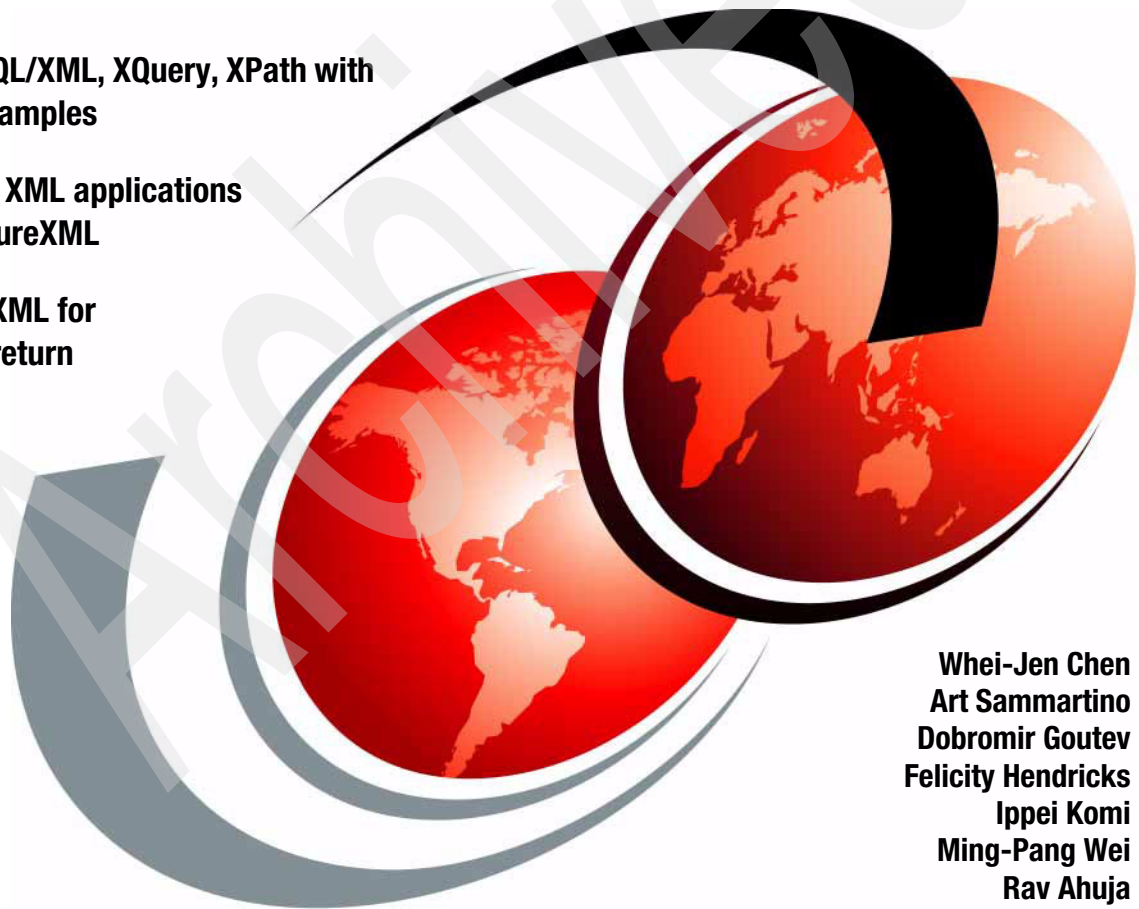


DB2 9 pureXML Guide

Learning SQL/XML, XQuery, XPath with
working examples

Developing XML applications
with DB2 pureXML

Managing XML for
maximum return



Whei-Jen Chen
Art Sammartino
Dobromir Goutev
Felicity Hendricks
Ippei Komi
Ming-Pang Wei
Rav Ahuja



International Technical Support Organization

DB2 9 pureXML Guide

January 2007

Archived

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

First Edition (January 2007)

This edition applies to DB2 9 for Linux, UNIX, and Windows.

© Copyright International Business Machines Corporation 2007. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	ix
Trademarks	x
Preface	xi
The team that wrote this redbook	xii
Acknowledgements	xiii
Become a published author	xiv
Comments welcome	xv
Chapter 1. Introducing DB2 9: pureXML	1
1.1 Growing importance of XML data	2
1.1.1 Growth of XML	2
1.1.2 The value of XML data	4
1.2 pureXML overview	4
1.2.1 Traditional methods for managing XML data	5
1.2.2 XML data management with DB2 9	6
1.2.3 Setting up databases for XML	7
1.2.4 XML optimized storage and XML data type	7
1.2.5 Getting XML data into the database	8
1.2.6 Querying XML data	8
1.2.7 Query optimization and indexes for XML	10
1.2.8 XML schema repository and validation	10
1.2.9 Full text search for XML	11
1.2.10 Annotated schema decomposition	11
1.2.11 Application development support	11
1.2.12 Tools and utilities	12
1.2.13 Benefits of DB2 pureXML technology	13
1.3 pureXML usage scenarios	13
1.3.1 Integration of diverse data sources	14
1.3.2 Forms and their processing	15
1.3.3 Document storage and querying	16
1.3.4 XML for transactions	17
1.3.5 Syndication and XML feeds	18
1.3.6 XML as a better data model	19
1.4 Summary	19
1.5 References	20
Chapter 2. Sample scenario description	21
2.1 Business requirements	22

2.1.1 Data modeling	23
2.2 Application description	25
2.2.1 Loan application	26
2.2.2 Loan processing	31
2.2.3 Loan management	34
2.3 Application setup	36
Chapter 3. XML database design	39
3.1 Architecture overview	40
3.2 Logical database design	43
3.2.1 XML data type	43
3.2.2 Relational structure versus XML structure	44
3.2.3 XML indexes	47
3.2.4 Views.	48
3.2.5 XML schema	49
3.2.6 XML schema design	51
3.2.7 Industry standards and XML schemas	56
3.2.8 XML data validation.	60
3.3 Physical database design	66
3.4 Creating a database	70
Chapter 4. Working with XML	73
4.1 XPath.	74
4.1.1 XQuery/XPath data model	75
4.1.2 Location paths.	78
4.1.3 Using location paths to retrieve nodes of an XML document	80
4.1.4 Predicates	86
4.2 XQuery	87
4.2.1 Types, expressions, and functions	88
4.2.2 FLWOR and selecting XML data.	103
4.2.3 Updating XML data	120
4.3 XQuery and SQL/XML.	126
4.3.1 XQuery with embedded SQL	126
4.3.2 SQL/XML.	127
4.3.3 When to use what	133
4.4 When and how to use namespaces	136
4.5 Getting XML data in and out of database	142
4.6 XML full-text search.	154
4.6.1 DB2 Net Search Extender.	154
4.6.2 Preparing the instance for text search	155
4.6.3 Full-text searching using DB2 NSE.	159
4.6.4 Taking advantage of Net Search Extender text search features.	162
4.6.5 Full-text search considerations	164

4.6.6 The NSE document model	167
Chapter 5. Managing XML data	173
5.1 XML indexes	174
5.1.1 XML index types	174
5.1.2 Creating XML indexes	176
5.1.3 How to look up information for XML indexes	181
5.1.4 Access plan	186
5.1.5 Best practices	195
5.2 Schema management	198
5.2.1 XML Schema Repository	199
5.2.2 XML schema registration/dropping	199
5.2.3 Querying XSR	203
5.2.4 XSR support on the Control Center	205
5.2.5 Schema evolution	206
5.3 IMPORT, EXPORT, and RUNSTATS	208
5.3.1 IMPORT	208
5.3.2 EXPORT	221
5.3.3 RUNSTATS	229
5.4 XML data security	233
5.4.1 LBAC	233
5.4.2 Row and column-level access control	234
5.4.3 Node-level access control	241
Chapter 6. Application development	249
6.1 The database application development environment	250
6.2 Application development tools	250
6.2.1 Developer Workbench	254
6.2.2 Developer Workbench: Visual Query Builder overview	256
6.3 Accessing pureXML from application overview	268
6.3.1 Application programming language support for XML	268
6.3.2 Considerations when updating and inserting XML data	269
6.3.3 Considerations when retrieving XML data	275
6.4 DB2 application development with CLI and ODBC	278
6.4.1 Setting up the CLI environment	278
6.4.2 Building CLI applications	279
6.4.3 XML data handling in CLI applications	282
6.4.4 Embedded SQL applications: overview	289
6.5 Building applications in C or C++	290
6.5.1 Building C/C++ applications with the sample build script	291
6.5.2 Declaring XML host variables	294
6.5.3 Referencing XML host variables	296
6.5.4 Declaring large object type host variables	297

6.5.5	Referencing LOB type host variables	299
6.5.6	Executing XQuery expressions in embedded SQL applications	299
6.6	Java application programming	302
6.6.1	Setting up the DB2 JDBC and SQLJ development environment	302
6.6.2	Building JDBC applications	303
6.6.3	Building SQLJ applications	309
6.7	Building DB2 applications with PHP	315
6.7.1	Setting up the PHP application development environment	316
6.7.2	Introduction to PHP application development for DB2	318
6.8	The DB2 .NET environment	321
6.8.1	Building sample applications for the DB2 .NET data provider	321
6.8.2	XML support in Visual Studio.NET: overview	322
6.8.3	XML data type support in Visual Studio .NET	322
6.8.4	XQuery support in Visual Studio.NET	334
6.9	XML and stored procedures	338
6.9.1	XML and XQuery support in SQL procedures	339
6.9.2	XML support in external routines	343
6.9.3	XML Schema Repository object registration	348
6.10	Web services	349
6.10.1	Components of Web Services	351
6.10.2	Web services in DB2 9	353
Appendix A. Sample data		361
A.1	Creating XMLLoan database	362
A.1.1	Creating database	362
A.1.2	Creating tables	362
A.2	contactInfo.xsd	366
A.3	Sample XML data	368
Appendix B. Additional material		373
	Locating the Web material	373
	Using the Web material	374
	System requirements for downloading the Web material	374
	How to use the Web material	374
Abbreviations and acronyms		375
Related publications		377
	IBM Redbooks	377
	Other publications	377
	Online resources	379
	How to get IBM Redbooks	380
	Help from IBM	380

Index 383

Archived

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

Trademarks

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

DB2®
DB2 Universal Database™
developerWorks®
IBM®
ibm.com®
IMS™

Informix®
iSeries™
pureXML™
Rational®
Redbooks™
Redbooks (logo) ™

WebSphere®
Workplace™
Workplace Forms™
z/OS®

The following terms are trademarks of other companies:

Oracle, JD Edwards, PeopleSoft, and Siebel are registered trademarks of Oracle Corporation and/or its affiliates.

Snapshot, and the Network Appliance logo are trademarks or registered trademarks of Network Appliance, Inc. in the U.S. and other countries.

eXchange, Java, JDBC, JDK, JVM, J2SE, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Expression, Microsoft, Visual Basic, Visual Studio, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

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

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

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

Preface

This IBM® Redbook, intended for IT managers, IT architects, DBAs, application developers, and other data server professionals, offers a broad understanding of the DB2® 9 feature pureXML™. The book is organized as follows:

- ▶ Chapter 1, “Introducing DB2 9: pureXML” on page 1, explores the importance of XML data and the necessity for managing it as a strategic business asset. We give you an overview of the DB2 9 pureXML technology and its features, and illustrate examples and scenarios for utilizing DB2 9 and pureXML.
- ▶ Chapter 2, “Sample scenario description” on page 21, introduces the sample online unsecured loan application, XMLoan. We cover the business requirements, data modeling, application descriptions, and application setup.
- ▶ Chapter 3, “XML database design” on page 39, provides information about the hybrid database design. We describe the DB2 9 database architecture, as well as logical and physical database design.
- ▶ Chapter 4, “Working with XML” on page 73, discusses how to work with XML. The topics covered include XPath, XQuery, SQL/XML, when and how to use namespaces, getting XML data in and out of the database, and XML full-text search.
- ▶ Chapter 5, “Managing XML data” on page 173, explains how to manage XML data stored in XML columns. We introduce the pureXML index features and schema management, and illustrate how to move data, including XML documents, in and out of a table using DB2 9 IMPORT and EXPORT utilities. We also show how RUNSTATS work with pureXML features. Finally, we describe some security solutions that correspond to pureXML features.
- ▶ Chapter 6, “Application development” on page 249, covers various aspects of application development using DB2. The information contained in this chapter features topics and examples that are specific to application development with XML. The subjects covered include database application development environment and tools, how to access pureXML from within an application, XML and stored procedures, and Web Services.

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, Poughkeepsie Center.

Whei-Jen Chen is a Project Leader at the International Technical Support Organization, San Jose Center. She has extensive experience in application development, database design and modeling, and DB2 system administration. Whei-Jen is an IBM Certified Solutions Expert in Database Administration and Application Development as well as an IBM Certified IT Specialist.

Art Sammartino is a certified consulting I/T specialist who has been with the IBM Database Migration Team (SMPO) since joining IBM seven years ago. He is responsible for assisting customers who are considering a database migration from competitive RDBMS products (Oracle®, Sybase, or Microsoft® SQL Server) to IBM DB2. In addition to demonstrating and performing database and application conversion, his experience includes supporting clients with application development environment and setup concerns, as well as migration tool installation and education. He is certified as both an IBM Database Administrator and an IBM Application Developer.

Dobromir Goutev is a Release Manager for vivatel in Sofia, Bulgaria. He has more than 15 years of experience in application design, development, teaching and consulting. His areas of expertise include relational databases, object-oriented analysis and design, and application architecture. He holds a Master of Science in Computer Science from Sofia University in Bulgaria.

Felicity Hendricks is an advisory software engineer working at Silicon Valley Laboratory. She is entering her seventh year at IBM. She is certified in IBM DB2 Universal Database™ and is currently a member of WebSphere® Federation Server service team. Her areas of expertise include DB2 Universal Database, WebSphere Information Integrator, and DB2 XML Extender. Felicity holds a Bachelor of Science degree in Computer Science from CSU Chico.

Ippei Komi is an IT specialist working at IBM Japan. Ippei has more than nine years of experience in IT as an application developer. His areas of expertise include relational databases, object-oriented analysis and design, and application architecture.

Ming-Pang Wei joined IBM in 2001 and has held various roles within IBM Canada and IBM Australia. He is an IBM Certified Solutions Expert in Database Administration and Application Development. Currently, he is working as an Application Development Specialist within the IBM DB2 Advanced Support Services team in IBM Toronto Lab where he helps customers and vendors get the best out of their applications and DB2.

Rav Ahuja is a worldwide DB2 program manager based at the IBM Toronto Lab. He has been working with DB2 for Linux®, UNIX®, and Windows® since version 1 and has held various roles in DB2 development, technical support, marketing, and product strategy. He works with customers and partners around the globe helping them build and benefit from DB2 and services-based solutions. Rav is a frequent contributor to DB2 papers, articles, and books. He holds a Computer Engineering degree from McGill University and MBA from University of Western Ontario.



Left to right: Ming-Pang, Dobromir, Felicity, Ippei, Whei-Jen, Rav, and Art

Acknowledgements

The authors express their deep gratitude for the help they received from **Susan Malaika** from the IBM Silicon Valley Laboratory.

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

Grant Hutchison
Budi Surjanto
Prashant Juttukonda
Ro Omro
Samir Kapoor
IBM Toronto Laboratory

Matthias Nicola
Cindy Saracco
Bert Van Der Linden
Christina Lee
Mayank Pradhan
Ted Wasserman
Henrik Loeser
IBM Silicon Valley Laboratory

Barry Faust
IBM Software Migration Project Office

Brian Williams
IBM Software Group

Denise Pirro
IBM Sales and Distribution

Many thanks to our support staff for their help in the preparation of this book:
Emma Jacobs, Sangam Racherla, Deanna Polm, and Yvonne Lyon.
International Technical Support Organization, San Jose Center

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 have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

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 e-mail to:

redbooks@us.ibm.com

- ▶ Mail your comments to:

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

Introducing DB2 9: pureXML

IBM DB2 9 (previously code named *Viper*) for Linux, UNIX, and Microsoft Windows (LUW) marks a new stage in the evolution of data servers. IBM has continually led the data management industry with the release of innovative technology, starting with Information Management System (IMS™) in the 1960s, invention of relational database model and Structured Query Language (SQL) in the 1970s, DB2 for the mainframe in 1980s, and now with DB2 9, a new generation data server with revolutionary pureXML technology.

