`

Appears in: RDB_node

Description: The table element specifies a table.

table (attributes)

Syntax:

```
<!ATTLIST table
          name CDATA #REQUIRED
          key CDATA #IMPLIED
          orderBy CDATA #IMPLIED>
```

Description: The name attribute specifies the name of the table.

The key attribute contains a space-separated list of the columns in the primary key. It is required only when shredding XML documents and only in the <table> tag(s) children of the top <RDB_node> tag.

The orderBy attribute specifies the name of the column used to sort the rows for the table when publishing data as XML.

text_node

Syntax:

```
<!ELEMENT text_node (column)>
```

Appears in: element_node

Description: The text_node element maps a column to an element. (The mapped element is specified by the parent of the text_node element.)

validation

Syntax:

```
<!ELEMENT validation (#PCDATA)>
```

Appears in: DAD

Description: The validation element specifies whether the XML document is to be validated before it is shredded or after it is created. The value of the validation element must be YES or NO. Mixed case is also allowed.

Xcollection

Syntax:

```
<!ELEMENT Xcollection (SQL_stmt, prolog, doctype?,
                       root_node)>
```

Appears in: DAD

Description: The Xcollection element specifies that the DAD document is a SQL mapping DAD document or RDB node DAD document. The XML Extender determines the document type by checking if the Xcollection element contains a SQL_stmt element.

E

Sample XML Tools

This appendix introduces XMLFilters and describes a number of sample tools developed during this project.

- Using XMLFilters
- The NameChanger tool
- The SAXCutter tool
- The TableCutter tool

Using XMLFilters

An XMLFilter is a SAX program that serves as both a SAX application and a SAX parser. As a SAX application, it implements the ContentHandler interface and listens for SAX events. As a SAX parser, it implements the XMLReader interface and fires new SAX events. Thus, it sits in a chain of SAX programs and fires new events in response to the events it receives (Figure E-1).

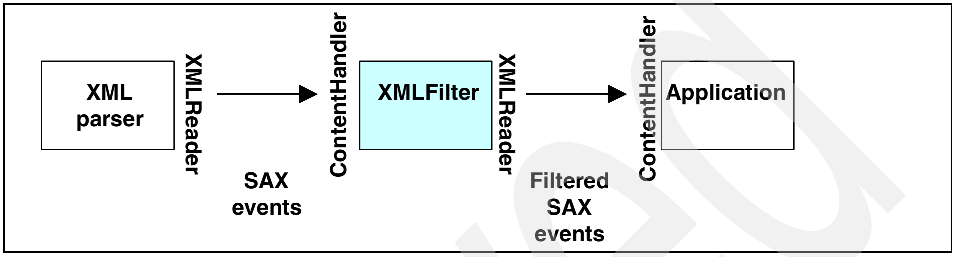


Figure E-1 An XML filter

XMLFilters can perform simple transformations. These are usually linear, where the XMLFilter fires a new event in response to each event it receives. As a simple example, you could use an XMLFilter to translate element type names from English to French. For example, in response to receiving a startElement event for a “Book” element, an XMLFilter could fire a startElement event for a “Livre” element. And in response to receiving a startElement event for a “Name” element, it could fire a startElement event for a “Nom” element.

XMLFilters can be used for more complex tasks as well. This appendix discusses two XMLFilter applications. The first XMLFilter is NameChanger, which changes child element names and attribute names depending on their context — that is, depending on their parent element type. For example, it can change the name of the “Title” element to “BookTitle” when it occurs inside a “Book” element and to “ArticleTitle” when it occurs inside an “Article” element. The second XMLFilter is SAXCutter, which cuts a document with repeating child elements (such as a list of sales orders) into a set of separate documents (such as individual sales orders).

Implementing XMLFilters

The easiest way to implement an XMLFilter is to extend the XMLFilterImpl class. This class implements both SAX application interfaces (ContentHandler, DTDHandler, ErrorHandler, and EntityResolver) and SAX parser interfaces (XMLFilter, which extends XMLReader). It does so in a very simple way: in response to each SAX event it receives, it passes the same event to the handlers registered with it. For example, when it receives a startElement event, it calls startElement in the ContentHandler that is registered with its XMLReader interface (Figure E-2).

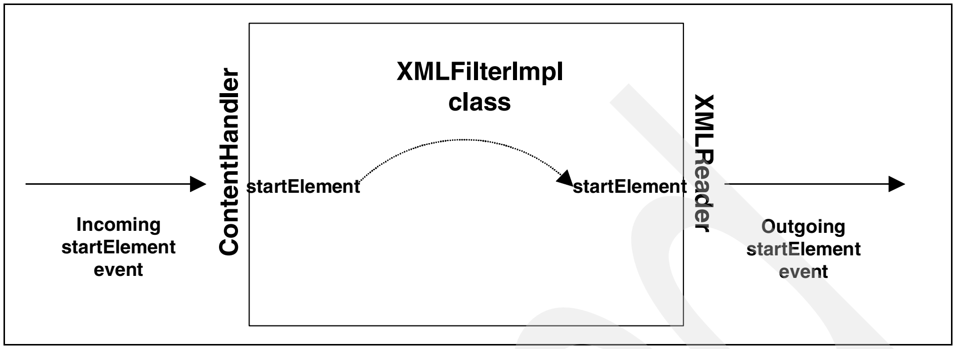


Figure E-2 The XML FilterImpl class

Because of this, a class that extends XMLFilterImpl only needs to override the methods it is interested in. For example, to implement an XMLFilter that translates element type names, you would need to override only two methods: startElement and endElement. Typically, any overridden methods will call the same method in its superclass (XMLFilterImpl). This allows the XMLFilterImpl code to pass the event on to the next SAX application (Figure E-3).