The technology in DB2 9 fundamentally transforms the way XML information is managed for maximum return while seamlessly integrating XML with relational data. It takes data services to new levels by lowering costs, delivering greater agility, and improving business insight, making DB2 9 an essential ingredient of the information-as-a-service infrastructure.

In this chapter, we:¹

- ▶ Explore the importance of XML data and the necessity for managing it as a strategic business asset.
- ▶ Provide an overview of the DB2 9 pureXML technology and its features.
- ▶ Illustrate examples and scenarios for utilizing DB2 9 and pureXML.

¹ Portions of this chapter are excerpted from the papers listed in 1.5, “References” on page 20.

1.1 Growing importance of XML data

XML technology has become pervasive in virtually all industries and sectors, owing to its versatility and neutrality for exchanging data among diverse devices, applications, and systems from different vendors. These qualities of XML along with its easy to understand self-describing nature, ability to handle structured, semi-structured and unstructured data, and support for Unicode have made XML a universal standard for data interchange.

1.1.1 Growth of XML

Nearly every company today comes across XML in some form. The amount of XML data that organizations have to deal with is growing at a rapid rate. In fact, it is estimated that the volume of XML data is growing twice as fast as that of the traditional data that typically resides in relational databases. Factors fueling the growth of XML data include:

- ▶ XML-based industry and data standards
- ▶ Service-oriented architectures (SOA) and Web services
- ▶ Web 2.0 technologies such as XML feeds and syndication services

XML-based industry standards

Almost every industry has multiple standards based on XML and there are numerous cross-industry XML standards as well. A few examples of XML-based industry standards are listed here:

- ▶ ACORD - XML for the Insurance Industry:
<http://www.acord.org/>
- ▶ FPML - Financial Product:
<http://www.fpml.org/>
- ▶ HL7 - Health Care:
<http://www.hl7.org/>
- ▶ IFX - Interactive Financial Exchange:
<http://www.ifxforum.org/>
- ▶ IXRetail - Standard for Retail operation:
<http://www.nrf-arts.org/>

- ▶ XBRL - Business Reporting / Accounting:
<http://www.xbrl.org/>
- ▶ NewsML - News / Publication:
<http://www.newsml.org/>

These standards facilitate purposes such as the exchange of information between the various players within these industries and their value chain members, data definitions for ongoing operations, and document specifications. More and more companies are adopting such XML standards or are being compelled to adopt them in order to stay competitive, improve efficiencies, communicate with their trading partners or suppliers, or just to perform everyday tasks.

SOA and Web services

Services-based frameworks and deployments are growing in popularity, owing to their ability to integrate systems, permit reuse of resources, and respond quickly to changing market conditions, allowing companies to save money and improve competitiveness. In services-based architectures, consumers and service providers exchange information using messages. These messages are invariably encapsulated as XML. As such, XML can provide the plumbing in SOA environments as illustrated in Figure 1-1. Therefore the drive towards information as a service and rapid adoption of SOA environments is also stimulating the growth of XML.

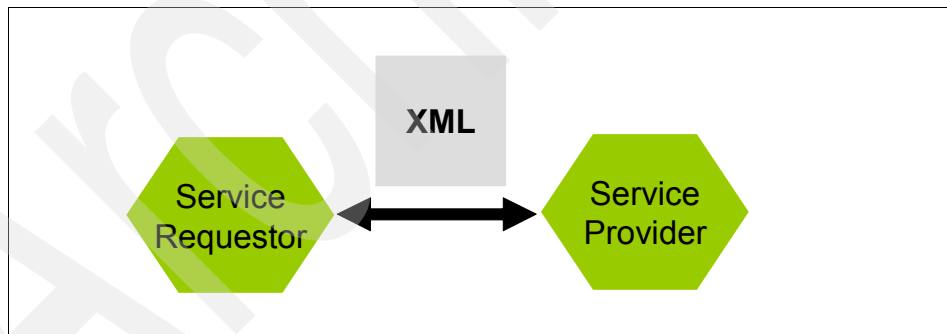


Figure 1-1 XML: The foundation for Web services

Web 2.0 technologies

Syndication is considered to be the heartbeat of Web 2.0, the next generation of the Internet. Atom and RSS feeds can be found abundantly on the Web, allowing the user to subscribe to them and be kept up-to-date about all kinds of Web content changes, such as news stories, articles, wikis, audio and video files.

Content for these feeds is rendered as XML files and can contain links, summaries, full articles, and even attached multimedia files such as podcasts. Syndication and Web feeds are transforming the Web as we know it. New business models are emerging around these technologies. As a consequence, XML data now exists not only in companies adopting XML industry standards, or enterprises implementing SOAs, but also on virtually every Web-connected desktop.

1.1.2 The value of XML data

As a result of XML industry standards becoming more prevalent, the drive towards SOA environments, and rapid adoption of syndication technologies, more and more XML data is being generated every day as Web feeds, purchase orders, transaction records, messages in SOA environments, financial trades, insurance applications, and other industry-specific and cross-industry data. That is, XML data and documents are becoming an important business asset containing valuable information (such as customer details, transaction data, order records, and operational documents).

The growth and pervasiveness of XML assets presents challenges and opportunities for companies. When XML data is harnessed, and the value of the information it contains is unlocked, it can translate into opportunities for organizations to streamline operations, derive insight, and become agile.

On the other hand, as XML data becomes more critical to the operations of an enterprise, it presents challenges in that XML data must be secured, maintained, searched, and shared. Depending on its use, XML data might also have to be updated, audited, and integrated with traditional data. All of these tasks must be done with the reliability, availability, and scalability afforded to traditional data assets.

That is, in order to unleash the potential of XML data, it requires storage and management services similar to what enterprise-class relational database management systems such as DB2 have been providing for relational data. In the next section, we explore how this type of data management maturity can be afforded to XML.

1.2 pureXML overview

pureXML is a new DB2 9 feature that provides the capability of storing XML data natively in a database table. This section introduces the pureXML functions and features.

1.2.1 Traditional methods for managing XML data

Until DB2 9, management of XML data has involved these common approaches:

- ▶ Storing XML documents on file systems
- ▶ Stuffing XML data into large objects (LOBs) in relational databases
- ▶ Shredding XML data into multiple relational columns and tables
- ▶ Isolating data into XML-only database systems

Frequently, these obvious choices for managing and sharing XML data do not meet performance requirements. File systems are fine for simple tasks, but they do not scale well when you have hundreds or thousands of documents. Concurrency, recovery, security, and usability issues become unmanageable. Using database management systems (DBMS) alleviate some of these constraints of file systems, but up until now the DBMS offerings have had their own limitations for managing XML data. Two common approaches for using relational database management systems (DBMSs) are called stuffing and shredding.

- ▶ *Stuffing* involves storing an XML document as a whole into a single VARCHAR or large object (CLOB or BLOB) column within a relational database. This approach works well as long as all you have to do is store and retrieve the XML documents in their entirety. However, if you have to query the contents of XML documents or retrieve fragments or specific elements/attributes/sub-trees, it involves scanning through each document at run-time, which can be highly unwieldy due to performance overhead.
- ▶ *Shredding* or decomposing involves mapping XML data to and from relations columns and tables. To store XML into the database, the XML document is shredded into its various pieces (elements and attributes) that are stored in separate columns, a process that involves some overhead. Complexity of the XML data and normalization rules might cause XML documents to span hundreds of columns in dozens of tables. Similarly, to reconstruct the document, all of these columns and tables must be accessed using complex queries and multitable joins that introduce unnecessary complexity. It might even become impossible to reconstruct some documents or preserve the original fidelity of data, such as for digital signatures. Furthermore, this approach also introduces rigidity of relational data models into the flexible nature of XML data formats.
- ▶ During recent years, a number of specialized DBMSs for XML data have been introduced that are aware of the XML data structures and allow more efficient management of XML data. However, most of these XML DBMSs are relatively new and introduce a largely unproven environment into an IT infrastructure, raising concerns about integration with traditional data, staff skills, and long-range viability.

1.2.2 XML data management with DB2 9

With the release of DB2 9, IBM is leading the way to a new era in data management. DB2 9 embodies technology that provides pure XML services. This pureXML technology is not just for data server external interfaces — rather, pureXML extends to the very core of the DB2 engine. The XML and relational services in DB2 9 are tightly integrated, thereby offering the industry's first pureXML and relational hybrid data server. Figure 1-2 illustrates the hybrid database.

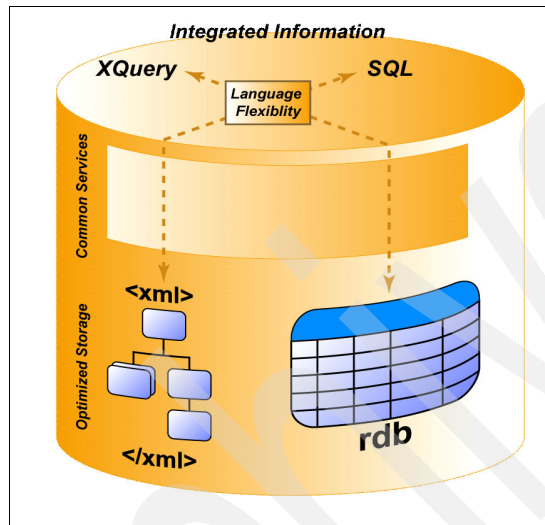


Figure 1-2 pureXML and relational hybrid database

The pureXML technology in DB2 9 includes the following capabilities:

- ▶ pureXML data type and storage techniques for efficient management of hierarchical structures common in XML documents.
- ▶ pureXML indexing technology to speed searches of subsets of XML documents.
- ▶ New query language support (for XQuery and SQL/XML) based on industry standards and new query optimization techniques.
- ▶ Industry-leading support for managing, validating, and evolving XML schemes.
- ▶ Comprehensive administrative capabilities, including extensions to popular database utilities.
- ▶ Integration with popular application programming interfaces (APIs) and development environments.

- ▶ XML shredding and publishing facilities for working with existing relational models.
- ▶ Enterprise proven reliability, availability, scalability, performance, security, and maturity that you have come to expect from DB2

In the next few sections we take a closer look at some of these capabilities and how to use them.

1.2.3 Setting up databases for XML

Creating a database for storing XML data in DB2 9 is no different from creating a database for relational data, because both types of data can be stored and accessed simultaneously in the same database. However, because XML data is typically represented as Unicode, in DB2 V9.1, the database must be created using a Unicode setting:

```
CREATE DATABASE test USING CODESET utf-8 TERRITORY US
```

1.2.4 XML optimized storage and XML data type

XML data is inherently hierarchical, and can be naturally represented in a tree form using nodes having parent, child, and sibling relationships. With DB2 9, collections of XML documents are captured in tables that contain one or more columns based on a new XML data type.

Unlike XML data types offered by some of the other RDBMs (which, under the covers, transform XML data and store using relational constructs), DB2 offers an XML data type that stores parsed XML documents and fragments with node-level granularity preserving the hierarchical structures of the original XML data. By not force fitting XML data into relational data types and not incurring the associated overhead, the pure XML data type in DB2 provides efficient access to XML data, or to specific portions of it.

You can create a table with both relational and XML data types:

```
CREATE TABLE orders (orderid INT, orderinfo XML)
```

You do not have to put any relational columns in the table, or if you prefer, you can have multiple columns of XML type along with multiple ones of relational types:

```
CREATE TABLE o2 (oid INT, otype CHAR(2), ocust XML, oinfo XML)
```

1.2.5 Getting XML data into the database

XML data can be easily entered into a table created with XML data types using the INSERT statement. Or, if you have to populate the database with a large number of XML documents, you can invoke the IMPORT command. The same as for relational data, these operations can be performed with user applications, DB2 graphical tools, or a DB2 command line interface. Example 1-1 shows an example of inserting an XML document into ORDERS table.

Example 1-1 Insert XML document

```
INSERT INTO orders (orderid, orderinfo) VALUES (5,  
  '<order>  
    <orderdate>2006-07-07</orderdate>  
    <customer id="8">  
      <name>XYZ</name>  
      <zip>12345</zip>  
    </customer>  
    <shipnote>Fragile Contents</shipnote>  
  </order>'  
);
```

1.2.6 Querying XML data

DB2 9 supports query languages familiar to both relational and XML programmers. That is, you access data using either SQL or XQuery, which is a new query language that supports navigational (or path-based) expressions. In fact, your applications can employ freely statements from both query languages, and a single query statement can actually incorporate both SQL and XQuery. The results of queries can return data in relational and XML formats, or a combination of both.

Here, we look at a few examples using the table ORDERS and the document inserted into this table in Example 1-1. Queries in Example 1-2 use SQL to access both relational and XML data:

Example 1-2 Using SQL to access both relational and XML data

```
SELECT * FROM orders WHERE orderid=5  
SELECT orderinfo FROM orders
```

Consider the XQuery shown in Example 1-3. This XQuery returns the entire XML document inserted in Example 1-1.

Example 1-3 Using XQuery to access XML data

```
XQUERY db2-fn:xmlcolumn('ORDERS.ORDERINFO');
```

The XQuery in Example 1-4 retrieves the orderdate from XML documents in the orders table.

Example 1-4 Using XQuery to access part of XML document

```
XQUERY
  for $d in db2-fn:xmlcolumn('ORDERS.ORDERINFO')/order/orderdate
  return $d;
```

Result:
<orderdate>
2006-07-07
</orderdate>

Finally, Example 1-5 shows code samples that combine SQL and XQuery. The first is an SQL/XML statement, and the next one embeds SQL within XQuery.

Example 1-5 Combining SQL and XQuery

```
-- retrieve the orderid of orders for a specific customer 'XYZ'
SELECT orderid FROM orders
WHERE xmlexists('$o[order/customer/name="XYZ"]'
  PASSING orderinfo AS "o");

-- retrieve info for orders matching specified criteria
XQUERY db2-fn:sqlquery(
  "SELECT orderinfo FROM orders WHERE orderid > 3"
)/order/customer[zip = "12345"];
```

We discuss in detail XQuery, SQL, and SQL/XML in Chapter 4, “Working with XML” on page 73.

1.2.7 Query optimization and indexes for XML

DB2 has two query language parsers: one for XQuery and one for SQL. Both parsers interoperate and generate a common, language-neutral, internal representation of queries. This means that queries written in either language are afforded the full benefits of DB2's cost-based query-optimization techniques, which include efficient rewriting of query operators and selection of a low-cost data access plan. In addition, DB2 can leverage new query and join operators, as well as new index processing techniques, to provide strong runtime performance for queries involving XML documents.

Along with new hierarchical storage management support for XML, DB2 9 features new indexing technology to speed up searches involving XML data. Like their relational counterparts, these new XML indexes are created with a familiar SQL DDL statement: CREATE INDEX. However, in addition to specifying the target column to index, users also specify an *xmlpattern*, essentially an XPath expression without predicates, to identify the subset of the XML document of interest. Example 1-6 shows a CREATE INDEX statement.

Example 1-6 Creating XML index

```
CREATE INDEX odindex ON orders(orderinfo) GENERATE  
KEY USING XMLPATTERN '/order/orderdate' as SQL DATE;
```

1.2.8 XML schema repository and validation

DB2 provides an XML schema repository where the schemas of XML documents can be registered for validation purposes. If you are not familiar with XML schemas, they are simply well-formed XML documents that dictate the structure and content of other documents. For example, XML schemas specify which elements are valid, in what order these elements should appear in a document, and which XML data types are associated with each element.

In keeping with the flexibility of XML, DB2 9 offers lots of options for validating XML data. Depending on requirements, you can choose not to validate XML data before it is inserted in the database, thereby allowing a single XML column to contain documents with different schemas.

However, if business requirements dictate conformance to specific schemas, you can validate documents using schemas registered with DB2. Furthermore, you can have documents in the same XML column validated with different versions of schemas. Therefore, changing a schema for evolving business requirements (also known as schema evolution) can be quickly accommodated. Using the INSERT statement in Example 1-7, the XML document will be validated during insertion.

```
INSERT INTO orders(orderinfo) VALUES XMLVALIDATE(?  
    ACCORDING TO XMLSCHEMA ID order.ordschema)
```

1.2.9 Full text search for XML

Full-text search is a common operation in document-centric XML applications. DB2's existing text search capabilities (Net Search Extender) have been enhanced to work with the new XML column type. Full-text indexes with awareness of XML document structures can be defined on any XML column in DB2 9. The documents in an XML column can be fully indexed or partially indexed, for example if it is known in advance that only a certain part of each document will be subject to full-text search, such as a *description* or *comment* element. Correspondingly, text search expressions can be applied to specific paths in a document.

1.2.10 Annotated schema decomposition

Even though the DB2 9 store can insert and query any XML document, there are cases where it still makes sense to shred XML documents into relational rows and columns. In certain usage scenarios, XML is only used to transport data to the database but the XML structure is irrelevant after the data is integrated with existing relational data. Shredding can also be required because many existing tools for data mining and business intelligence only work on the relational format of the data. DB2 offers an improved decomposition facility that automates the process of mapping XML data into relational tables.

1.2.11 Application development support

As an integral part of pureXML technology, DB2 9 contains rich support for developing XML centric and hybrid applications to simplify coding, reduce development time, and improve application change agility. XML development support in DB2 9 includes:

- ▶ Support for common programming languages and application interfaces
 - Languages: C/C++, Java™, C#, Visual Basic®, Cobol, PHP
 - Interfaces: JDBC™, CLI / ODBC, .NET, Embedded SQL, SQLJ
- ▶ Support for querying data using XQuery and SQL/XML (or both together)
- ▶ Comprehensive XML capabilities in Developer Workbench (including XQuery builder)
- ▶ Tight integration with Visual Studio® and .NET
- ▶ New code samples and DB2 SAMPLE database enhancements

Take, for instance, DB2 support for XML in JDBC; the Universal DB2 driver for JDBC has been enhanced for XML data. XML data for query results and input and output parameters can be bound using Java data types such as strings, byte arrays, and streams. Because JDBC 3.0 currently does not define a native XML data type, DB2 provides an extension XML type known as `com.ibm.db2.DB2Xml`.

The DB2Xml extension has a number of very useful methods that makes working with XML data easy. In Example 1-8, a column is retrieved as a DB2 XML object. Then the `getDB2String` method returns the serialized representation of the XML value (without XML declaration) as a string object. The method `getDB2XMLBinaryStream` ("UTF-16") then returns a binary stream with the XML value encoded in UTF-16, including a matching XML declaration.

Example 1-8 Using DB2 provided JDBC extension to access XML data

```
com.ibm.db2.jcc.DB2Xml xml1 =  
    (com.ibm.db2.jcc.DB2Xml) rs.getObject ("xml_stuff");  
String s = xml1.getDB2String();  
InputStream is = xml1.getDB2XMLBinaryStream("UTF-16");
```

1.2.12 Tools and utilities

Most of the standard DB2 tools and utilities have been enhanced with integrated support for pureXML technology, as follows:

- ▶ Import/Export: Getting data XML documents in and out of the database
- ▶ Backup/Restore: XML data is backed up and restored alongside relational
- ▶ Runstats: Updating statistics for query optimization
- ▶ High Availability and Disaster Recovery (HADR): Using secondary or off-site systems
- ▶ Advanced Security: Including fine-grained, label-based access control
- ▶ Stored procedures: Including ability to pass XML documents as parameters

A number of graphical tools and interfaces in DB2 have also been enhanced for XML. For instance, in addition to navigating and viewing XML objects using Control Center, you can visually specify XML data types while creating tables, build XML index expressions, execute queries involving XML, and view query access paths and index usage with Visual Explain. You can use drag-and-drop features of the XQuery builder (part of DB2 Developer Workbench) to develop even sophisticated XQuery statements with relative ease.

1.2.13 Benefits of DB2 pureXML technology

With integrated hybrid data management in DB2 9, you get an industry-leading, easy-to-use, standards-based, and enterprise-proven data server for both relational and XML data. Regardless of whether you want to power mission-critical solutions, or embed a free edition of the DB2 data server with your small-business applications, with DB2 9 and pureXML you can:

- ▶ *Reduce development time* through code simplification and avoiding XML-relational transformations in your applications.
- ▶ *Increase agility* through versatile XML schema evolution, allowing you to quickly modify applications as result of changing or introducing new services, products, or business processes
- ▶ *Improve insight* by harnessing previously unmanaged XML data and providing quicker query processing through XML-optimized storage and indexing.

For instance, Storebrand Group, a large financial services company in Europe, is seeing dramatic benefits as a result of using DB2 9 pureXML technology for powering their SOA solution. Development tasks that took them anywhere from two to eight hours with relational databases, now take them less than 30 minutes with DB2 pureXML. Schema changes using a relational data model that could take up to one week to implement, can now be done in a matter of minutes with DB2 9. Long-running queries running over shredded XML data that previously took days to complete now execute in seconds or minutes with pureXML.

With the pure XML support available in IBM DB2 9, it is far easier, faster and less expensive to run queries, share and retrieve data, and make document changes in response to new business requirements without impacting applications.

- Thore Thomassen, Senior Enterprise Architect, Storebrand Group

1.3 pureXML usage scenarios

There are many types of solutions and applications that can leverage pureXML services provided by DB2 9. In this section, we cover several common usage scenarios.

1.3.1 Integration of diverse data sources

The flexible data model of XML and the ability to join diverse documents in XQuery are ideal for integrating data from different sources. DB2 9 can serve as a data integration hub in a couple of different ways, depending on the requirements. Data from different sources can be converted to XML² and stored in DB2 9, where it can be joined. Alternatively, a distributed integration approach can be used where data can be left in their source repositories and accessed from a DB2 9 database using XML messages. You can also go a step further and make DB2 9 the integrating database in an SOA environment, using Web services to communicate between different systems, including existing data sources.

This last approach, also known as Service Oriented Integration (SOI), is what Storebrand is using to achieve integration across different products, IT infrastructures, and business processes to respond flexibly to customer requests. A diagram of their DB2 9-based SOI environment is shown in Figure 1-3.

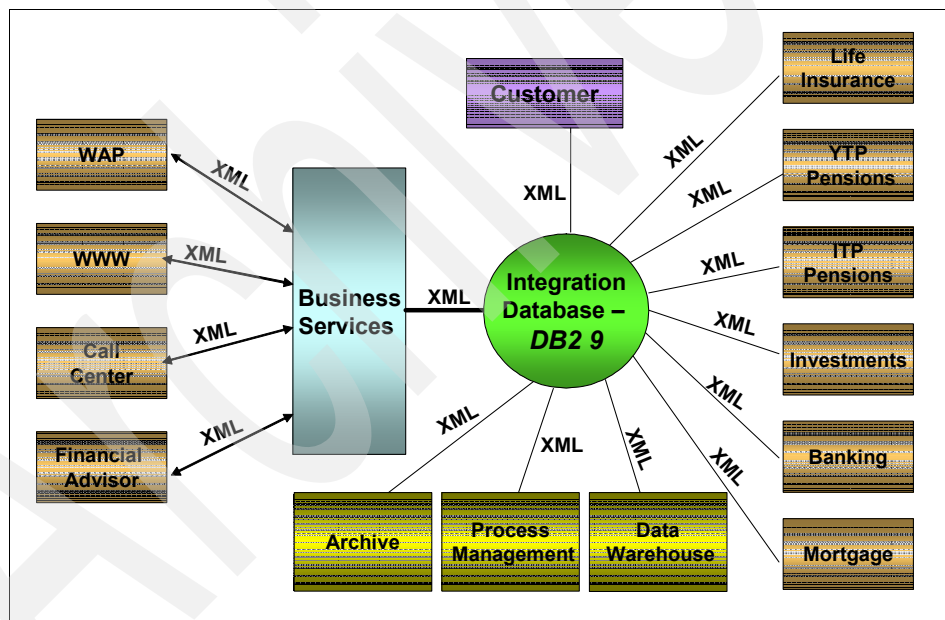


Figure 1-3 DB2 9 based on SOI environment

² Companies such as Exegenix provide conversion services to XML.

1.3.2 Forms and their processing

Paper-based forms are being replaced by electronic forms such as XForms for all sorts of goods and services. For example, the insurance industry has life insurance application and claim forms; banks have loan application forms and credit card application forms; and the government and public sectors have their fair share of forms. One method for storing data in electronic forms is simply to save the various fields of the form in different columns of a relational database. This has several drawbacks similar to the shredding approach. Query complexity increases for large forms, digital signatures are compromised, and changes to forms require time-consuming schema changes in the database, along with costly application rewrites.

A more elegant solution for many types of eForms (such as IBM Workplace™ Forms) is to store them as a whole in DB2 9 using XML. For example, some of the taxation departments are resorting to this approach for tax filings. Tax returns are complex documents and mapping to relational schemas requires complex shredding and queries. There are numerous changes to tax forms every year. By storing these tax forms as XML documents in DB2 9, “shredding can be avoided and changes to forms can be easily accommodated using the flexible schema evolution support.

After the forms are stored in the database, they are subject to subsequent processing and workflows. An insurance application, for example, goes through approvals that could involve multiple departments and be subject to status enquiries from brokers and customers. DB2 9 provides quick query and update access to subportions of documents (for example, qualifying information, approval sections, and status fields) and allows for multiple people to access a large library of documents simultaneously. See Figure 1-4.

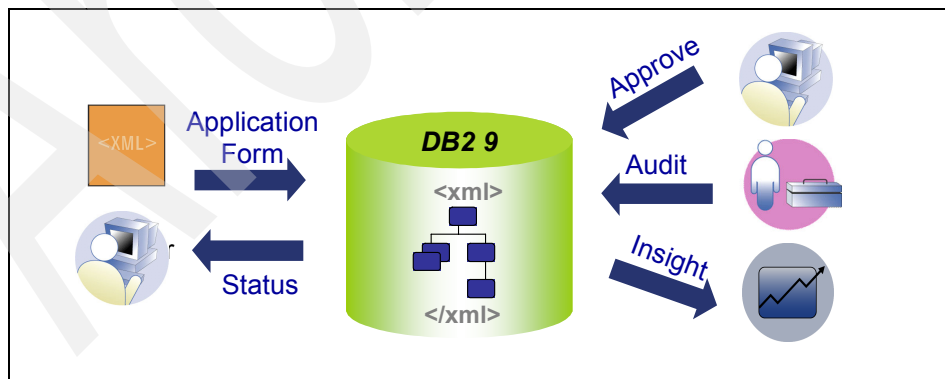


Figure 1-4 Storing Forms data as XML in DB2 9

1.3.3 Document storage and querying

XML is often classified as data-centric or document-centric. *Data-centric* XML tends to be more structured, whereas *document-centric* XML usually has less structure to it (although there can be some structural aspects). For example, legal contracts have some structured fields such as names, entities, dates, locations, and addresses. However, most of the contractual data, such as terms and clauses, appear as text. These types of XML documents are a natural fit for storage and management within DB2 9.

Nextance, a contracts management solution provider, is leveraging DB2 9 precisely for this purpose. As illustrated in Figure 1-5, users of a DB2 9-powered Nextance solution can quickly access both structured and unstructured portions within contract documents due to XML optimized storage, XML indexes, and full-text search capabilities in DB2. Versatile subdocument level access in DB2 also allows Nextance to easily search for and re-use clauses in other documents, while allowing Nextance to customize the solution (make schema changes easily) for each client, and attend to the evolving nature of contracts.



Figure 1-5 Managing contracts with Nextance and DB2 9

Another example is that of a manufacturer using DB2 9 to store technical manuals. Their finished products are made up of mechanical parts, each consisting of other parts or subassemblies. The subparts and subassemblies have their own instructions or manuals. Each manual has some structured sections and some free-form sections. By using DB2 9 to store these manuals as XML documents, they are able to build complete manuals of the finished products using the hierarchy of manuals, while being able to quickly drill down and access instructions for subassemblies. They are also able to easily adapt and update the manuals for new models of their products.

DB2 9 is also suitable for simple or application managed document processing. Library bibliography and online documentation are examples of simple document processing applications. These applications can leverage the power of XQuery for search or dynamic composition of required document elements from the underlying XML documents. Applications such as Wikis and Blogs are also examples that require simple document object management and processing capabilities. These applications store, update, search, and retrieve text and other fragments in XML. Administrators and power users are likely to perform some additional query tasks on such application driven simple document systems. DB2 9 provides the necessary capabilities for these simple requirements.

There are numerous other examples where DB2 9 is an excellent fit for storing document-centric XML. Many business objects are now being generated as XML documents, such as orders, invoices, and even spreadsheets and word-processed documents. Storing these documents in a DB2 database allows for reliable management, fast subdocument level access, and the ability to derive deep insight. DB2 9 can also be used as a building block for other content and document management applications.

1.3.4 XML for transactions

The DB2 engine has been proven to deliver scale and performance for even the most demanding transactional requirements. Benchmark data confirms that the same holds true for XML transactions. As an example, see:

<http://www-128.ibm.com/developerworks/db2/library/techarticle/dm-0606schiefer/>

As business-critical transactions are conducted using XML, DB2 with pureXML capabilities becomes a natural fit as well. There are numerous drivers behind XML transactions. Message-based transactions within service-oriented architectures are one such driver. Companies want to capture critical business data, route it appropriately, prioritize processing (such as by customer value) and perform analysis on these data items. For example, a goods distributor is moving towards a message-based infrastructure to manage business interactions with other value-chain members. XML data is used as the data objects of the transactions (such as for purchase and sale of goods). This XML data (such as buy/sell records) can be retrieved, updated, searched according to filtering criteria, and analyzed using the XML capabilities in DB2 9.

Industry standards are also driving XML-based transactions. For example, FIXML is being used for trading stocks and other financial instruments. DB2 9 can serve as the transactional engine behind such financial transactions and store the resulting XML data in a native format for fast queries and updates, but also provides the flexibility to evolve schemas rapidly to handle new types of financial instruments and changes in standards. As a case in point, FIXML has had new versions every one to two years.

1.3.5 Syndication and XML feeds

Syndicating feeds on the Web popularized by blog and news sites are now being used for a variety of commercial purposes. DB2 9 offers the ability to serve XML (Atom or RSS) feeds through a Web services interface. Off-the-shelf feed readers can request current or dynamically generated feeds from DB2 and access individual XML data entries using appropriate links contained in the feeds.

DB2 can also serve as a repository for XML feeds and their data. Incoming XML feeds can be saved using the pureXML storage in DB2 and can be used for updating the database, deriving insight, or even for generating new feeds. For example, a brokerage firm can receive constant feeds about financial transactions or price tickers, which can be stored reliably using DB2 9. Figure 1-6 illustrates that DB2 9 can service XML feeds through a Web services interface and as a repository for XML feeds.

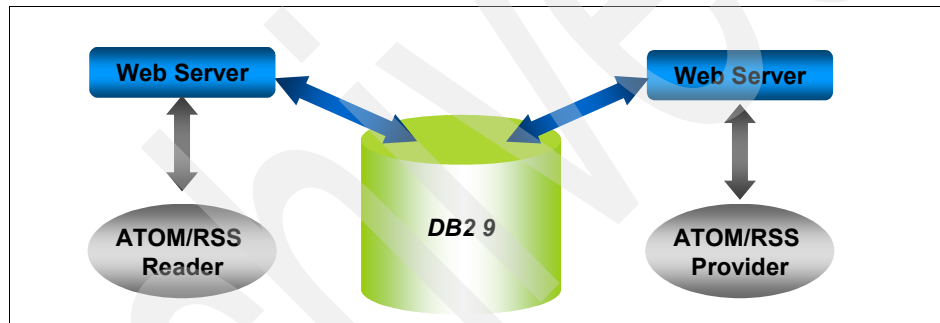


Figure 1-6 DB2 9 facilitates XML feeds

Consider a commercial scenario involving syndication benefits. A traditional method for a goods manufacturer or distributor looking to unload excess inventory is to sell it to a local overstock inventory buyer who typically offers only a few cents on the dollar. There are now a number of online exchanges and trading platforms (such as amazon, eBay, and overstock.com) that open up new markets and channels for selling inventory. The goods manufacturer/ distributor using DB2 9 can have the database automatically analyze and post listings for overstock goods on such online marketplaces using Web services.

Furthermore, the goods vendor can use feeds to update information such as inventory data and pricing information in real time, as a result of supply and demand variations and changing inventory levels. The goods vendor can also store feeds using DB2 for other similar and competitive products, analyze data from these feeds, and adjust their offering tactics dynamically, often automatically using rule-based criteria.