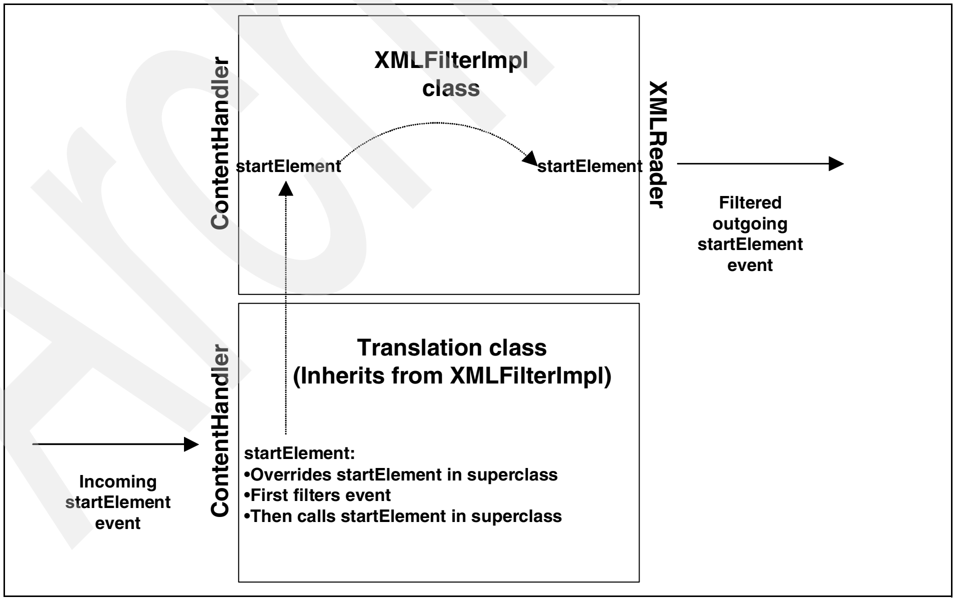

XMLFilterImpl class
ContentHandler
startElement
startElement
XMLReader
Filtered outgoing startElement event
Translation class (Inherits from XMLFilterImpl)
ContentHandler
startElement:
•Overrides startElement in superclass
•First filters event
•Then calls startElement in superclass
Incoming startElement event

Figure E-3 Extending the XML FilterImpl class

This is shown in the code in Example E-1:

Example: E-1 Translator class

```
public class Translator extends XMLFilterImpl
   public Translator()
   {
      super();
   }
   public Translator(XMLReader xmlReader)
   {
      super(xmlReader);
   }
   public void startElement(String uri, String localName, String qName,
                    Attributes attrs)
      throws SAXException
   {
      String newLocalName = translate(localName);
      super.startElement(uri, newLocalName, "", attrs);
   }
   public void endElement(String uri, String localName, String qName),
      throws SAXException
   {
      String newLocalName = translate(localName);
      super.endElement(uri, newLocalName, "", attrs);
   }
   private String translate(String name)
   {
      // Code not shown.
   }
```

Using XMLFilters from a SAX application

Applications that use XMLFilters are SAX applications. That is, they implement interfaces such as ContentHandler. To use an XMLFilter, an application must:

- Instantiate an XMLReader (parser).
- Instantiate an XMLFilter and register the XMLReader with the XMLFilter.
- Register itself with the XMLFilter.
- Call the parse method in the XMLReader.

See Example E-2 on page 623:

Example: E-2 RunTranslator class

```
public class RunTranslator {
   // Constructors not shown
   public void runTranslator(String xmlFilename)
   {
      XMLReader xmlReader = getXMLReader();                // Step 1
      Translator translator = new Translator(xmlReader);   // Step 2
      translator.setContentHandler(this);                  // Step 3
      InputSource src = new InputSource(new FileInputStream(xmlFilename));
      xmlReader.parse(src);                                // Step 4
   }
   // ContentHandler implementation not shown. Note that all of the
   // work of the application is done in ContentHandler methods.
```

In this manner, a SAX application can chain together a number of XMLFilters, passing each to the constructor of the next. A transformation is thus broken down into a number of distinct steps, each performed by a different XMLFilter.

The NameChanger sample

The NameChanger sample consists of two Java classes: NameChanger and NameTester. NameChanger is an XMLFilter that changes element and attribute names. NameTester is a sample SAX application that uses NameChanger.

The NameChanger tool

The NameChanger class is an XMLFilter that changes names based on context. That is, it can change an element or attribute name based on a parent-specific basis. For example, it might change the name of the Title element to BookTitle when it appears in a “Book” element and to “ChapterTitle” when it appears in a “Chapter” element. New names do not have to concatenate the parent and child names. For example, you might change the name of a “Number” element to “ISBN” when it appears in the a “Book” element.

The NameChanger class is useful when shredding XML documents into relational tables, since the XML Extender in the past did not allow any two leaf nodes (element types or attributes) to have the same name.

An application specifies what names to change by passing in a hash table. The keys of the hash table give the name of the parent element type and the child element type or the attribute, separated by a caret (^). For example:

```
Chapter^Title
Book^Title
```

The values of the hash table are the new names. Since the java.util.Property class extends the java.util.Hashtable class, the sets of element type and attribute names to change can be stored in Java property files. For example:

```
Chapter^Title=ChapterTitle
Book^Title=BookTitle
```

Most of the work in NameChanger is done in the startElement and endElement classes, which call private functions to change element type and attribute names

Note: The NameChanger class is not namespace aware, and must be run with XML namespace processing off. That is, the application must set the value of the http://xml.org/features/namespaces feature in the XMLReader to false and the value of the http://xml.org/features/namespace-prefixes feature to true.

The code for the NameChanger class is shown in Example E-3:

Example: E-3 NameChanger class

```
import java.util.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
/*
 * XMLFilter that changes the names of elements attributes
 * based on their context.
 *
 * <p>NameChanger changes the names of elements and attributes
 * based on their context -- that is, the parent element in which
 * they appear.</p>
 *
 * <p>The names to be changed are passed in in hashtables. The
 * keys of these hashtables have the form:</p>
 *
 * <pre>
 *    &lt;parent-name>^&lt;child-name>
 * </pre>
 *
 * <p>where &lt;parent-name> is the local name of the parent
 * element and &gt;child-name> is the local name of the child
 * element or attribute.</p>
 *
 * <p>The values of these hashtables are the names to which the child
 * element or attribute is to be changed.</p>
 *
 * <p>Note that a Properties object is a Hashtable, so this information
 * is very easily stored in a Properties file. For example:</p>
 *
 * <pre>
 *    Magazine^Title=MagazineTitle
 *    Book^Title=BookTitle
 *    Book^Number=ISBN
 * </pre>
 *
 * <p>WARNING! This class does not support XML namespaces. It must
 * be called using a parser whose namespace processing is turned off.
 * (The namespace property is false and the namespace-prefix property
 * is true.)</p>
 *
 * @version 1.0
 */
public class NameChanger extends XMLFilterImpl
   private Hashtable elementNames = null, attrNames = null;
   private Stack parentStack = new Stack();
   /*
    * Create a new NameChanger.
    */
   public NameChanger()
   {
      super();
   }
   /*
    * Create a new NameChanger and set the parent XMLReader.
    *
    * @param parent The parent XMLReader
    */
   public NameChanger(XMLReader parent)
   {
      super(parent);
   }
   /*
    * Set a hashtable of element names to change.
    *
    * <p>For more information, see the introduction.</p>
    *
    * @param elementNames The hashtable. May be null.
    */
   public void setElementTypeNames(Hashtable elementNames)
   {
      this.elementNames = elementNames;
   }
   /*
    * Set a hashtable of attribute names to change.
    *
    * <p>For more information, see the introduction.</p>
    *
    * @param attrNames The hashtable. May be null
    */
   public void setAttributeNames(Hashtable attrNames)
   {
      this.attrNames = attrNames;
   }
   //************************************************************
   //* Overridden ContentHandler methods
   //************************************************************
   public void startElement(String uri, String localName, String qName,
Attributes attrs)
      throws SAXException
   {
      String     newQName;
      Attributes newAttrs;
      // Get the new QName. This may be the same as the
      // existing QName.
      newQName = getNewQName(qName);
      // Get attributes that use the new names. These
      // may use existing names.
      newAttrs = getNewAttributes(qName, attrs);
      // Pass on the startElement event with the new names.
      super.startElement(uri, "", newQName, newAttrs);
      // Push the local name of the current element onto the stack.
      parentStack.push(qName);
   }
   public void endElement(String uri, String localName, String qName)
      throws SAXException
   {
      String newQName;
      // Pop the name of the current element off the stack.
      parentStack.pop();
      // Get the new QName. This may be the same as the
      // existing QName.
      newQName = getNewQName(qName);
      // Pass on the endElement event with the new names.
      super.endElement(uri, "", newQName);
   }
   //************************************************************
   //* Private methods
   //************************************************************
   private String getNewQName(String qName)
   {
      String key, name, parent;
      // If there are no new element names, just return the QName.
      if (elementNames == null) return qName;
      // Build a key from the parent element's QName and this
      // element's QName. If there is a corresponding new name
      // in the hashtable of element names, use it. Otherwise, use the
      // existing name.
      parent = (parentStack.empty()) ? "" : (String)parentStack.peek();
      key = parent + "^" + qName;
      name = (String)elementNames.get(key);
      if (name != null)
         return name;
      else
         return qName;
   }
   private Attributes getNewAttributes(String qName, Attributes attrs)
   {
      AttributesImpl attrsImpl;
      String         key, name, attrName;
      // If there are no new attribute names, just return the attributes.
      if (attrNames == null) return attrs;
      // Create a new AttributesImpl object and copy the attributes
      // to it, changing names as necessary.
      attrsImpl = new AttributesImpl();
      for (int i = 0; i < attrs.getLength(); i++)
      {
         // Build a key from the element's QName and the attribute's
         // QName. If there is a corresponding new name in the
         // hashtable of attribute names, use it. Otherwise, use the
         // existing name.
         attrName = attrs.getQName(i);
         key = qName + "^" + attrName;
         name = (String)attrNames.get(key);
         if (name != null) attrName = name;
         // Add the attribute.
         attrsImpl.addAttribute(attrs.getURI(i),
                                "",
                                attrName,
                                attrs.getType(i),
                                attrs.getValue(i));
      }
      return attrsImpl;
   }
```

The NameTester application

The NameTester class is a SAX application that uses the NameChanger XMLFilter. Its command line syntax is:

```
java NameTester <input-file> <output-file> <elem-file> <attr-file>
```

where <input-file> and <output-file> are the names of the XML input and output files, and <elem-file> and <attr-file> are the names of Java properties files.

The properties files specify which element and attribute names are to be changed. They use the format shown in the previous section, where the parent and child element names (or element and attribute names) are concatenated with a caret (^) to form the property name, and the property value is the new element or attribute name. For example:

```
Chapter^Title=ChapterTitle
Book^Title=BookTitle
```

The ContentHandler methods in the NameTester class write the modified XML document to a file using the FileWriter class. If you want to do something else with the modified document, you might want to use a different Writer class. For example, to store the document in a database, use the StringWriter class to first write the document to a string, then pass this string to the database.

Example E-4 shows the code for the NameTester class.