1.3.6 XML as a better data model

As you build new applications, you might find that the XML flexibility, schema versatility, and hierarchical nature can provide a better data model than relational data models, for example, when dealing with:

- ▶ Semi-structured or unstructured data:
 - Healthcare records
 - Biological data
 - Contracts
 - Insurance claims
- ▶ Inherently hierarchical, nested or complex data:
 - Manuals
 - Books
 - Catalogs
 - Bills of materials
 - Land records
- ▶ Data with changing or evolving schemas:
 - Forms
 - Changing industry standard documents
 - New product versions
- ▶ Data with null or multiple values:
 - Addresses
 - Phone numbers (home, office, mobile)
 - Such data in patient records

If you decide to use XML as the data model for your applications, DB2 9 is a great choice for storing and managing this XML data for all of the reasons discussed previously.

1.4 Summary

In this chapter we reviewed the pervasiveness of XML data, its relevance for all kinds of organizations, and the importance for managing it well. We introduced pureXML technology in DB2 9 and how it can unlock the latent potential of XML with performance and development time/cost savings. We also examined several examples and business scenarios where usage of DB2 9 along with its pureXML technology is highly applicable.

1.5 References

This chapter contains references or excerpts from papers and articles indicated below:

- ▶ Saracco, C. M. *Managing XML for Maximum Return*, IBM White Paper, October 2005.
<ftp://ftp.software.ibm.com/software/data/pubs/papers/managingxml.pdf>
- ▶ Nicola, Matthias and Bert Van der Linden. *Native XML Support in DB2 Universal Database*, Proceedings of the 31st Annual VLDB, 2005.
<http://www.vldb2005.org/program/paper/thu/p1164-nicola.pdf>
- ▶ Saracco, C. M. *What's New in DB2 Viper: XML to the Core*, IBM developerWorks® article, February 2006.
<http://www-128.ibm.com/developerworks/db2/library/techarticle/dm-0602saracco/>

Sample scenario description

In this chapter, we introduce our sample online unsecured loan application, XMLoan. In this application we explore DB2 9 XML native data type and DB2 native XML storage. We demonstrate how to take advantage of the new data type and DB2 9 features in combination with other essential e-business technologies such as Web services, Web application server, and DB2 9 database server.

This chapter covers the following topics:

- ▶ Business requirements
- ▶ Data modeling
- ▶ Application description
- ▶ Sample environment requirements and setup

The design approach of our XMLoan application outlined in the following sections is not as thorough as it would be in real life. However, it serves our purpose for this book, which is to provide a showcase for what DB2 9 pureXML has to offer for realistic e-business applications.

2.1 Business requirements

This section describes the business requirements of FAMDI Bank, a fictional financial institution. FAMDI Bank is a growing business which has to update and streamline their unsecured loan application process. They are looking for a way to provide real-time status updates in order to better serve their customers. In addition, they also require a solution to accept new loan applications anytime from anywhere in order to reach more potential customers. FAMDI Bank would like to achieve this by having a Web application that enables customers to apply for unsecured loan applications, make monthly payments, and send feedback from the bank Web site.

Figure 2-1 illustrates a high-level overview of the bank business model for processing an unsecured loan application.

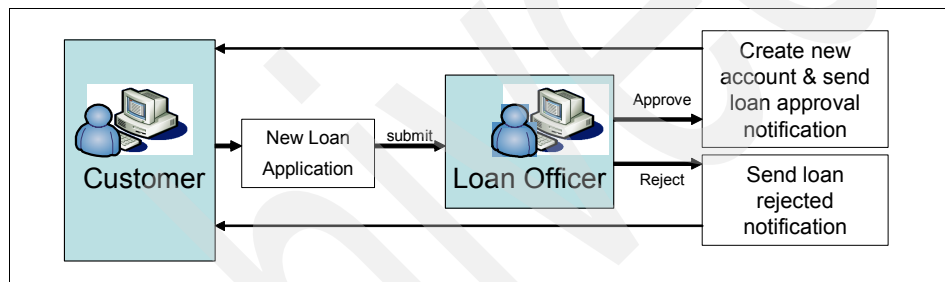


Figure 2-1 Unsecured loan process

The application should have the following functions:

- ▶ The customer can select the loan products and submit the loan application.
- ▶ The customer can make a payment.
- ▶ The customer can send feedback.
- ▶ The loan office can process the loan.
- ▶ The management team can create reports to analyze the loan, payment, and customer feedback.

2.1.1 Data modeling

With the business requirement identified, we can determine all entities and relationships of those entities in table and database designs.

Figure 2-2 shows the high-level of our data model including columns, keys, and the relationship between entities. The boxes in the diagram provide a logical description of the tables we have implemented for the XMLoan sample application. The columns with XML data type are noted with XML in parentheses.

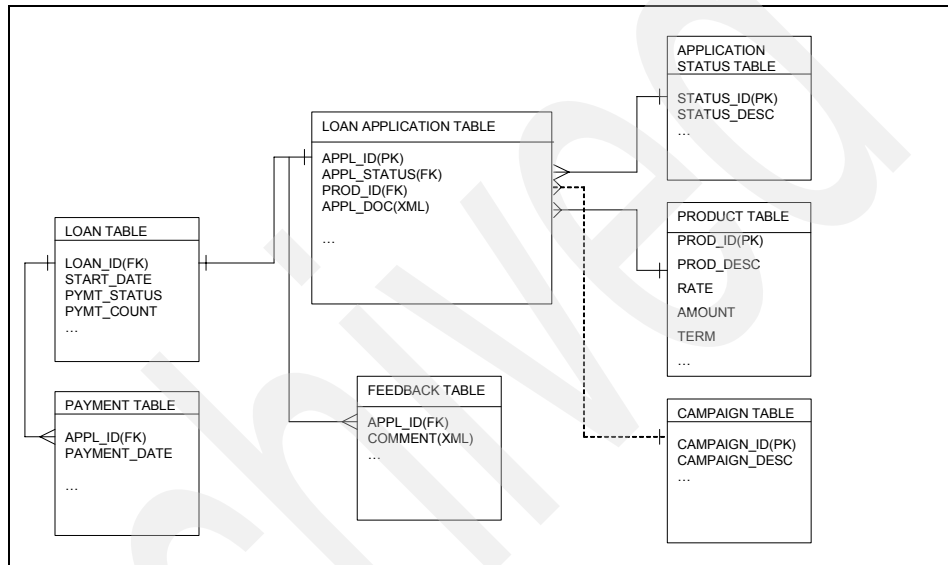


Figure 2-2 The online loan application database entities

Tables

The XMLoan sample application uses the following DB2 9 tables:

► CAMPAIGN

The CAMPAIGN table contains information about predefined advertising campaign sources and their description. This information helps the bank to identify the successful campaigns for future advertising.

The primary key is CAMPAIGN_ID.

► **FEEDBACK**

The FEEDBACK table saves the information entered by the customer from the bank Web site. The information includes the loan application ID, a rating of the loan process, and comments about the loan products. The COMMENT column is defined as XML data type to store customer feedback.

The foreign key LOAN_APPL_ID references the LOAN_APPLICATION table.

► **LOAN**

The LOAN table contains records for approved loans. Loan ID, loan start date and loan status ID are stored in this table.

The foreign key LOAN_ID references the PAYMENT table.

► **LOAN_APPLICATION**

The LOAN_APPLICATION table contains information pertaining to the loan application process, such as loan application ID, loan application status, and application documents, which are stored as documents in an XML column. The loan application ID is generated by our application after the customer submitted their applications. APP_DOC is an XML column to store the loan application.

The primary key is LOAN_APPL_ID. The foreign key PROD_ID references the PRODUCT table.

► **PAYMENT**

The PAYMENT table contains payment records. The information includes application ID and payment date.

The foreign key APPL_ID references the LOAN table.

► **PRODUCT**

The PRODUCT table contains information about all unsecured loan products offer by the bank. The information includes the loan product ID, a description for each product, the interest rate, the loan amount, and the loan term.

The primary key is PROD_ID.

► **APPL_STATUS**

The APPLICATION_STATUS table contains information about the loan application status.

The primary key is STATUS_ID.

2.2 Application description

The application *XMLoan* serves as an online unsecured loan application processing for the bank to enable the company to attract more potential customers.

The XMLoan application is a two-sided application. One is the customer side and the other is the bank employee side, which is the loan officer side. The customer side provides a Web interface for a customer to apply for a new loan, make a payment on an existing loan, or to submit feedback about the loan process or product offers. The bank employee side provides a Web interface for the loan officer to process the loan, manage customer records, and run monthly reports.

Figure 2-3 shows the home page of XMLoan application.

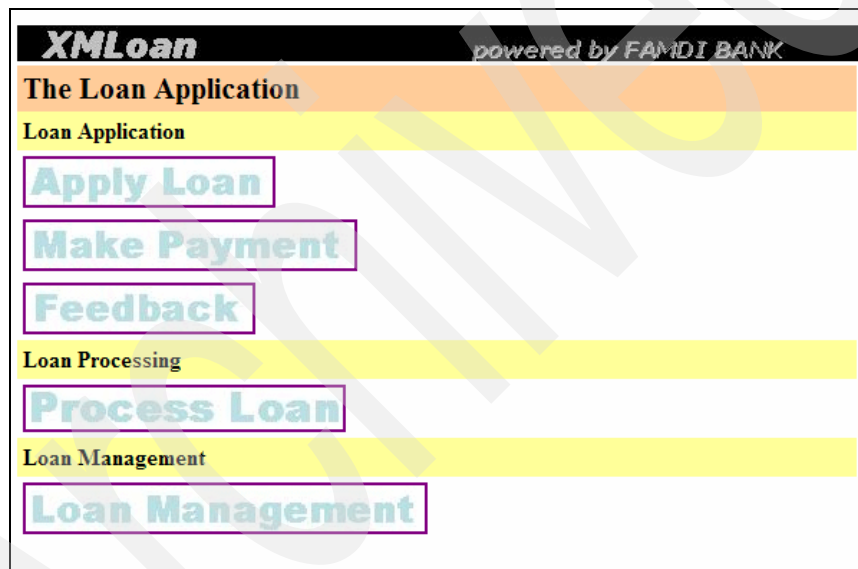


Figure 2-3 XMLoan application home page

2.2.1 Loan application

The loan application will utilize a Web interface for the customer. This interface allows the customer to create a new loan application, make a monthly payment, or submit feedback on the loan process. When a customer creates a new loan application, he is presented with a confirmation page that includes the loan application identification number and a message that the loan was successfully submitted to the bank. The customer can optionally provide an e-mail address and receive a notification through e-mail in addition to the Web confirmation. The loan process flow is shown in Figure 2-4.

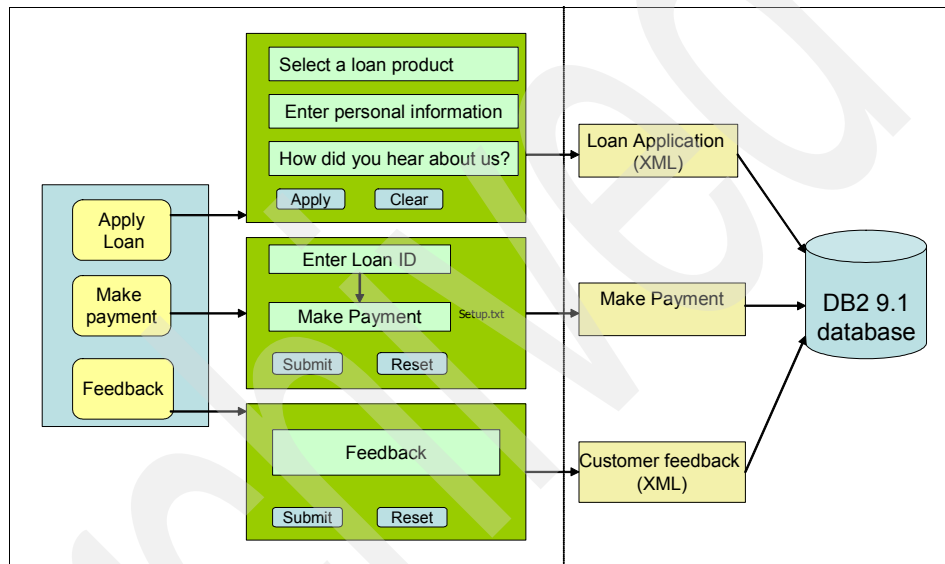



Figure 2-4 Loan application process flow

Apply Loan

The loan application form is launched when the customer selects **Apply Loan**, as shown in Figure 2-5. The customer selects a loan product from the Product drop-down menu, and enters the required information such as name, address, and financial data.

Address  http://localhost:8080/XMLoan/loanForm.jsp

XMLoan powered by FAMDI BANK

Loan Application Form

Loan Product Selection

PRODUCT:

Personal data

First Name:

Last Name:

Date of Birth (YYYY-MM-DD):

SSN (xxx-xx-xxxx):

Street:

City:

State:

Zip:

Home Phone:

Email Address:

Company:

Position:

Business Phone:

Income:

Debt:

Expenses:

Assets:

Campaign

How did you hear about us?

Figure 2-5 Online loan application form

After having successfully submitted their unsecured loan information, the customer receives a confirmation message and is assigned a loan application identification. The confirmation page is shown in Figure 2-6.

XMLoan powered by FAMDI BANK

Loan Application -Succeeded-

Your request has been received.

Your request application ID is 119

[Back to Apply Loan](#)

[Back to Home](#)

Figure 2-6 Loan submission confirmation page

Apply Loan process program flow

When the customer clicks **Apply Loan**, the XMLoan application accesses the PRODUCT and CAMPAIGN tables to retrieve product and campaign descriptions. This information is rendered in the Loan Product Selection and Campaign fields of the loan application form. After the customer clicks the **Apply** button to submit the loan request, the values that the customer entered are verified for validity. If the information is all valid, the XMLoan application takes the input values and generates a well-formed XML document. The XML document then gets inserted into the APPL_DOC column of type XML in the LOAN_APPLICATION table within the DB2 9 database.

Figure 2-7 shows the program flow.

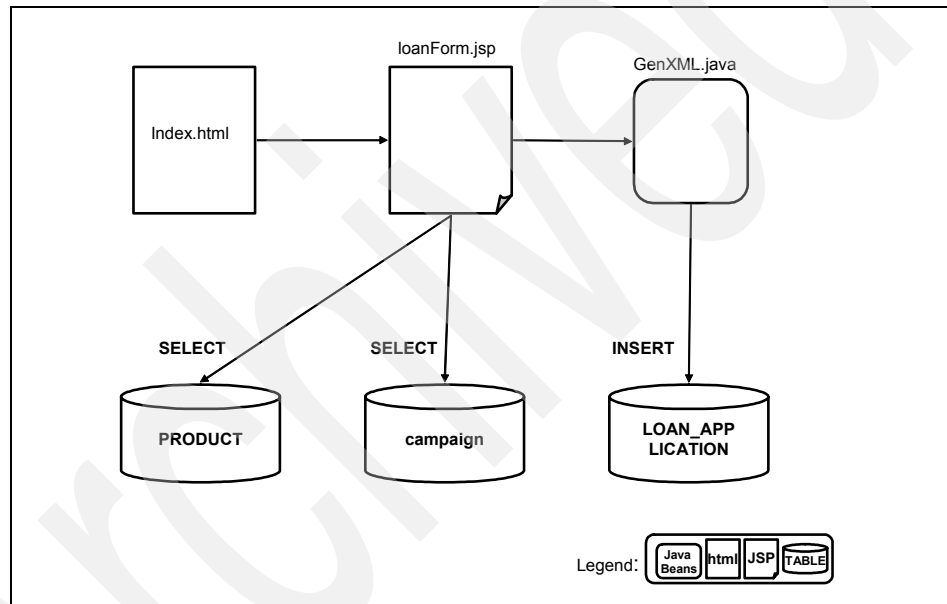


Figure 2-7 Loan application process behind the scenes

Make Payment

In addition to applying for a new loan, the customer can make a payment on an existing loan account. To make a payment, the customer can select the hyperlink **Make Payment** from the home page. The customer is presented with a payment page where they can enter a loan identification number to bring up their record. By entering the Loan ID and selecting the **Enter** button, a new page is launched to allow the customer to view payment information and make a monthly payment on their loan account.

By selecting the **Make Payment** button, the customer confirms that the payment will be sent to the bank, and receives a confirmation message with the payment history displayed. The Make Payment process flow is illustrated in Figure 2-8.

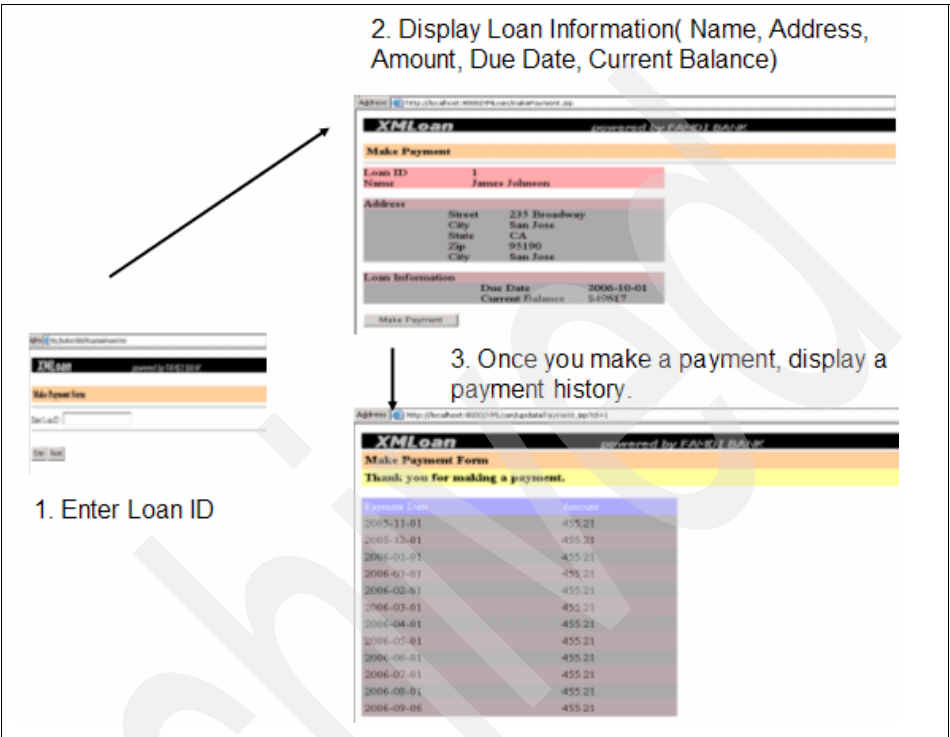


Figure 2-8 Make payment process flow

Making Payment process program flow

When the customer starts a loan payment process by selecting the **Make Payment** hyperlink, behind the scenes the application loads the makePayment.html page where it accepts the Loan ID as input from the customer and starts the makePayment.jsp. The application then retrieves the customer record such as name, address, loan total amount, current due date, and current balance from the LOAN_APPLICATION table and renders the information to makePayment.jsp where the customer is able to confirm the payment.

After the customer clicks the **Make Payment** button, the application updates the PYMT_STATUS column in LOAN table and inserts a new record to the PAYMENT table, as well as retrieving the payment history to render to the updatePayment.jsp for the customer to view. The application flow for the Make Payment process is shown in Figure 2-9.

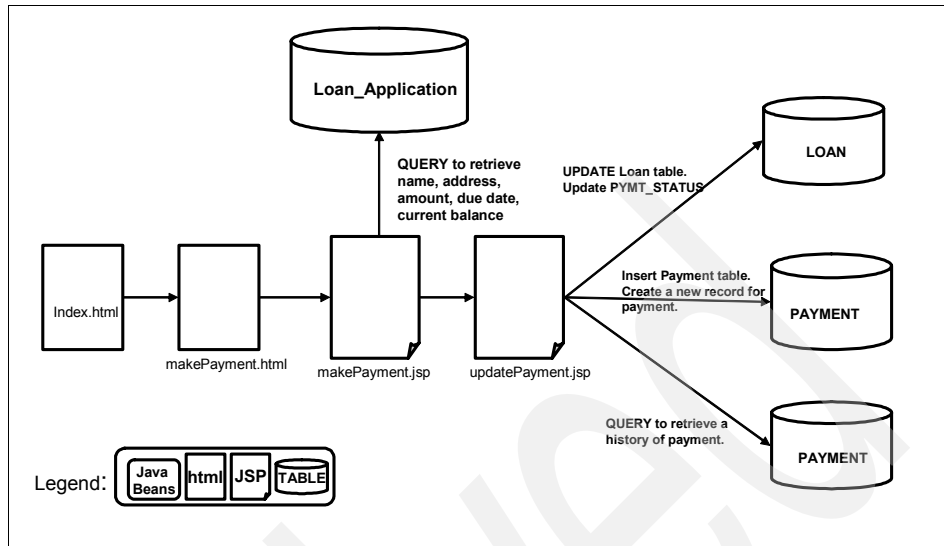


Figure 2-9 Make payment process program flow

Feedback

Alternately, the customer can submit feedback about the loan request process or current unsecured loan product offers by selecting the **Feedback** hyperlink from the home page (Figure 2-10).

XMLoan powered by FAMDI BANK

Feedback Form

Enter Application ID : 1

Select Rating : ☐ Excellent! ☒ Good ☐ Acceptable ☐ Neutral ☐ Need improvement

comment :
This is good. Thanks!

Submit Reset

Figure 2-10 Feedback form

Feedback process program flow

When the customer completes the form and selects the **Submit** button, the application receives the inputs. It then uses the values to generate an XML file and stores it into the FEEDBACK table within the XMLRB database. Figure 2-11 illustrates the program flow behind the scenes.

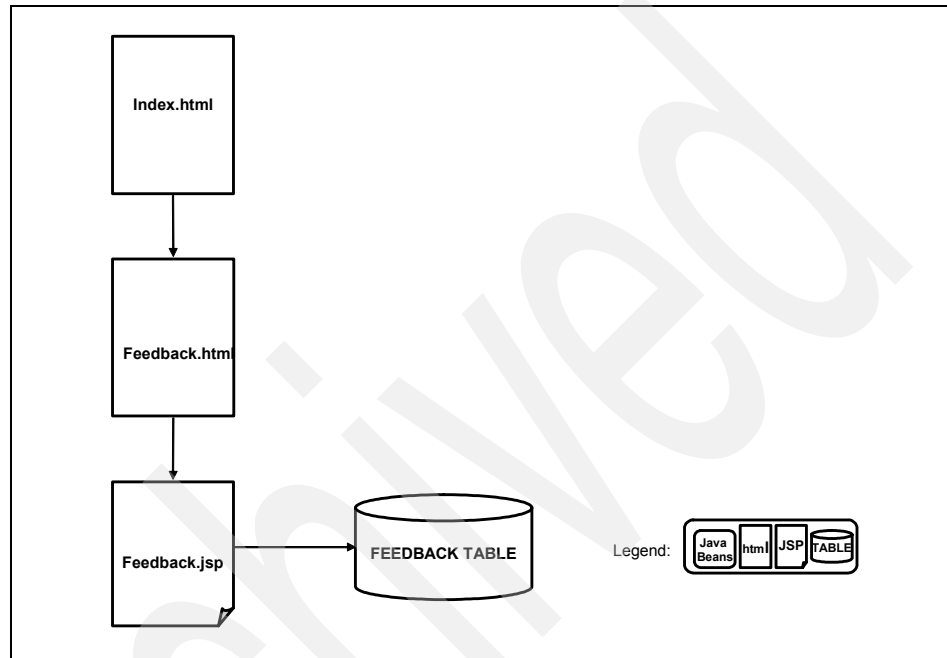


Figure 2-11 Feedback process program flow

2.2.2 Loan processing

The XMLoan sample application utilizes a Web interface for the loan officer. The loan officer uses this interface to review submitted unsecured loan applications. The loan application will be verified for accuracy and analyzed for viability. Based on analysis, the loan officer will either approve the loan, reject the loan, or present the customer with a new loan offer if the customer does not qualify for the amount requested. When the loan officer selects the **Loan Process** hyperlink on the loan home page, the loan officer is presented with the Loan Process page with a listing of all submitted loans. See Figure 2-12.

XMLoan

powered by FAMDI BANK

New Applications

Select one to proceed.

APPLICATION ID	Loan Documents
17	Process
18	Process
19	Process
20	Process
21	Process
22	Process
23	Process
24	Process
25	Process
26	Process
27	Process
28	Process
29	Process
30	Process
31	Process

Figure 2-12 Main loan process page

The loan officer can start processing an unsecured loan application by selecting any application from the loan listing. When a loan application is selected, the processing page (Figure 2-13) is presented, where the loan officer can analyze the record and approve or reject the loan.

Address <http://localhost:8080/XMLoan/processApp.jsp?ID=17>

XMLoan
powered by FAMDI BANK

Application Processing

Application ID(=17)

Personal Information

First Name	Last Name	SSN	Date Of Birth	Company	Postion
John	000001	Smith000001	092-43-1317	2/23/1967	My company Developer

Financial Information

Income	Debt	Assets
11300.0	34000.0	900.0

Product Information

Description	RATE	AMOUNT	TERM
\$5000 for 12 months at 6.25%	6.25	442.71	12

To be approved: \$50000 for 120 months at 9.25%

Figure 2-13 Process loan application page

When the loan officer approves a loan request, a confirmation page will be displayed with a message. The message states that the loan has been successfully approved, and a welcome notification is sent to the customer to let him or her know of the loan approval decision.

When the loan officer rejects a loan request, a loan decision notification is sent to the customer. The notification includes information explaining the decision and, optionally, proposes a new loan offer for which the customer can be approved.

Process loan application process program flow

When the loan officer selects **Process loan**, the application loads the application listing page. As a loan application identification number is selected, the application accesses the LOAN_APPLICATION table to retrieve the loan document. It also accesses the PRODUCT table to retrieve the product information and renders to the processApp.jsp page.

If the **Approve** button is selected, the applicationApproved.jsp page is loaded and the application inserts a new record to the LOAN table, as well as updating the APPL_STATUS column within the LOAN_APPLICATION table with the

approved status. Alternately, if the **Reject** button is selected, the application loads the applicationRejected page and just updates the APPL_STATUS column in the LOAN_APPLICATION table. The process is illustrated in Figure 2-14.

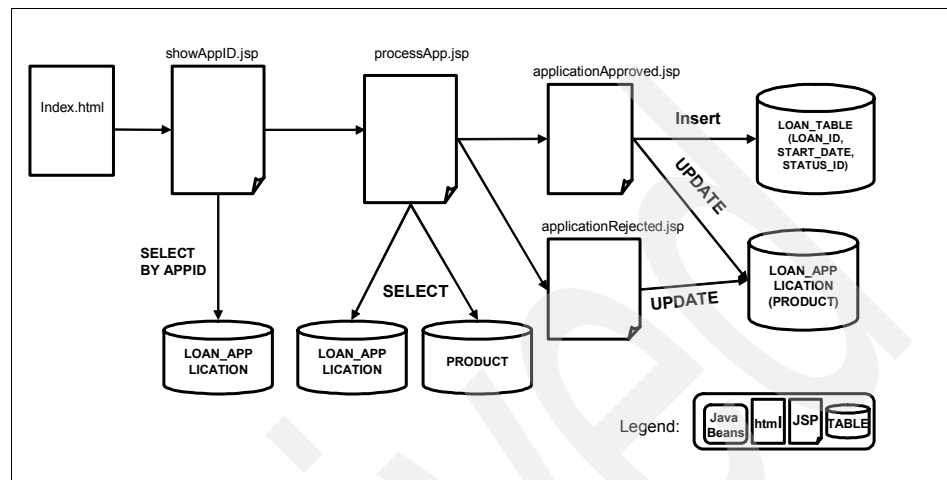


Figure 2-14 Loan processing program flow

2.2.3 Loan management

The XMLoan sample application utilizes the Web interface for the loan officer. The loan officer will use this interface to display and manage loan records. When the loan officer selects the **Loan Management** hyperlink from the loan home page, the loan officer has the option to run monthly reports such as monthly loan statistics, payment analysis, campaign analysis, and customer satisfaction analysis. Figure 2-15 shows the loan management interface.

XMLoan <small>powered by FAMDI BANK</small>		
Reports		
Execute Report	Descriptions of Reports	Show Queries
LOAN LIST	This lists all of the loan applications.	Show Query
PAYMENT ANALYSIS	This report lists the number of payments behinds. Negative counts indicates there is advance payment made.	Show Query
CAMPAIGN ANALYSIS	This report shows the active loan generated by each campaign.	Show Query
CUSTOMER SATISFACTION	This report shows the products rating from customers.	Show Query
Back to HOME		

Figure 2-15 Loan management interface

When the loan officer selects a monthly report, a separate report page is displayed for viewing and analysis. For example, when clicking **Customer Satisfaction**, the Customer Satisfaction report is displayed (Figure 2-16).

Product ID	Product Description	Rating	Count
1	\$5000 for 12 months at 6.25%	3	2
1	\$5000 for 12 months at 6.25%	4	1
2	\$10000 for 36 months at 7.50%	0	1
2	\$10000 for 36 months at 7.50%	2	1
2	\$10000 for 36 months at 7.50%	3	1
3	\$20000 for 60 months at 8.75%	1	1
3	\$20000 for 60 months at 8.75%	2	1
3	\$20000 for 60 months at 8.75%	3	1
4	\$30000 for 84 months at 9.00 %	1	1
4	\$30000 for 84 months at 9.00 %	4	2
5	\$50000 for 120 months at 9.25%	0	1
5	\$50000 for 120 months at 9.25%	1	3
5	\$50000 for 120 months at 9.25%	2	1

[Back to HOME](#)
[Back to Run Reports](#)

Figure 2-16 Customer Satisfaction report

In the Reports page, we provide Show Queries buttons for each report. In a real-life application, this field would not be shown to the loan officer. This was added so that we can show you the XQuery constructed for each report. Figure 2-17 shows the Customer Satisfaction query.

```
with fb (appl_id, rating) as
(select appl_id, xmlcast(xmlquery('$a/Feedback/Entry/Rating/text()') passing feedback.comment as "a") as integer) from feedback)
select p.prod_id,prod_desc, fb.rating, sum(1) count
from fb, loan_application as la, product as p
where (fb.appl_id = la.appl_id) and (la.prod_id = p.prod_id)
group by p.prod_id,prod_desc, rating
order by p.prod_id, rating
```

Figure 2-17 Sample query page

Loan Management process program flow

When the loan officer clicks the **Loan Management** hyperlink in the home page, the XMLoan application loads the report listing page, which shows all reports that the loan officer can display. If any monthly report hyperlink is selected, the application accesses the LOAN_APPLICATION table to retrieve necessary information to render to corresponding result page. Figure 2-18 illustrates the Loan Management process program flow.

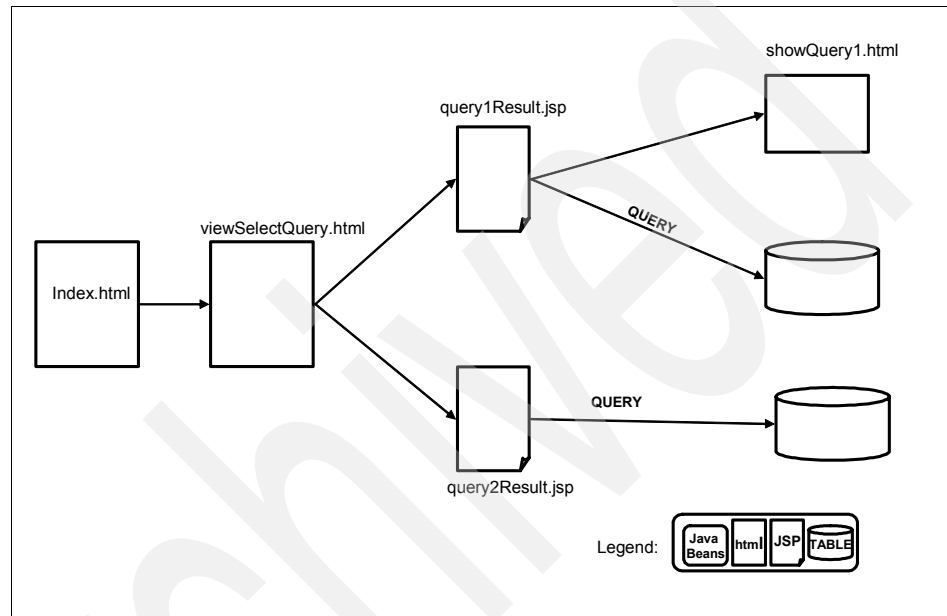


Figure 2-18 Loan Management process program flow

2.3 Application setup

In a real-life implementation of our XMLoan application, the customer side application would be hosted on one or more HTTP Servers, and the loan officers would have their own loan officer side application server, which is normally behind the bank network firewall. To keep the setup simple, and to make it possible for you to easily replicate our environment, we host the customer side and the loan officer sides of the application on the same machine where our DB2 9 database server is residing.