Example: E-4   NameTester class

```
import java.util.*;
import java.io.*;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
/**
 * SAX application that tests the NameChanger.
 *
 * <p>NameTester is a SAX application (ContentHandler) that uses
 * the NameChanger class to change names in an XML document, then
 * writes this to an output file.</p>
 *
 * <p>The command line syntax of the NameTester is:</p>
 *
 * <pre>
 *    java NameTester &lt;input-file> &lt;output-file> &lt;elem-file>
&lt;attr-file>
 * </pre>
 *
 * <p>where &lt;elem-file> and &lt;attr-file> are the names of property
 * files containing the new names. In these files, property names are of
 * the form:
 *
 * <pre>
 *   parent-element-name^child-element-name=new-element-name
 * </pre>
 *
 * <p>or:</p>
 *
 * <pre>
 *   element-name^attribute-name=new-attribute-name
 * </pre>
 *
 * <p>For example:</p>
 *
 * <pre>
 *    Magazine^Title=MagazineTitle
 *    Book^Title=BookTitle
 *    Book^Number=ISBN
 * </pre>
 *
 * <p>NameChanger uses a FileWriter to write to the changed document
 * to a file. If you want to do something else with the document,
 * you can modify the code. For example, to store the document in
 * a database, use a StringWriter to build a string from a document,
 * then pass this to the database with an INSERT statement.</p>
 *
 * @version 1.0
 */
public class NameTester implements ContentHandler
   private Writer writer;
   /*
    * Create a new NameTester.
    */
   public NameTester()
   {
   }
   /*
    * Run the NameTester from the command line.
    *
    * <p>See the introduction for the command line syntax.</p>
    */
   public static void main (String[] argv)
   {
      try
      {
         NameTester test = new NameTester();
         if (argv.length != 4)
            throw new Exception("Syntax: java NameTester " +
               "<input-file> <output-file> <elem-file> <attr-file>");
         else
         {
            test.run(argv[0], argv[1], argv[2], argv[3]);
         }
      }
      catch(Exception e)
      {
         e.printStackTrace();
      }
   }
   /*
    * Run the NameTester.
    *
    * @param oldXMLFilename Name of the XML file to change names of.
    * @param newXMLFilename Name of the XML file to write to.
    * @param elemFilename Name of the file containing new element type
names.
    *    May be null.
    * @param attrFilename Name of the file containing new attribute names.
    *    May be null.
    */
   public void run(String oldXMLFilename, String newXMLFilename,
                   String elemFilename, String attrFilename)
      throws Exception
   {
      XMLReader   xmlReader;
      NameChanger changer;
      // Get a new XMLReader and set namespace processing off.
      xmlReader = getXMLReader();
      xmlReader.setFeature("http://xml.org/sax/features/namespaces",
false);

xmlReader.setFeature("http://xml.org/sax/features/namespace-prefixes",
true);
      // Get a new NameChanger and set the ContentHandler to this
      // NameTester object.
      changer = new NameChanger(xmlReader);
      changer.setContentHandler(this);
      // Set the new element type and attribute names.
      changer.setElementTypeNames(getProperties(elemFilename));
      changer.setAttributeNames(getProperties(attrFilename));
      // Open the output file for writing, parse the input file, and
      // close the output file.
      writer = new FileWriter(newXMLFilename);
      changer.parse(new InputSource(new FileInputStream(oldXMLFilename)));
      writer.close();
   }
   //*********************************************************
   //* ContentHandler methods
   //*********************************************************
   /** For internal use only. */
   public void startDocument () throws SAXException
   {
   }
   /** For internal use only. */
   public void endDocument() throws SAXException
   {
   }
   /** For internal use only. */
   public void startElement (String uri, String localName, String qName,
Attributes attrs)
     throws SAXException
   {
      int i;
      // Start the element.
      try
      {
         writer.write('<');
         writer.write(qName);
         // Append the attributes. Note that this includes xmlns
attributes.
         for (i = 0; i < attrs.getLength(); i++)
         {
            writer.write(' ');
            writer.write(attrs.getQName(i));
            writer.write("=\"");
            appendEscapedString(attrs.getValue(i));
            writer.write('"');
         }
         // Close the element.
         writer.write('>');
      }
      catch (IOException io)
      {
         throw new SAXException(io);
      }
   }
   /** For internal use only. */
   public void endElement (String uri, String localName, String qName)
throws SAXException
   {
      try
      {
         writer.write("</");
         writer.write(qName);
         writer.write('>');
      }
      catch (IOException io)
      {
         throw new SAXException(io);
      }
   }
```

```
/** For internal use only. */
public void characters (char ch[], int start, int length)
  throws SAXException
{
   try
   {
      appendEscapedString(ch, start, length);
   }
   catch (IOException io)
   {
      throw new SAXException(io);
   }
}
/** For internal use only. */
public void ignorableWhitespace (char ch[], int start, int length)
   throws SAXException
{
   try
   {
      writer.write(ch, start, length);
   }
   catch (IOException io)
   {
      throw new SAXException(io);
   }
}
/** For internal use only. */
public void processingInstruction (String target, String data)
   throws SAXException
{
   try
   {
      writer.write("<?");
      writer.write(target);
      writer.write(' ');
      writer.write(data);
      writer.write("?>");
   }
   catch (IOException io)
   {
      throw new SAXException(io);
   }
}
/** For internal use only. */
public void startPrefixMapping(String prefix, String uri)
   throws SAXException
{
}
/** For internal use only. */
```

```
public void endPrefixMapping(String prefix)
   throws SAXException
{
}
/** For internal use only. */
public void setDocumentLocator (Locator locator)
{
}
/** For internal use only. */
public void skippedEntity(String name)
   throws SAXException
{
}
//**********************************************************
//* Private methods
//**********************************************************
private XMLReader getXMLReader()
   throws Exception
{
   // Get an XMLReader. See the documentation for
   // javax.xml.parsers.SAXParserFactory for information
   // about configuring your system to use your XML parser.
   SAXParserFactory factory = SAXParserFactory.newInstance();
   return factory.newSAXParser().getXMLReader();
}
private Properties getProperties(String filename)
   throws IOException
{
   Properties props;
   if (filename == null) return null;
   props = new Properties();
   props.load(new FileInputStream(filename));
   return props;
}
private void appendEscapedString(String string)
   throws IOException
{
   appendEscapedString(string.toCharArray(), 0, string.length());
}
private void appendEscapedString(char[] chars, int start, int length)
   throws IOException
{
   int    save;
   // This method appends a string to the sub-document StringBuffer.
   // It replaces &, <, >, ', and " with entity references.
   save = start;
   for (int i = start; i < start + length; i++)
   {
      switch(chars[i])
```

```
         {
            case '&':
            case '<':
            case '>':
            case '\'':
            case '"':
               // When we encounter a character that needs to be escaped as
               // an entity, append any characters that haven't been
written,
               // adjust the save point, and append the entity reference.
               if (save < i)
               {
                  writer.write(chars, save, i - save);
               }
               save = i + 1;
               switch(chars[i])
               {
                  case '&':
                     writer.write("&amp;");
                     break;
                  case '<':
                     writer.write("&lt;");
                     break;
                  case '>':
                     writer.write("&gt;");
                     break;
                  case '\'':
                     writer.write("&apos;");
                     break;
                  case '"':
                     writer.write("&quot;");
                     break;
               }
            default:
               break;
         }
      }
      // If there are any characters that haven't yet been appended,
      // append them now.
      if (save < start + length)
      {
         writer.write(chars, save, start + length - save);
      }
   }
```

The SAXCutter sample

The SAXCutter sample consists of two classes: SAXCutter and CutterTester. SAXCutter is an XMLFilter that cuts a document into smaller documents. CutterTester is a sample SAX application that uses SAXCutter.

The SAXCutter tool

The SAXCutter class is an XMLFilter that cuts a document into smaller documents. That is, when it encounters an element of a particular type, it starts a new document, using that element as the root element. This is useful for cutting large documents that consist of many repeating sub-documents into smaller documents.

For example, suppose you want to copy a large number of XML sales order documents from one database to another. You could extract them from the first database and build a single, large XML document that contains all of them, which you would then pass to the second database. Using the SAXCutter class, you could extract each document — cutting on the name of the root element type of these documents — and store it in the second database.

As input, the SAXCutter takes the namespace URI and local name of the element type you want to cut on. It then calls the complete set of SAX events (starting with startDocument and ending with endDocument) for each sub-document it finds.

Most of the work in SAXCutter is done in the startElement and endElement methods, which track whether it is inside or outside a cut element. When it is inside a cut element, it passes all events to the application. When it is outside a cut element, it traps (ignores) all events except namespace events, which it tracks to ensure that each sub-document has the proper namespace declarations.

> **Important:** The SAXCutter class is namespace aware, and must be run with XML namespace processing on. That is, the application must set the value of both the http://xml.org/features/namespaces feature and the http://xml.org/features/namespace-prefixes feature in the XMLReader to true.