The XMLoan application, setup script, and data files are available for download from the IBM Redbook Web site. The download details are given in Appendix B, “Additional material” on page 373.

Following are the installation instructions for the XMLoan application:

1. Install Apache 41: The Apache HTTP Server can be downloaded from the following Web site:

<http://tomcat.apache.org/download-41.cgi>

Apache Tomcat 4.1 requires the J2SE™ Software Development Kit (SDK).

Note: If you do not have J2SE, you can use the copy that comes with DB2 9 located under `<DB2_install_directory>\SQLLIB\java\jkd`. The default `<DB2_install_directory>` for Windows is `C:\Program Files\IBM\`.

2. Install DB2 9: You can run any DB2 9 edition with the XMLoan application.

DB2 Express-C is available for free download at the following IBM DB2 Universal Database Web site:

<http://www-306.ibm.com/software/data/db2/udb/db2express/>

3. Add the following environment variables:

`JAVA_HOME=<db2_install_directory>\SQLLIB\java\jkd`
`CATALINA_HOME=C:\Program Files\Apache Group\Tomcat 4.1`

Edit the existing environment variable PATH and add `%JAVA_HOME%\bin` to the path.

4. Copy `db2jcc.jar` and `db2jcc_license_cu.jar` from `<DB2_install_directory>\SQLLIB\java` to `%CATALINA_HOME%\common\lib` directory.

5. Copy `XMLoan.war` to `%CATALINA_HOME%\webapps\` directory.

Apache will deploy the application automatically.

6. Run `setup.txt` from DB2 Command line processor or DB2 Command Editor to create a UTF-8 database, tables, and populate data required for the tables, as follows:

`DB2 -tvf setup.txt`

The DDLs for creating database and tables are listed in A.1, “Creating XMLoan database” on page 362.

7. Install a partial update stored procedure `DB2XMLFUNCTIONS.jar` using the following steps:

- a. Start DB2 command line processor.

- b. Set up the DB2 environment variable using the following command:

`DB2SET DB2_USE_DB2JCCT2_JROUTINE=on;`

- c. Update the Java heap size using the following command:

```
DB2 UPDATE DBM CFG USING JAVA_HEAP_SZ 1024;
```

- d. Install the stored procedure jar file into DB2 using the following commands:

```
DB2 -TD;  
CONNECT TO xmlrb USER db2admin USING db2admin;  
CALL SQLJ.INSTALL_JAR('file:///c:/temp/DB2XMLFUNCTIONS.jar',  
db2xmlfunctions, 0);
```

You have to replace the c:/temp with the directory where the XML application is downloaded.

- e. Register the stored procedure: Use the command shown in Example 2-1 to create the stored procedure. You can copy and paste, then run the command in CLP.

Example 2-1 Create stored procedure

```
CREATE PROCEDURE db2xmlfunctions.XMLUPDATE(  
  IN COMMANDSQL VARCHAR(32000),  
  IN QUERYSQL VARCHAR(32000),  
  IN UPDATESQL VARCHAR(32000),  
  OUT errorCode INTEGER, OUT errorMsg VARCHAR(32000))  
  DYNAMIC RESULT SETS 0  
  LANGUAGE JAVA  
  PARAMETER STYLE JAVA  
  NO DBINFO  
  FENCED  
  NULL CALL MODIFIES SQL DATA  
  PROGRAM TYPE SUB  
  EXTERNAL NAME  
  'db2xmlfunctions:com.ibm.db2.xml.functions.XMLUpdate.Update';
```

8. Start the XMLoan application: Open your browser and enter the following URL on your Web address:

<http://localhost:8080/XMLoan>

The application should start on your browser.

XML database design

It is crucial to design a database that maintains the consistency and integrity of different forms of data. There are more options when you design a hybrid database. In this section, we discuss the options available. Whether the database is hybrid or pure relational, the database design activities include both logical design and physical design.

In this chapter, we discuss the following topics:

- ▶ Architecture overview
- ▶ Logical database design
- ▶ Physical database design

3.1 Architecture overview

DB2 9 is a hybrid database system. It can have both relational data and native XML data in the same database. DB2 9 introduces an XML data type. The relational data is stored in tabular structures and the XML data is stored in tree structure. The structure enables XML data to be stored in its hierarchical form within columns of a table. Because data is stored in its native form, both types of data can benefit from the performance.

On top of both tabular structures and tree structures, there is one hybrid database engine that processes both types of data. There are two different parsers to process SQL and XQuery. A single compiler is used for both languages. DB2 9's compiler and optimizer can handle both languages. An application can use the combinations of SQL and XQuery to access relational and XML data in the hybrid database.

Figure 3-1 shows an architecture overview of the hybrid database system.

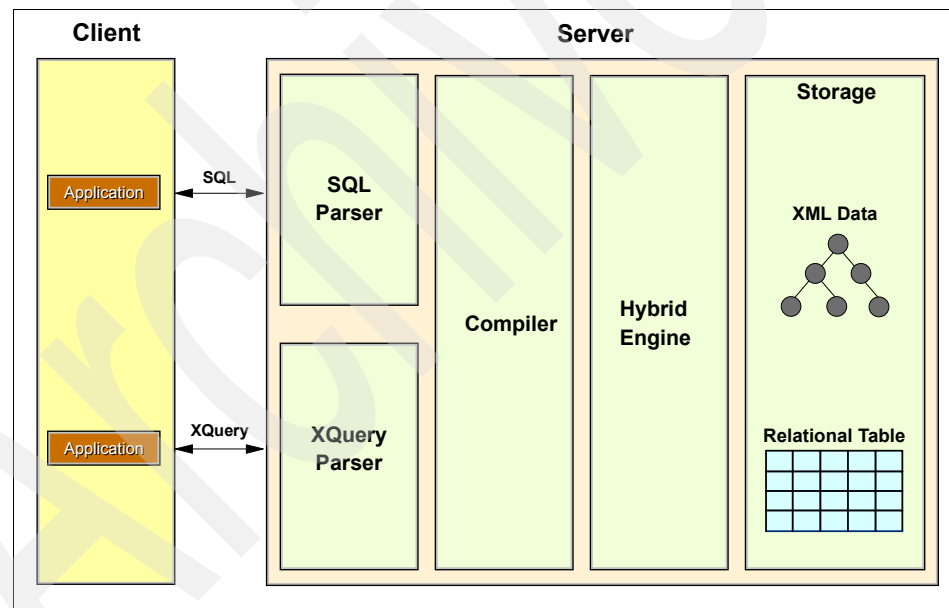


Figure 3-1 DB2 architecture

DB2 9 treats XML as a native data type. It has a pureXML storage, meaning that XML data is stored in XML form, which is a hierarchical structure. Figure 3-1 shows how XML and relational data are stored separately.

Example 3-1 shows an example of an XML document.

Example 3-1 XML document

```
<Customer>
  <Name>
    <FirstName>John</FirstName>
    <LastName>Smith</LastName>
  </Name>
  <DateOfBirth>1967-02-23</DateOfBirth>
  <SSN>123-45-6789</SSN>
  <Address>
    <Street>46 South Main Street</Street>
    <City>Los Gatos</City>
    <State>CA</State>
    <Zip>95030</Zip>
  </Address>
  <Employer>
    <Company>My company</Company>
    <Position>Developer</Position>
  </Employer>
</Customer>
```

Figure 3-2 shows an XML document in hierarchical form.

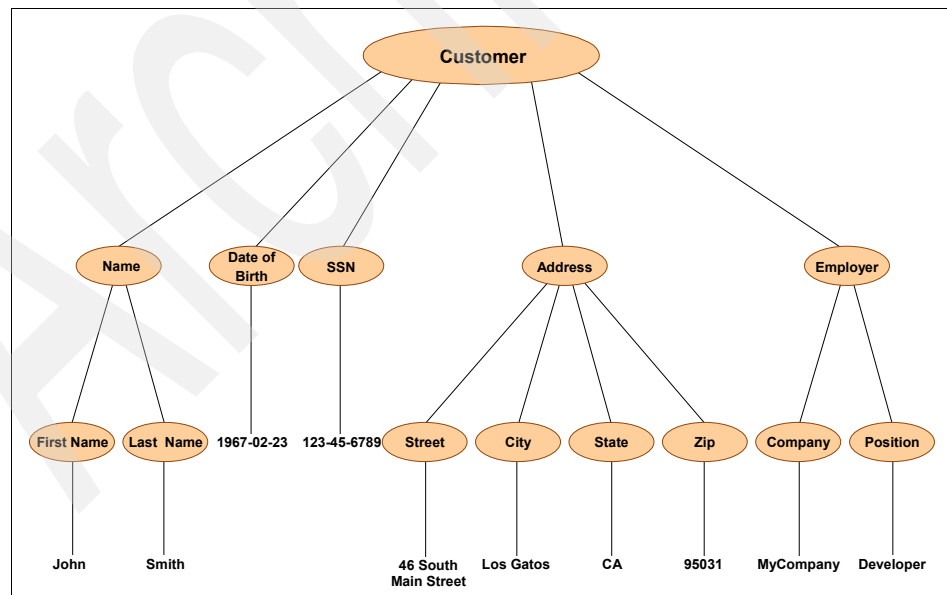


Figure 3-2 XML document in hierarchical form

An XML document must be well-formed in order to be inserted and imported into an XML column. Any attempt to insert or import an XML document that is not well-formed into an XML column will fail with an error. You can insert and import an XML document up to two gigabytes in size into an XML column. Similar to the long data types (LONG VARCHAR, LONG VARGRAPHIC, and LOB data), XML data is stored separately from the other contents of a table and can be stored in its own individual table space. DB2 9 stores XML data contained in table columns of the type XML in auxiliary XML storage objects. If the XML columns are stored in system managed space (SMS), the files associated with XML storage objects have the file type extension .xda.

XML data objects are stored separately from parent table objects. For each row of XML type column, there is an XML data specifier (XDS) stored in the table. The XDS has the information to access the XML data stored in the disk. The XDS is also used for IMPORT and EXPORT utilities.

Figure 3-3 shows the relationship among table, XDS, and an XML data column.

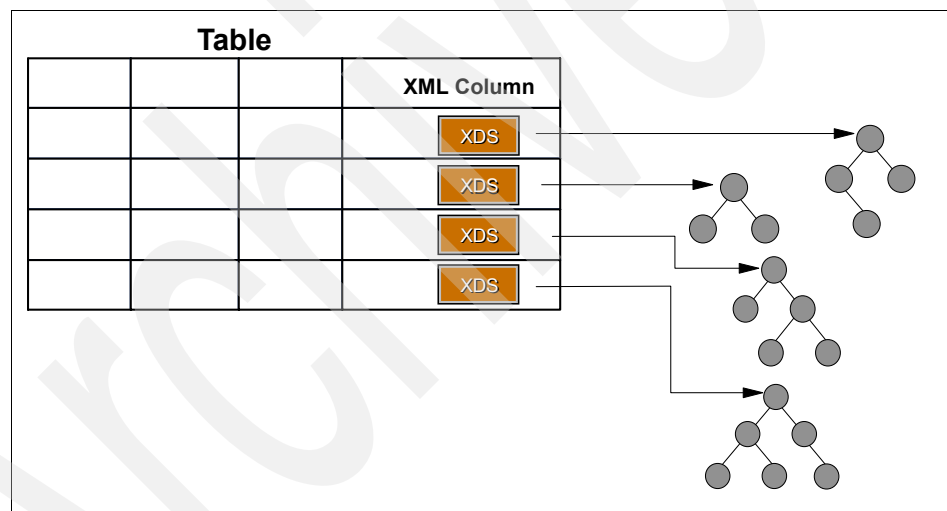


Figure 3-3 table, XDS, and XML data relationship

DB2 9 supports XML document validation with XML schemas. The validation usually takes place in insert and import time. XML schemas used for validation are registered in an XML Schema Repository (XRS). The XML schema is different from the schema in the relational database. A *relational database schema* is a collection of named database objects, which is used to logically group database objects and is also used as a name qualifier. An *XML schema* is a language that describes the structure and constraints for the contents of XML documents. XML schemas are discussed in more detail in 3.2.5, “XML schema” on page 49.

3.2 Logical database design

When you design a database on DB2 9, you will want to take advantage of the new pureXML features that are now available. In this section, we discuss the new data type XML, XML index, and views. In a hybrid database, table design includes considerations of what kind of data should go to relational data columns and what should go to XML columns. In some cases, you might prefer to decompose XML documents into relational tables. Another consideration for storing XML documents in a database is document validation, which DB2 9 now supports during insert and import. In this section, we discuss the validation types and the advantages and disadvantages of validation.

3.2.1 XML data type

DB2 9 introduces the XML data type. In Version 9.1, DB2 supports tables with the XML data type in a Unicode (UTF-8 code set) database. All XML data must be stored in the database in the UTF-8 code set. Unlike a VARCHAR or a CLOB type, the XML type has no length associated with it. The XML storage and processing architecture imposes no limit on the size of an XML document. Currently, only the client-server communication protocol limits XML bind-in and bind-out to 2 GB per document.

You can insert only well-formed XML documents. They must adhere to certain syntax rules specified in the W3C standard for XML. DB2 checks the well-formedness of inserting XML document at insert time. The insert statement will fail with the error code SQL16110 if the XML document is not well-formed. When the XML document is inserted into the database, it is stored in a parsed hierarchical form (tree structure). Because the XML structure is hierarchical, this form is the most natural way to store XML documents.

The XML data type is a native DB2 9 data type. DB2 uses an internal representation to process the XML data type, which is not compatible with the string data type. An XML data type can be transformed to a string type by using the function XMLSERIALIZE. In applications, you can bind an XML column to a binary, string, or XML type variable. Conversely, a string data type can be transformed to an XML data type by using the function XMLPARSE. In applications, you can also bind an XML column to a binary, string, or XML type variable. You can insert, update, and delete XML data in XML data type columns with SQL data manipulation statements just as you can with other relational data type columns.

XML document as LOB type

An XML document can be inserted into a LOB/LONG VARCHAR type column. The advantage of doing this is that the insert is fast because there is no necessity

for parsing for the LOB/LONG VARCHAR type. Selecting the whole XML document is also fast because, unlike XML type, the data is not stored in tree structure and there is no necessity for serialization. The trade-off is slow performance for searching and extracting the XML document. Because there is no parsing at insert time, the XML document is not checked for well-formedness.

For the XML type, all XML documents are parsed and checked to see if they are well-formed. The select of whole XML documents from XML type columns also takes more time than from LOB/LONG VARCHAR type columns because of serialization. The search, extract, and partial update from XML type columns is faster than for LOB/LONG VARCHAR type columns. XQuery is available for XML type columns.

In general, XML documents can be stored as LOB/LONG VARCHAR types if one or more of the following statements are true:

- ▶ No process is required on the XML document. It is not necessary to search, to extract, or to partially update the document. For example, the document must be kept intact for business rules or legal reasons.
- ▶ The XML document is from a trusted source that guarantees its well-formedness, and there is a requirement for validating it.
- ▶ The well-formedness is not important for the XML document.
- ▶ The only operations on the XML document are insert and whole document select. The performance of insert and select are the most important.

3.2.2 Relational structure versus XML structure

When it comes to hybrid database design, the first question would be: What data should be stored in relational form, and what should be stored in XML? The answer depends on whether relational schema or XML schema would offer a better way to describe your data. The main difference between relational schema and XML schema is that relational schema describes data as strongly structured and typed. XML schema describes data as loosely structured and typed. XML schema describes data order, but relational schema does not.

In general, data that has the following properties should be stored in a relational structure:

- ▶ The data is processed with other relational data.
- ▶ The data is better described in tabular format.
- ▶ The data has value that is independent of XML hierarchies.
- ▶ The data has to be accessed by applications that process relational data.

There are many considerations for designing a relational database. In this book, we focus our discussion on the XML data type. You can find relational database

design detail in *Administration Guide: Planning*, SC10-4223. The following link directs you to the PDF file of this DB2 9 manual:

ftp://ftp.software.ibm.com/ps/products/db2/info/vr9/pdf/letter/en_US/db2d1e90.pdf

In general, data that has the following properties should be stored in XML:

- ▶ The data is better described in hierarchal format.

High complexity of the hierarchy data might require a high number of relational tables to store it and can be difficult to map into relational structure. XML is the most natural way to store such data.

- ▶ The schema is constantly changing and evolving.

Business rules can change and affect the schema. In general, it is easier to do XML schema evolution than to change relational schema. We discuss the schema evolution in more detail in 5.2.5, “Schema evolution” on page 206.

- ▶ Many attributes of the data are empty or unknown.

If you map the data into relational tables, there will be many null values in the tables. If the data is complicated and large, you might require many relational tables, and most of the values in the table would be null. To process the data in relational tables, you usually have to join many tables with complicated SQL statements. The XML schema is more flexible. You do not have to store the null values if the XML schema is well-designed. The XQuery accessing such data in XML would not be complicated.

- ▶ There is little data with a highly complex structure.

If you store such data in relational tables, you will have complicated relational schemas, which means you require many tables. Managing these tables can have overhead. The SQL query to access such data requires joining many tables. If you have to process this data together with other data, the SQL query will be even more complicated. A small amount of data with a highly complex structure should be stored in XML.

Decomposing XML documents into relational tables

XML documents can be decomposed into a relational table. Conversely, decomposed XML documents in relational tables can be composed and published to an XML document. However, the published XML document might be different from the original XML document. During the process of decomposing, the XML document loses most of its structure in order to map into the relational table; not all the tags are stored in the relational tables. For example, the order of the XML document is lost. If the order of the document is important, the XML document should be stored in a CLOB/LONG VARCHAR or XML type column.

Example 3-2 shows a sample XML document.

Example 3-2 XML document

```
<ORDER_ID='83492' CUST_ID='93457'>
  <ITEM>
    <PROD_ID>94872</PROD_ID>
    <PROD_NAME>PEN</PROD_NAME>
    <PRICE>19.95</PRICE>
    <QUANTITY>30</QUANTITY>
  </ITEM>
  <ITEM>
    <PROD_ID>94866</PROD_ID>
    <PROD_NAME>BINDER</PROD_NAME>
    <PRICE>7.95</PRICE>
    <QUANTITY>26</QUANTITY>
  </ITEM>
  <ITEM>
    <PROD_ID>92219</PROD_ID>
    <PROD_NAME>LABELS</PROD_NAME>
    <PRICE>12.95</PRICE>
    <QUANTITY>250</QUANTITY>
  </ITEM>
</ORDER>
```

An order can have one or more items. We require two relational tables to decompose the XML document. The ORDER table has two columns. ORDER_ID is the primary key. The ITEM table has 5 columns. ORDER_ID is a foreign key from ORDER table. An order can have many items. Table 3-1 and Table 3-2 show decomposed data in relational tables.

Table 3-1 ORDER table

ORDER_ID	CUST_ID
83492	93457

Table 3-2 ITEM table

ORDER_ID	PROD_ID	PROD_NAME	PRICE	QUANTITY
83492	94872	PEN	19.95	30
83492	94866	BINDER	7.95	26
83492	92219	LABLES	12.95	250

After the XML data is decomposed into relational data, you can use SQL to query the tables and add some indexes to improve the query performance.

Decomposing XML documents can be the right approach if the XML structure is simple. When decomposing XML documents, for every element that occurs more than one time, you usually require a separate table to represent it. For instance, in our previous example, the item element requires a separate table. This is fine with simple XML documents with a small number of elements that occur more than once. For complex XML documents, the insertion and selection of the data might then involve a huge number of tables. Also, for complex XML documents, decomposing is not always practical.

In general, XML documents can be decomposed if the data has the following properties:

- ▶ The XML document's structure can be decomposed to a reasonable number of relational tables.
- ▶ The XML document is used for data exchange. The original document is not important after the data is exchanged. The decomposing usually does not keep the structure of the document; for example, the order is lost.
- ▶ Partial update is frequent and update performance is important. The relational tables usually have better performance in individual column updates.
- ▶ The XML schema does not change. The XML schema is usually more flexible than relational schema. That means schema evolution is easier in XML schema. If the does not change, decomposing the XML document into relational tables would be a reasonable choice.
- ▶ The XML document must be mapped into existing relational tables. You might have already have some existing relational tables and you want to map your XML document into those existing relational tables.
- ▶ The XML document must be processed by an application that only has the ability to access a relational table.

3.2.3 XML indexes

Indexes provide a way to speed up finding and accessing data. They are typically used to improve query performance. DB2 9 supports XML index creation. Like relational indexes, while XML indexes provide faster queries, they can also make update, insert, and delete actions slower because the update, insert, and delete require index data. Indexes also have space overhead, requiring extra space to store the index data. You can have a relational index which is composed of one or more table columns. For an XML index, every index uses a particular XML pattern expression to index paths and values in XML documents stored within a single column.

Instead of providing access to the beginning of a document, entries in an index over XML data provide access to nodes within the document by creating index keys based on XML pattern expressions. Because multiple parts of an XML document can satisfy an XML pattern, multiple index keys can be inserted into the index for a single document.

If the performance of the query is the only priority, there is no necessity to perform update, insert, and delete, and if you do not care about space overhead, you can index everything. In reality, storage space is limited and most data requires update, insert, and delete. In general, data that is queried frequently and does not require modification is good candidate for indexing. What do you index? DB2 9 comes with tools to help, such as visuals and text-based explanations. You can test you queries against the data with explanations. By studying the explanation output, you would know if your index is useful and if there is another, better index. For more information about how to index XML data, see 5.1, “XML indexes” on page 174.

3.2.4 Views

You can create relational views from data in an XML column by invoking the function XMLTABLE, which is discussed in 4.3.2, “SQL/XML” on page 127. We show how to create a view with XMLTABLE here. In Example 3-3, a table with an XML column is created and one record inserted.

Example 3-3 Create a table with XML column

```
CREATE TABLE loan_application(appl_id bigint, appl_doc xml, appl_status
integer, prod_id integer);

insert into loan_application values( 11111,xmlparse(document'<xml
doc>Preserve whitespace),10,20);
```

You can create a view with relational column data and data from some elements in the XML document. Example 3-4 shows creating the view loan_application_view with three columns. The firstName and lastName are taken from an XML document.

Example 3-4 Creating the view

```
CREATE VIEW loan_application_view AS SELECT appl_status, t.lastName,
t.firstName FROM loan_application,
xmltable('$lo/application/customer/name' passing appl_doc as "lo"
columns lastName char(20) path 'lastName', firstName char(20) path
'firstName')as t;
```

Example 3-5 shows selecting the view `loan_application_view` and the result.

Example 3-5 Selecting the view

```
select * from loan_application_view
```

APPL_STATUS	LASTNAME	FIRSTNAME
10	John	Smith

1 record(s) selected.

Creating relational views for XML column data is easy, but you have to consider that DB2 does not use XML column indexes when queries are issued against such a view. For example, if you have an index on the element `firstName` and you issue an SQL query that restricts the result of the `lastName` column to be `Smith`, the index on element `firstName` is not used. DB2 would read all the XML documents and search for `Smith` in element `firstName`. If you have a lot of data, performance might not be as you expected.

If the query also has a highly restrictive predicate involving the indexed traditional SQL columns, you can mitigate a slow performance problem. It is because DB2 uses the relational index to filter qualifying rows to a small number and applies any XML query predicate to these interim results before returning the final result set.

3.2.5 XML schema

XML schema is a language that defines the structure and data constraints of XML instance documents. XML schema is published as a recommendation by W3C. The XML language is also referred to as XML Schema Definition (XSD). An XML schema defines the elements and attributes that can appear in a document. It also define the parent and children relationship between elements, the data types, and values for elements and attributes. An XML schema can consist of more than one XML schema document.

Example 3-6 is a sample XML schema. The line numbers are not part of the XML schema; they are presented for illustration purposes only.

Example 3-6 A sample XML schema

```
1 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
2   <xsd:element name="employee">
3     <xsd:complexType>
4       <xsd:sequence>
5         <xsd:element name="id" type="xsd:integer"/>
```

```
6         <xsd:element name="name" type="xsd:string"/>
7         <xsd:element name="dateOfBirth" type="xsd:date"/>
8     </xsd:sequence>
9 </xsd:complexType>
10 </xsd:element>
11 </xsd:schema>
```

Line 1 shows the namespace prefix `xds` in the root element in the XML document. In this example, the prefix is `xds`, but it can be an arbitrary name. A *namespace* is a set of names that can be used as element and attribute names in an XML document.

XML namespaces provide a mechanism to qualify an attribute, and an element name to avoid the naming conflict in XML documents. For example, if a health insurance company receives insurer information from a different company as an XML document, it is quite possible that two or more companies have the same element name defined, but representing different things in different formats. Qualifying the elements with a namespace resolves the name-conflicting issue.

Line 2 declares an element named `employee`. In XML schema, you must have a name and its data type to define an *element*. Once a data type is defined, the instance element in the XML instance document can only have the value of the data type defined in the XML schema. An element can be defined as either a simple type or complex type.

A *simple type element* cannot contain any elements or attributes. A simple element has the format:

```
<xs:element name="element name" type="data type"/>.
```

The following two elements, defined in lines 5 and 6 of Example 3-6 on page 49, are simple elements. The types `xsd:string` and `xsd:integer` are XML schema pre-defined data types:

```
<xsd:element name="id" type="xsd:integer"/>
<xsd:element name="name" type="xsd:string"/>
```

A *complex type element* can contain elements or attributes. The element `employee`, defined in line 3 of Example 3-6 on page 49, is a complex type. It contains three other elements: `id`, `name`, and `dateOfBirth`. In this example, the definition of the complex type is inside of the element `employee`. This type is called a *local complex type*, and it can only be used to define the element and its children elements.

If you want a complex data type that can be used to define other elements, you can declare the complex data type globally, as shown in Example 3-7. The complex type `employeeType` is declared globally; it is not inside the `employee` element and can be used to define other elements in the same schema document. It also can be used in other schema documents if the other schema documents include or import it.

Example 3-7 Globally declare complex type `employeeType`

```
<xsd:complexType name="employeeType">
  <xsd:sequence>
    <xsd:element name="id" type="xsd:integer"/>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="dateOfBirth" type="xsd:date"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="employee" type="employeeType"/>
```

Just as with any XML documents, the XML schema documents have to be well-formed. You can use an XML schema to validate XML instance documents. A good XML schema should correctly describe business concepts.

3.2.6 XML schema design

A good schema design is crucial for managing XML data successfully. *Unified Modeling Language* (UML) is a visual design language for XML. You can also use UML to explain XML schemas to someone who does not understand XML technology.

UML modeling

Unified Modeling Language (UML) is an industry standard for modeling business concepts. It is an object-orientated language. UML is one of the import modeling languages that can assist you in building XML schemas. You can use UML to represent business concepts in graphic notions, and you can easily turn these graphic notions into XML schemas.

In a UML diagram, a box represents a business concept or a class. A line represents relationship. You can turn a sentence into UML diagram. Figure 3-4 shows a UML diagram representing the sentence *Loan officer approves a loan*. In the Loan officer box, the pertinent information regarding Loan officer is identified. The Loan box identifies the loan-related information. This information often becomes the attributes in an XML schema. The line, Approve, identifies the relationship between Loan officer and Loan.

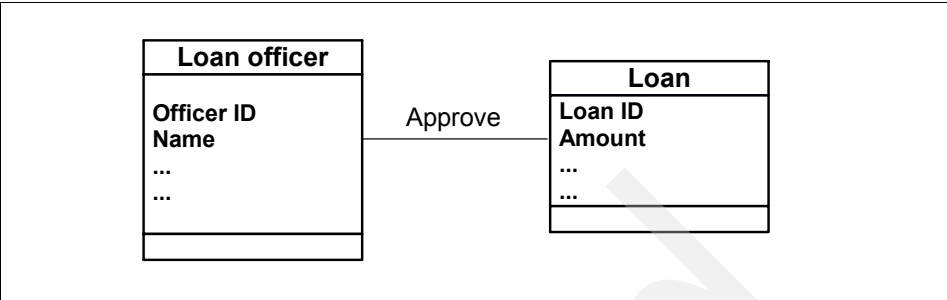


Figure 3-4 UML diagram representing “loan officer approves loan”

We used UML in the schema design process for our sample application, XMLoan, described in Chapter 2, “Sample scenario description” on page 21. We started by gathering information about the loan application business process and studying the hard copy of the loan application form. For a loan application, the customer must provide their personal and financial information. This data should be able to be correlated to the existing account in the bank, if any. The bank also wants to perform market analysis. We then produce a simple loan application UML diagram as shown in Figure 3-5.

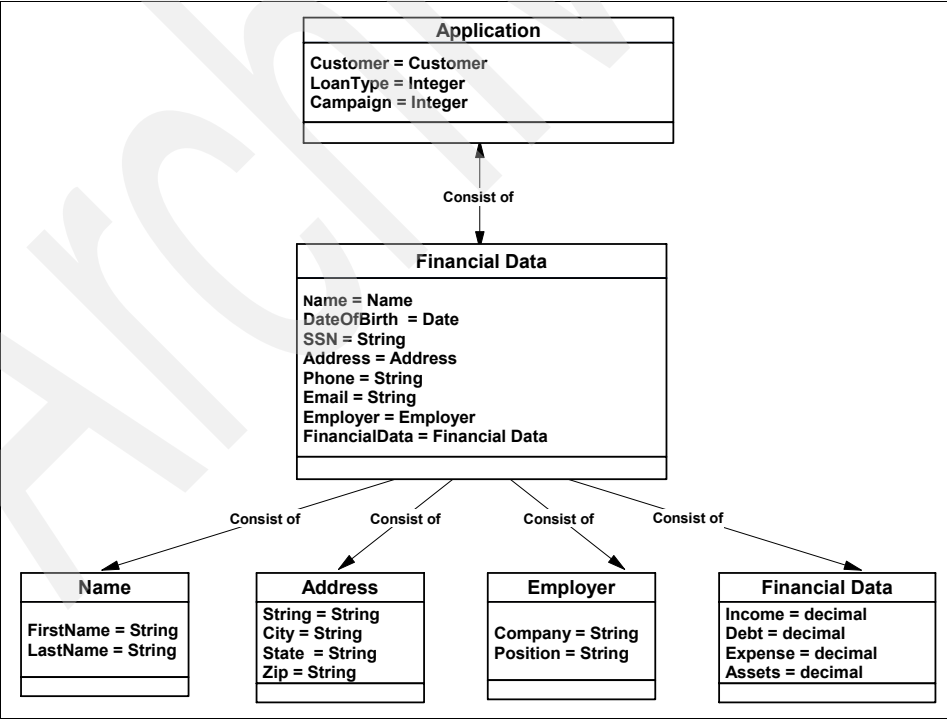


Figure 3-5 UML diagram

Figure 3-5 describes business concepts and relationships as follows:

- ▶ *Application* consists of customer, loan type, and campaign.
- ▶ *Customer* consists of name, date of birth, social security number, address, phone, e-mail, employer, and financial data.
- ▶ *Name* consists of first name and last name.
- ▶ *Address* consists of street, city, state, and zip code.
- ▶ *Employer* consists of company and position.
- ▶ *Financial data* consists of income, debt, expenses, and assets.

From the business process point of view, a loan officer or bank manager can read the UML diagram as a loan application form separated into different sections. They can identify easily if any information is missing. For an application developer or DBA, the UML diagram can be easily transformed into an XML schema. Example 3-8 is the schema mapped from Figure 3-5.

Example 3-8 application.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:include schemaLocation="complextypes.xsd"/>
  <xsd:element name="application">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="customer" type="Customer"/>
        <xsd:element name="loanType" type="xsd:integer"/>
        <xsd:element name="campaign" type="xsd:integer"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

The Application class in Example 3-8 is represented by a complex data type, which consists of three elements: customer, loanType, and campaign. The elements loanType and campaign have the built-in data type, string. The element customer has a complex data type Customer. Notice that XML is case-sensitive; Customer (capitalized) is different from customer.

In this case, Customer with capital C is the complex data type and customer is the element. The complex type Customer is not defined in this schema document. The expression "<xsd:include schemaLocation='complextypes.xsd'/" means to include another XML document complextypes.xsd. The definition of complex data type Customer is in the schema document complextypes.xsd, as shown in Example 3-9.