Example E-5 shows the code for the SAXCutter class.:

Example: E-5 SAXCutter class

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import java.io.*;
import java.util.*;
/**
 * XMLFilter that cuts an XML document into sub-documents.
 *
 * <p>A common characteristic of data-centric XML documents is
 * that they are often a series of smaller documents with identical
 * schemas that are wrapped in a single root element and passed as
 * a larger document. For example:</p>
 *
 * <pre>
 *    &lt;SalesOrders>
 *       &lt;SalesOrder>
 *          ...
 *       &lt;/SalesOrder>
 *       &lt;SalesOrder>
 *          ...
 *       &lt;/SalesOrder>
 *       ...
 *       &lt;SalesOrder>
 *          ...
 *       &lt;/SalesOrder>
 *    &lt;/SalesOrders>
 *
 * <p>SAXCutter is an XMLFilter that cuts such documents into a series
 * of sub-documents, based on the name of a cut element type. For example,
 * the previous document would most likely be cut on the SalesOrder element
 * type. Everything outside a cut element (except namespace declarations)
 * is ignored. Everything inside the cut element (including the cut element
 * itself) is passed on to the next SAX ContentHandler in the chain.</p>
 *
 * <p>WARNING! SAXCutter is always namespace-aware and must be run with
 * XML namespace processing on. That is, the namespace and
namespace-prefixes
 * properties of the parser must be set to true.</p>
 *
 * @version 1.0
 */
public class SAXCutter extends XMLFilterImpl
  private boolean cutting;
  private int cutElementDepth;
  private String cutURI, cutLocalName;
  private Hashtable uriStacks = new Hashtable();
  /*
   * Create a new SAXCutter.
   */
  public SAXCutter()
  {
    super();
  }
  /*
   * Create a new SAXCutter and set the parent XMLReader.
   *
   * @param parent The parent XMLReader
   */
  public SAXCutter(XMLReader parent)
  {
    super(parent);
  }
  /*
   * Set the name of the element type to cut on.
   *
   * @param uri Namespace URI of the element type.
   * @param localName Local name of the element type.
   */
  public void setCutElementType(String uri, String localName)
  {
    if (uri == null)
       throw new IllegalArgumentException("SAXCutter.setCutElementType: "
+
           "If there is no namespace URI, use an empty string, not a
null.");
    cutURI = uri;
    cutLocalName = localName;
  }
  //*****************************************************
  //* Overridden ContentHandler methods
  //*****************************************************
  // The methods in ContentHandler look for an element of the
  // type to cut on. When they find this element, they start a
  // new document and pass the element's contents on to the
  // next ContentHandler in the chain. When they reach the end
  // of this element, they end the document.
  //
  // Everything outside a cut element (except prefix/URI mappings)
  // is ignored.
  /* ContentHandler method for internal use. */
  public void parse(InputSource input)
    throws IOException, SAXException
  {
    if (cutLocalName == null)
       throw new SAXException("Cutter element type not set.");
    cutting = false;
    super.parse(input);
  }
  /* ContentHandler method for internal use. */
  public void parse(String systemId)
    throws IOException, SAXException
  {
    if (cutLocalName == null)
       throw new SAXException("Cutter element type not set.");
    cutting = false;
    super.parse(systemId);
  }
  /* ContentHandler method for internal use. */
  public void startDocument()
    throws SAXException
  {
    // Trap the call to startDocument so it isn't passed on to the
application.
  }
  /* ContentHandler method for internal use. */
  public void endDocument()
    throws SAXException
  {
    // Trap the call to endDocument so it isn't passed on to the
application.
  }
  /* ContentHandler method for internal use. */
  public void startElement(String uri, String localName, String qName,
Attributes attrs)
    throws SAXException
  {
    AttributesImpl attrsImpl;
    String         nsPrefix, nsURI, attrName;
    if (uri.equals(cutURI) && localName.equals(cutLocalName))
    {
       if (!cutting)
       {
          // If the current element is a cut element and we are not
          // already cutting, then set the global variables and start
          // a new sub-document.
          cutting = true;
          cutElementDepth = 1;
          super.startDocument();
          // If there are prefix/URI mappings, we need to add xmlns
          // attributes for these, as well as calling startPrefixMapping
          // and endPrefixMapping.
          if (!uriStacks.isEmpty())
          {
             // Copy the attributes to a new AttributesImpl object so we
             // can add more xmlns attributes.
             attrsImpl = new AttributesImpl();
             for (int i = 0; i < attrs.getLength(); i++)
             {
                attrsImpl.addAttribute(attrs.getURI(i),
                                       attrs.getLocalName(i),
                                       attrs.getQName(i),
                                       attrs.getType(i),
                                       attrs.getValue(i));
             }
             // Loop through the prefix/URI mappings.
             Enumeration prefixes = uriStacks.keys();
             while (prefixes.hasMoreElements())
             {
                // Call startPrefixMapping.
                nsPrefix = (String)prefixes.nextElement();
                nsURI = (String)((Stack)uriStacks.get(nsPrefix)).peek();
                super.startPrefixMapping(nsPrefix, nsURI);
                // Add xmlns attributes that are not in the list.
                attrName = (nsPrefix.length() == 0) ? "xmlns" : "xmlns:"
+ nsPrefix;
                if (attrsImpl.getIndex(attrName) == -1)
                {
                   attrsImpl.addAttribute("", nsPrefix, attrName,
"CDATA", nsURI);
                }
             }
             // Set attrs to use the new attributes.
             attrs = attrsImpl;
          }
       }
       else // if (cutting)
       {
          // If the current element is a cut element and we are already
          // cutting, then increment the cut element depth. We need to
          // track this in case cut elements are nested inside cut
elements.
          cutElementDepth++;
       }
    }
    // If we are cutting, pass on the startElement event.
    if (cutting)
    {
       super.startElement(uri, localName, qName, attrs);
    }
  }
  /* ContentHandler method for internal use. */
  public void endElement(String uri, String localName, String qName)
    throws SAXException
  {
    if (cutting)
    {
       // Pass on the endElement event.
       super.endElement(uri, localName, qName);
       // If the current element is a cut element, decrement the
       // cut element depth.
       if (uri.equals(cutURI) && localName.equals(cutLocalName))
       {
          cutElementDepth--;
       }
       // If the cut element depth falls to 0, this sub-document is done.
       if (cutElementDepth == 0)
       {
          // Call endPrefixMapping to tear down the prefix/URI mappings.
          Enumeration prefixes = uriStacks.keys();
          while (prefixes.hasMoreElements())
          {
             String prefix = (String)prefixes.nextElement();
             super.endPrefixMapping(prefix);
          }
          // Set the cutting flag to false and end the document.
          cutting = false;
          super.endDocument();
       }
    }
  }
  /* ContentHandler method for internal use. */
  public void characters (char[] ch, int start, int length)
    throws SAXException
  {
    if (cutting)
       super.characters(ch, start, length);
  }
  /* ContentHandler method for internal use. */
  public void ignorableWhitespace (char[] ch, int start, int length)
    throws SAXException
  {
    if (cutting)
       super.ignorableWhitespace(ch, start, length);
  }
  /* ContentHandler method for internal use. */
  public void processingInstruction(String target, String data)
    throws SAXException
  {
    if (cutting)
       super.processingInstruction(target, data);
  }
  /* ContentHandler method for internal use. */
  public void skippedEntity(String name)
    throws SAXException
  {
    if (cutting)
       super.skippedEntity(name);
  }
```

```
/* ContentHandler method for internal use. */
public void startPrefixMapping(String prefix, String uri)
   throws SAXException
{
   // The SAX 2.0 specification is not clear if a default prefix
   // is a null string or an empty string. We need an empty string.
   if (prefix == null) prefix = "";
   if (cutting)
   {
      // If we are cutting, just pass on startPrefixMapping events.
      super.startPrefixMapping(prefix, uri);
   }
   else // if (!cutting)
   {
      // If we are not cutting, we need to cache the namespace
      // for later use, since it may affect the prefix/URI mappings
      // used in the cut part of the document.
      //
      // We cache URIs in a Hashtable keyed by prefix. Since each
      // prefix can be used multiple times (masking a higher level
      // use of the same prefix) we store stacks of URIs in the
      // hashtable, rather than the URIs themselves.
      // Get the stack for the prefix. If the prefix is not yet
      // used, add a new stack to the hashtable.
      Stack stack = (Stack)uriStacks.get(prefix);
      if (stack == null)
      {
         stack = new Stack();
         uriStacks.put(prefix, stack);
      }
      // Push the URI onto the stack.
      stack.push(uri);
   }
}
/* ContentHandler method for internal use. */
public void endPrefixMapping(String prefix)
   throws SAXException
{
   // The SAX 2.0 specification is not clear if a default prefix
   // is a null string or an empty string. We need an empty string.
   if (prefix == null) prefix = "";
   if (cutting)
   {
      // If we are cutting, just pass on endPrefixMapping events.
      super.endPrefixMapping(prefix);
   }
   else // if (!cutting)
   {
      // Get the stack for the prefix and pop off the top-level
```

```
         // URI. If the stack is empty, remove it from the hashtable.
         Stack stack = (Stack)uriStacks.get(prefix);
         stack.pop();
         if (stack.empty())
            uriStacks.remove(prefix);
      }
   }
```

The CutterTester application

The CutterTester class is a SAX application that uses the SAXCutter XMLFilter. Its command line syntax is:

```
java CutterTester <input-file> <local-element-type-name> [<uri>]
```

where <input-file> is the name of the XML input file and <local-element-type-name> and <uri> are the local name and optional namespace URI of the element type to cut on.

The ContentHandler methods in the CutterTester class write the sub-documents to separate files whose names are constructed by appending the numbers 1, 2, 3, ... to the basename of the input file. For example, if the name of the input file is orders.xml, the name of the output files will be orders1.xml, orders2.xml, and so on. If you want to do something else with the sub-documents, you might want to use a different Writer class. For example, to store the sub-documents in a database, use the StringWriter class to first write each document to a string, then pass this string to the database.

Example E-6 shows the code for the CutterTester class:

Example: E-6 CutterTester class

```
import java.io.*;
import java.util.*;
import org.xml.sax.*;
import javax.xml.parsers.SAXParserFactory;
/**
 * SAX application that tests the SAXCutter.
 *
 * <p>CutterTester is a SAX application (ContentHandler) that uses
 * the SAXCutter class to cut an XML document into sub-documents,
 * it then writes to separate files. The output filenames are constructed
 * by appending 1, 2, 3, ... to the base name of the original file.
 * For example, if the original filename is orders.xml, the output
 * filenames are orders1.xml, orders2.xml, and so on.</p>
 *
 * <p>The command line syntax of the CutterTester is:</p>
 *
 * <pre>
 *    java CutterTester &lt;filename> &lt;local-element-type-name>
[&lt;<uri>]
 * </pre>
 *
 * <p>where &lt;local-element-type-name> is the local name of the
 * element type to cut on and &lt;uri> is the (optional) namespace uri
 * of the element type to cut on.</p>
 *
 * <p>CutterTester uses a FileWriter to write to the sub-documents
 * to files. If you want to do something else with the sub-documents,
 * you can modify the code. For example, to store sub-documents in
 * a database, use a StringWriter to build a string from a sub-document,
 * then pass this to the database with an INSERT statement.</p>
 *
 * @version 1.0
 */
public class CutterTester implements ContentHandler
   private String baseName, extension;
   private int    fileNumber;
   private Writer writer;
   /*
    * Create a new CutterTester.
    */
   public CutterTester()
   {
   }
   /*
    * Run the CutterTester from the command line.
    *
    * <p>See the introduction for the command line syntax.</p>
    */
   public static void main (String[] argv)
   {
      try
      {
         CutterTester test = new CutterTester();
         if ((argv.length == 0) || (argv.length > 3))
            throw new Exception("Syntax: java CutterTester <filename>
<local-element-type-name> [<uri>]");
         else
         {
            String uri = (argv.length == 3) ? argv[2] : "";
            test.run(argv[0], uri, argv[1]);
         }
      }
      catch(Exception e)
      {
         e.printStackTrace();
      }
   }
   /*
    * Run the CutterTester.
    *
    * @param filename Name of the file to cut.
    * @param uri Namespace URI of the element type to cut on.
    * @param localName Local name of the element type to cut on.
    */
   public void run(String filename, String uri, String localName)
      throws Exception
   {
      XMLReader xmlReader;
      SAXCutter cutter;
      // Get a new XMLReader and set namespace processing on.
      xmlReader = getXMLReader();
      xmlReader.setFeature("http://xml.org/sax/features/namespaces", true);

xmlReader.setFeature("http://xml.org/sax/features/namespace-prefixes",
true);
      // Get a new SAXCutter, set the ContentHandler to
      // this CutterTester object, and set the cut element type.
      cutter = new SAXCutter(xmlReader);
      cutter.setContentHandler(this);
      cutter.setCutElementType(uri, localName);
      // Set up the global variables.
      baseName = getBaseName(filename);
      extension = getExtension(filename);
      fileNumber = 1;
      // Parse the input file.
      cutter.parse(new InputSource(new FileInputStream(filename)));
   }
   //*******************************************************
   //* ContentHandler methods
   //*******************************************************
   public void startDocument () throws SAXException
   {
      String     filename;
      // Construct the next filename and increment the file number,
      // then open a FileWriter over the file.
      filename = baseName + String.valueOf(fileNumber) + extension;
      fileNumber++;
      try
      {
         writer = new FileWriter(filename);
      }
      catch (IOException io)
      {
         throw new SAXException(io);
      }
   }
   public void endDocument() throws SAXException
   {
      // Close the file.
      try
      {
         writer.close();
      }
      catch (IOException io)
      {
         throw new SAXException(io);
      }
   }
   public void startElement (String uri, String localName, String qName, 
Attributes attrs)
     throws SAXException
   {
      int i;
      try
      {
         // Start the element.
         writer.write('<');
         writer.write(qName);
         // Append the attributes. Note that this includes xmlns 
attributes.
         for (i = 0; i < attrs.getLength(); i++)
         {
            writer.write(' ');
            writer.write(attrs.getQName(i));
            writer.write("=\"");
            appendEscapedString(attrs.getValue(i));
            writer.write('"');
         }
         // Close the element.
         writer.write('>');
      }
      catch (IOException io)
      {
         throw new SAXException(io);
      }
   }
   public void endElement (String uri, String localName, String qName) 
throws SAXException
   {
      try
      {
```

```
      writer.write("</");
      writer.write(qName);
      writer.write('>');
   }
   catch (IOException io)
   {
      throw new SAXException(io);
   }
}
public void characters (char ch[], int start, int length)
  throws SAXException
{
   try
   {
      appendEscapedString(ch, start, length);
   }
   catch (IOException io)
   {
      throw new SAXException(io);
   }
}
public void ignorableWhitespace (char ch[], int start, int length)
   throws SAXException
{
   try
   {
      writer.write(ch, start, length);
   }
   catch (IOException io)
   {
      throw new SAXException(io);
   }
}
public void processingInstruction (String target, String data)
   throws SAXException
{
   try
   {
      writer.write("<?");
      writer.write(target);
      writer.write(' ');
      writer.write(data);
      writer.write("?>");
   }
   catch (IOException io)
   {
      throw new SAXException(io);
   }
}
```

```
public void startPrefixMapping(String prefix, String uri)
   throws SAXException
{
}
public void endPrefixMapping(String prefix)
   throws SAXException
{
}
public void setDocumentLocator (Locator locator)
{
}
public void skippedEntity(String name)
   throws SAXException
{
}
//**********************************************************
//* Private methods
//**********************************************************
private XMLReader getXMLReader()
   throws Exception
{
   // Get an XMLReader. See the documentation for
   // javax.xml.parsers.SAXParserFactory for information
   // about configuring your system to use your XML parser.
   SAXParserFactory factory = SAXParserFactory.newInstance();
   return factory.newSAXParser().getXMLReader();
}
private String getBaseName(String filename)
{
   int period = filename.lastIndexOf('.');
   if (period == -1) return filename;
   return filename.substring(0, period);
}
private String getExtension(String filename)
{
   int period = filename.lastIndexOf('.');
   if (period == -1) return null;
   return filename.substring(period);
}
private void appendEscapedString(String string)
   throws IOException
{
   appendEscapedString(string.toCharArray(), 0, string.length());
}
private void appendEscapedString(char[] chars, int start, int length)
   throws IOException
{
   int    save;
   // This method appends a string to the sub-document StringBuffer.
```

```
      // It replaces &, <, >, ', and " with entity references.
      save = start;
      for (int i = start; i < start + length; i++)
      {
         switch(chars[i])
         {
            case '&':
            case '<':
            case '>':
            case '\'':
            case '"':
               // When we encounter a character that needs to be escaped as
               // an entity, append any characters that haven't been
written,
               // adjust the save point, and append the entity reference.
               if (save < i)
               {
                  writer.write(chars, save, i - save);
               }
               save = i + 1;
               switch(chars[i])
               {
                  case '&':
                     writer.write("&amp;");
                     break;
                  case '<':
                     writer.write("&lt;");
                     break;
                  case '>':
                     writer.write("&gt;");
                     break;
                  case '\'':
                     writer.write("&apos;");
                     break;
                  case '"':
                     writer.write("&quot;");
                     break;
               }
            default:
               break;
         }
      }
      // If there are any characters that haven't yet been appended,
      // append them now.
      if (save < start + length)
      {
         writer.write(chars, save, start + length - save);
      }
   }
```

The TableCutter tool

The TableCutter tool is used to help convert XML documents to a set of files that use the DB2 load format. As input, it accepts a document that conforms to the following DTD, where the contents of the Table element use the DB2 load format. (This document is constructed by transforming the original document with XSLT.)

```
<!ELEMENT Tables (Table+)>
<!ELEMENT Table (#PCDATA)>
<!ATTLIST Table
          Name CDATA #REQUIRED>
```

The TableCutter tool uses the SAXCutter class to extract the contents of each Table element and stores them in separate files.

Example E-7 shows the code for the TableCutter class.

Example: E-7   TableCutter class

```
import java.io.*;
import java.util.*;
import org.xml.sax.*;
import javax.xml.parsers.SAXParserFactory;
/**
 * Cuts a document that uses the load table format into DB2 load files.
 *
 * <p>The TableCutter is designed to cut a document that uses the load
 * table format into separate DB2 load files. The load table format is
 * very simple:
 *
 * <pre>
 *    &lt;!ELEMENT Tables (Table+)>
 *    &lt;!ELEMENT Table (#PCDATA)>
 *    &lt;!ATTLIST Table
 *              Name CDATA #REQUIRED>
 * </pre>
 *
 * The PCDATA in the table element matches the load format used by DB2.
 * For example, a document that contains the data for two tables might
 * look like:
 *
 * <pre>
 * &lt;Tables>
 * <Table Name="Orders">"123",20030708,"543"
 * "456",20030709,"563"&lt;/Table>
 * &lt;Table Name="Items">"123",1,"123",10
 * "123",2,"ab-c",5
 * "456",1,"CD32",100
 * "456",2,"HSA230",8&lt;/Table>&lt;Tables>
 * </pre>
 *
 * <p>The command line syntax of the TableCutter is:</p>
 *
 * <pre>
 *    java TableCutter &lt;filename> &lt;output-directory>
 * </pre>
 *
 * @version 1.0
 */
public class TableCutter implements ContentHandler
   private Writer writer;
   private String outputDirectory;
   /*
    * Create a new TableCutter.
    */
   public TableCutter()
   {
   }
   /*
    * Run the TableCutter from the command line.
    *
    * <p>See the introduction for the command line syntax.</p>
    */
   public static void main (String[] argv)
   {
      try
      {
         TableCutter cutter = new TableCutter();
         if (argv.length != 2)
            throw new Exception("Syntax: java TableCutter <filename> 
<output-directory>");
         else
         {
            cutter.cut(argv[0], argv[1]);
         }
      }
      catch(Exception e)
      {
         e.printStackTrace();
      }
   }
   /*
    * Cut the document.
```

```
     *
     * @param filename Full path of the file to cut.
     * @param outputDirectory Full path of the output directory.
     */
    public void cut(String filename, String outputDirectory)
       throws Exception
    {
       XMLReader xmlReader;
       SAXCutter cutter;
       // Get a new XMLReader and set namespace processing on.
       xmlReader = getXMLReader();
       xmlReader.setFeature("http://xml.org/sax/features/namespaces", true);

 xmlReader.setFeature("http://xml.org/sax/features/namespace-prefixes",
 true);
       // Get a new SAXCutter, set the ContentHandler to
       // this TableCutter object, and set the cut element type to Table.
       cutter = new SAXCutter(xmlReader);
       cutter.setContentHandler(this);
       cutter.setCutElementType("", "Table");
       // Set the global variables.
       this.outputDirectory = outputDirectory;
       writer = null;
       // Parse the input file.
       cutter.parse(new InputSource(new FileInputStream(filename)));
    }
    //************************************************************
    //* ContentHandler methods
    //************************************************************
    public void startDocument () throws SAXException
    {
    }
    public void endDocument() throws SAXException
    {
    }
    public void startElement (String uri, String localName, String qName,
 Attributes attrs)
      throws SAXException
    {
       String tableName, filename;
       // If we are not on the Table element, just return.
       if ((uri.length() != 0) || (!localName.equals("Table"))) return;
       // Create a new output file.
       try
       {
          tableName = attrs.getValue("Name");
          filename = outputDirectory + tableName + ".del";
          writer = new FileWriter(filename);
       }
       catch (IOException io)
       {
          throw new SAXException(io);
       }
    }
    public void endElement (String uri, String localName, String qName)
 throws SAXException
    {
       // If we are not on the Table element, just return.
       if ((uri.length() != 0) || (!localName.equals("Table"))) return;
       // Close the file.
       try
       {
          writer.close();
       }
       catch (IOException io)
       {
          throw new SAXException(io);
       }
       // Set the writer to null so that we only write to the file when
       // we are inside the Table element.
       writer = null;
    }
    public void characters (char ch[], int start, int length)
      throws SAXException
    {
       try
       {
          if (writer != null) writer.write(ch, start, length);
       }
       catch (IOException io)
       {
          throw new SAXException(io);
       }
    }
    public void ignorableWhitespace (char ch[], int start, int length)
       throws SAXException
    {
       try
       {
          if (writer != null) writer.write(ch, start, length);
       }
       catch (IOException io)
       {
          throw new SAXException(io);
       }
    }
    public void processingInstruction (String target, String data)
       throws SAXException
```

```
{
}
public void startPrefixMapping(String prefix, String uri)
   throws SAXException
{
}
public void endPrefixMapping(String prefix)
   throws SAXException
{
}
public void setDocumentLocator (Locator locator)
{
}
public void skippedEntity(String name)
   throws SAXException
{
}
//**********************************************************
//* Private methods
//**********************************************************
private XMLReader getXMLReader()
   throws Exception
{
   // Get an XMLReader. See the documentation for
   // javax.xml.parsers.SAXParserFactory for information
   // about configuring your system to use your XML parser.
   SAXParserFactory factory = SAXParserFactory.newInstance();
   return factory.newSAXParser().getXMLReader();
}
```

F

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/SG246994
```

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, SG246994.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

File name	*Description*
sg246994.zip	Sample code for following the samples throughout the book

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space	3 GB
Operating System	Windows 2000 or Windows NT
Processor	700 MHz or better
Memory	512 MB, recommended 784 MB

How to use the Web material

Unzip the contents of the Web material sg2469947code.zip file onto your hard drive. This creates the folder structure c:\SG246994\Scenario:

`Scenario` Files required to go through the scenario (Chapter 14, “Scenario implementation using DB2 functionality” on page 401 and Chapter 13, “Worked scenario” on page 393)

`...\testharness` Files generated by Application Developer for testing `retrieveXML` and `storeXML` (Chapter 12, “XML and database tools in Application Developer” on page 337)

`...\xmltest` Files generated by Application Developer for testing `retrieveXML` and `storeXML` (Chapter 15, “Scenario with Application Developer tools” on page 453)

`...\wsad` Files required for DB2 Web services (Chapter 16, “DB2 Web services and XML with Application Developer” on page 475)

`...\SampleTools` Contains the code for the tools used in Chapter 7, “Bulk processing of XML documents” on page 191, and described in more detail in Appendix E, “Sample XML Tools” on page 619.

`...\WORFTest` Sample code for manual creation of a WORF Web application (Chapter 10, “Web services in DB2” on page 269)

...\udf	Sample UDF functions and invocations
...\worf-soap-code	JAR files required to install WORF and DB2 Web services in WebSphere Application Server.

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 on ordering these publications, see “How to get IBM Redbooks” on page 660. Note that some of the documents referenced here may be available in softcopy only.

- *Data Federation with IBM DB2 Information Integrator V8.1*, SG24-7052

Other publications

These publications are also relevant as further information sources:

- *DB2 XML Extender Administration and Programming V8*, SC27-1234
- *DB2 UDB for z/OS V8 XML Extender Administration and Programming*,SC18-7431
- *DB2 UDB V8 Data Movement Utilities Guide and Reference*, SC09-4830
- *DB2 Information Integrator V8 Developer’s Guide,* SC18-7359

Online resources

These Web sites and URLs are also relevant as further information sources:

- DB2 XML Extender Web site

  `http://www-306.ibm.com/software/data/db2/extenders/xmlext/index.html`

- W3C Web site

  `http://www.w3c.org`

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

Index

IBM

Redbooks

XML for DB2 Information Integration

(1.0” spine)
0.875”<->1.498”
460 <-> 788 pages

XML for DB2 Information Integration

Redbooks

Marrying XML documents and databases

Scenarios of XML usage

Using IBM WebSphere Studio Application Developer to build Web Services and XML

In many organizations, relational databases are the backbone for data storage and retrieval. Over the last couple of years, XML has become the de facto standard to exchange information between organizations, as well as between departments or applications within the same organization. Since data tends to live in databases, it needs to be converted from a relational format into an XML format when involved in those data exchanges, as well as converted (back) from XML into a relational format for storage, or for handling by other applications.

How can we achieve this? This IBM Redbook describes how to design the mapping between XML and relational data, and vice versa, to enable a flexible exchange of information.

IBM provides a number of products to help you bridge the gap between XML and its relational database, DB2. The DB2 engine itself provides support to generate XML fragments from relational data through the use of SQL/XML built-in functions. DB2 also provides the DB2 XML Extender. It allows you to perform XML composition, like SQL/XML, but also provides functionality to decompose XML documents and store XML documents intact inside the database. XML Extender also provides a set of transformation and validation functions. Another option to work with XML is to use the XML wrapper, a part of the set of non-relational wrappers of DB2 Information Integrator. This redbook also looks at the IBM tools available to assist you when dealing with XML, specifically WebSphere Application Developer and DB2 Control Center.

To add a more practical angle, these functions and products are illustrated through the development of a simple application.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-6994-00 ISBN 0738490032