Example 3-9 Complextype.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="Customer">
    <xsd:sequence>
      <xsd:element name="name" type="Name"/>
      <xsd:element name="dateOfBirth" type="xsd:date"/>
      <xsd:element name="ssn" type="xsd:string"/>
      <xsd:element name="address" type="Address"/>
      <xsd:element name="phone" type="xsd:string"/>
      <xsd:element name="email" type="xsd:string"/>
      <xsd:element name="employer" type="Employer"/>
      <xsd:element name="financialData" type="FinancialData"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Name">
    <xsd:sequence>
      <xsd:element name="firstName" type="xsd:string"/>
      <xsd:element name="lastName" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  <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="zip" type="xsd:integer"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Employer">
    <xsd:sequence>
      <xsd:element name="company" type="xsd:string"/>
      <xsd:element name="position" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="FinancialData">
    <xsd:sequence>
      <xsd:element name="income" type="xsd:decimal"/>
      <xsd:element name="debt" type="xsd:decimal"/>
      <xsd:element name="expenses" type="xsd:decimal"/>
      <xsd:element name="assets" type="xsd:decimal"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

The complex data type Customer has eight elements. The element name has the complex data type Name. The element address has the complex data type Address. The element employer has the complex data type Employer. The element financialData has the complex data type FinancialData. The rest of the elements of the Customer complex data type are built-in data types. The complex data types Name, Address, Employer, and FinancialData are also defined in this XML schema document.

Example 3-10 shows an instance document of the application schema.

Example 3-10 Application.xml

```
<?xml version="1.0"?>
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\application.xsd">
  <customer>
    <name>
      <firstName>Smith</firstName>
      <lastName>John</lastName>
    </name>
    <dateOfBirth>1967-02-23</dateOfBirth>
    <ssn>123-45-6789</ssn>
    <address>
      <street>46 East Main Street</street>
      <city>Los Gatos</city>
      <state>CA</state>
      <zip>95030</zip>
    </address>
    <phone>234-567-8901</phone>
    <email>smith.john@my.com</email>
    <employer>
      <company>My company</company>
      <position>Developer</position>
    </employer>
    <financialData>
      <income>5000</income>
      <debt>10000</debt>
      <expenses>30000</expenses>
      <assets>200000</assets>
    </financialData>
  </customer>
  <loanType>10</loanType>
  <campaign>20</campaign>
</application>
```

If you had not started building the XML schema by drawing the UML diagram first, it would be tricky to identify all object types and the relationships among them. The UML diagram provides not only the big picture, but also the detail of the business model. The UML diagram can also be used to explain the schema to people who are not familiar with XML schemas, such as the loan department manager and the loan officer. The UML diagram can be easily mapped into an XML schema.

When designing a schema, you also want to consider what business data should be kept together in a single XML document. The XML document granularity does impact performance. For more details, refer to *15 best practices for pureXML performance in DB2 9* on the IBM developerWorks Web site:

<http://www-128.ibm.com/developerworks/db2/library/techarticle/dm-0610nicola/>

3.2.7 Industry standards and XML schemas

Corporations in various industries receive, process, store, and send data every day. The formats of the data can differ among corporations, and even among departments in the same corporation. If two corporations want to exchange data, they have to convert it from one format to another. Because data conversion is usually costly and time-consuming, it would be helpful to have standards for that exchange.

Industry standards exist that define agreed-upon ways to exchange information between and within companies. Developing your applications using a standard XML schema and DTD ensures that data exchange can take place between corporations or within a company without experiencing problems. In the following sections we list the URLs for some of these standards-defining organizations.

Financial services industry

In the financial services area, industry standards include:

► ACORD:

The Association for Cooperative Operations Research and Development (ACORD) develops and maintains various electronic standards for the insurance, reinsurance, and related financial services industries. ACORD standards encompass Life and Annuity, Property and Casualty/ Surety and Reinsurance industry segments. For more information about ACORD, refer to:

<http://www.acord.org/>

► FpML:

Financial products Markup Language (FpML) is the business information exchange standard for electronic dealing and processing of financial derivatives instruments. It establishes a new protocol for sharing information, and dealing in swaps, derivatives, and structured products. For more information about FpML, refer to:

<http://www.fpml.org/>

► FIXML:

The Financial Information eXchange™ (FIXML) protocol is a messaging standard developed specifically for the real-time electronic exchange of securities transactions. For more information, refer to the Web site:

<http://www.fixprotocol.org/>

► MISMO:

The Mortgage Industry Standards Maintenance Organization, Inc. (MISMO) was established by the Mortgage Bankers Association (MBA) to coordinate the development and maintenance of Internet-based Extensible Markup Language (XML) real estate finance specifications. MISMO utilizes an open and democratic vendor-neutral approach to the development and maintenance of a single real estate finance XML DTD transaction repository. MISMO has published specifications that support mortgage insurance applications, mortgage insurance loan boarding, secondary, bulk pricing, real estate services, credit reporting, and underwriting process areas. For more information, refer to the Web site:

<http://www.mismo.org/default.html>

► XBRL:

eXtensible Business Reporting Language (XBRL) is a language for the electronic communication of business and financial data. It provides major benefits in the preparation, analysis, and communication of business information. For more information, refer to the Web site:

<http://www.xbrl.org>

► IFX:

The Interactive Financial eXchange (IFX) is an XML-based, financial messaging protocol. It is an object model represented by an XML Schema, a communications protocol with well-considered business rules, and a Targeted Financial Solution. For more information, refer to the Web site:

<http://www.ifxforum.org/standards/>

Other industry standards

Several other industry standards are available for various business sectors such as tax, health care, publishing, and sports. Following are some of those standards:

► Tax XML:

Tax XML is an initiative to research and analyze personal and business tax reporting and compliance information, represented in XML, to facilitate interoperability in a way that is open, flexible, and international in scope. For more information, refer to the Web site:

http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tax

► HL7:

Health Level Seven (HL7) is one of several American National Standards Institute (ANSI) accredited Standards Developing Organizations (SDOs) operating in the health care arena. HL7's domain is clinical and administrative data. For more information, refer to the Web site:

<http://www.hl7.org/>

► ARTS:

The Association for Retail Technology Standards (ARTS) is an international membership organization. RT is devoted to reducing the costs of technology through standards. AETS has four standards: The Standard Relational Data Model, UnifiedPOS, IXRetail, and the Standard RFPs. Association for Retail Technology Standards (ARTS) and International XML Retail Cooperative (IXRetail) XML schemas for use by retailers provides retailers the ability to create, deliver, and archive digital receipts using in-store point-of-sale transaction data. For more information, refer to the Web site:

<http://www.nrf-arts.org/>

► NewsML:

News Markup Language (NewsML) is an XML-based standard to represent and manage news throughout its life cycle, including production, interchange, and consumer use. NewsML 1.0 was approved by IPTC (International Press Telecommunications Council) in October 2000. The current version is NewsML 1.3, which was published in October 2003. For more information, refer to the Web site:

http://www.newsml.org/pages/spec_main.php

► SportsML:

Sports Markup Language (SportsML) is an XML-based standard for the interchange of sports data and statistics. The current release of SportsML is Version 1.0, which was ratified by the IPTC (International Press Telecommunications Council). For more information, refer to the Web site:

<http://www.sportsml.com/specifications.php>

► XPRL:

eXtensible Public Relations Language (XPRL) is an XML-based standard that is being designed for use in the public relations sector. It is an open initiative, which developers can use to create business-oriented programs for PR. XPRL defines how computer data relating to PR campaigns is stored and shared across the Internet. For more information, refer to the Web site:

<http://www.xprl.org/>

► PhotoML:

Photo Markup Language (PhotoML) is a standard for describing the details of photo creation, processing, and content in a collection of photographs. It can be used for a wide variety of photographic formats, including roll film (such as 35mm and 120/220), sheet film (such as 4x5 and 8x10) and digital images. For more information, refer to the Web site:

<http://www.wohlberg.net/public/software/photo/photoml/>

► ThML:

Theological Markup Language (ThML) is an XML-based standard that is being used to mark up texts for the Christian Classics Ethereal Library and other projects. For more information, refer to the Web site:

<http://www.ccel.org/ThML/>

► XBITS:

XML Book Industry Transaction Standards (XBITS) is a Working Group of IDEAlliance that is designing an XML-based standard to facilitate bidirectional electronic data exchanges between publishers, printers, paper mills, and component vendors. For more information, refer to the Web site:

<http://www.idealliance.org/xbits/>

► CBML:

Comic Book Markup Language (CBML) is a TEI-based XML vocabulary (with DTD and schema representations) designed to accommodate the XML encoding of comic books and graphic novels. For more information, refer to the Web site:

<http://www.cbml.org/>

► GJXDM:

The Global JXDM (GJXDM) is an XML standard for criminal justice information exchanges, providing law enforcement, public safety agencies, prosecutors, public defenders, and the judicial branch with a tool to share data and information in a timely manner. For more information, refer to the Web site:

<http://it.ojp.gov/jxdm/3.0/index.html>

IT standards

Following are some IT standards used across the industries:

- Web Services provide a way of describing and publishing a general purpose and agreed interface for accessing data and applications, through the Web Services Description Language (WSDL) notation. The Web Services approach provides loose coupling between clients and the data or applications being accessed and is important for enabling service-oriented architecture (SOA).
- Atom (and RSS) provide an agreed way for publishing summaries of changes to data and for interested parties to easily locate these summaries easily. Atom also makes it possible for general-purpose software readers to offer a human or programmatic interface to subscribe to changes, to be notified when the changes happen, and to review the changes. RSS is similar to Atom, except it has not been standardized and thus has many variants.
- XForms is an agreed way to enable a Web forms interface. An XForm can load external XML documents as initial data in the browser, and can submit the results to the server as XML. By including the browser in the XML pipeline through XFORMS, it means that you can have end-to-end XML, right up to the user's desktop. This eliminates data conversions, thereby reducing processing overhead.

3.2.8 XML data validation

DB2 9 supports validating XML documents with XML schema at insert or import time. The XML schemas have to be registered in XML Schema Repository (XSR) before it can be used for validation. We discuss XSR more in 5.2, “Schema management” on page 198. The XML document in the same XML type column can be validated by a different XML schema of your choice in the insert/import time. You can also choose not to validate the XML. In general, there are three choices:

- Validate on the server.
- Validate within an application.
- Do not validate.

Validation on the server

DB2 9 supports validation on the server. You should validate incoming XML documents on the server if the incoming XML documents must be valid, but the XML documents are from an untrusted source. For instance, suppose that a mortgage company has received an XML document of an application from a mortgage broker. All brokers develop their own applications to fill out the application forms, and to generate XML documents. Because the mortgage company have no control over these applications, and the applications might or might not follow the industrial standard XML schemas, the incoming XML documents are considered as being from an untrusted source. The mortgage company must ensure that all XML documents are valid, and can validate all XML documents on the server side at insert and import time.

The XMLVALIDATE function is used for validating XML documents with an insert statement in an application. The XMLVALIDATE function checks XML documents against the specified XML schema and makes sure that the XML document satisfies the constraints in the XML schema. Details about how to validate at import time are discussed in 5.3, “IMPORT, EXPORT, and RUNSTATS” on page 208. When validating XML documents, the schema information passed to the XML validate function can be either explicitly passed (*explicit validation*) or implicitly inferred from the XML document (*implicit validation*).

Explicit validation

Explicit validation means that the information of a precise XML schema is explicitly specified in XMLVALIDATE function. There are two ways to specify the information:

- ▶ Use the namespace and the schema location of the XML schema.
- ▶ Use an SQL identifier.

Example 3-11 is a primary XML schema document that has been registered with the SQL identifier *sample.pets*.

Example 3-11 XML schema pets.xsd

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.itso.org/pets"
xmlns:pe="http://www.itso.org/pets" xmlns:ca="http://www.itso.org/cat"
xmlns:do="http://www.itso.org/dog">
<xs:import namespace="http://www.itso.org/cat" schemaLocation="cat.xsd"
/>
<xs:import namespace="http://www.itso.org/dog" schemaLocation="dog.xsd"
/>
  <xs:element name="PETS">
    <xs:complexType>
```

```

        <xs:sequence>
            <xs:element name="DOG" type="do:DOG"/>
            <xs:element name="CAT" type="ca:CAT"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:schema>

```

Example 3-12 shows an XML document, `pets.xml`, which requires validation during insert.

Example 3-12 XML document `pets.xml`

```

<?xml version="1.0"?>
<pe:PETS xmlns:pe="http://www.itso.org/pets"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.itso.org/pets pets.xsd">
    <DOG>
        <NAME>SPOT</NAME>
        <AGE>2</AGE>
    </DOG>
    <CAT>
        <NAME>TOM</NAME>
        <AGE>1</AGE>
    </CAT>
</pe:PETS>

```

Example 3-13 shows explicitly validating an XML document with an SQL identifier. `<XML document>` means that the content of the XML document `pets.xml`. `sample.pets` is the SQL identifier of the registered schema.

Example 3-13 Explicit validation using SQL identifier

```

insert into test values xmlvalidate (xmlparse(document'<XML document>'
Preserve whitespace) ACCORDING TO XMLSCHEMA ID sample.pets)

```

An XML schema can contain one or more XML schema documents. One of these XML documents must be the primary schema, which is at the top of the hierarchy. In our example, `pets.xsd` is the primary with two imported schema, `cat.xsd` and `dog.xsd`. When you use the namespace and the schema location of the XML schema to validate the XML document, the namespace and the schema location you specified must match the namespace and the schema location of the XML schema primary document.

Example 3-14 shows explicitly validating the XML document with the use of namespace and the schema location of the XML schema. `http://www.itso.org/pets` is the namespace and `http://sample` is the schema location. `<XML document>` means the content of the XML document `pets.xml`.

Example 3-14 Explicit validation using namespace

```
insert into test values xmlvalidate (xmlparse(document'<XML document>'
Preserve whitespace) ACCORDING TO XMLSCHEMA URI
'http://www.itso.org/pets' LOCATION 'http://sample')
```

Which one do you use, the SQL identifier or the namespace and the schema location of the XML schema? Both have the same performance and provide the same functionality. You can choose one over the other, depending on how you design the database and application. You might have the same schema that registers in different databases in different names. If you use the namespace and the schema location of the XML schema, you do not have to change your application to different SQL identifiers for the same schema. Sometimes, the SQL identifier is more convenient, if your application only accesses one database and you know the exact SQL identifier that associates to the schema you want to use.

Implicit validation

Implicit validation means that the schema information is not passed by the INSERT or IMPORT statement. The schema hints are from the XML document that is inserted or imported. The schema hints are used to find the specific schema to validate the XML document.

The schema hints are specified in the following attributes in the XML document:

- ▶ *xsi:schemaLocation* has two values, the namespace and the schema location to the namespace. If the XML document does not have a namespace, *xsi:noNamespaceSchemaLocation* would be used.
- ▶ *xsi:noNamespaceSchemaLocation* has only one value that is the schema location.

When using implicit validation, DB2 9 searches the catalog tables using the values specified in the schema hints and finds the XML specific schema to validate the XML document.

We use a simple example to demonstrate implicit validation. Example 3-15 shows an XML schema document `person.xsd`, which we use to implicitly validate the XML instance document `person.xml`.

Example 3-15 Person.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://person" xmlns:per="http://person">
  <xsd:element name="person">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="name" type="xsd:string" />
        <xsd:element name="age" type="xsd:integer" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Whether explicit or implicit validation is being done, the schema used for validation must be registered in the database. Example 3-16 shows the commands to register schema person.xsd.

Example 3-16 Register schema

```
register xmlschema http://person from c:\person.xsd as john.person
complete xmlschema john.person
```

Example 3-17 shows an XML document person.xml to be inserted into the table TEST using implicit validation with schema person.xsd. The actual data in person.xml is irrelevant. We are interested in the attribute xsi:schemaLocation, which contains the schema hints. It has the first value http://person as the namespace, and the second value http://person as the schema location. Our example shows that the namespace and the schema location have the same value, but they do not have to be the same.

Example 3-17 Person.xml

```
<?xml version="1.0"?>
<per:person xmlns:per="http://person"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://person http://person" >
  <name>John Doe</name>
  <age>36</age>
</per:person>
```

Example 3-18 shows the command to implicitly validate with the schema hints in schemaLocation in the XML document during the insert time. <XML document> means the content of the XML document person.xml.

Example 3-18 Implicit validation

```
insert into test values xmlvalidate (xmlparse(document'<XML document>'
preserve whitespace))
```

If the XML document is valid, it will be inserted into the TEST table. If it is not valid, the insert fails with error code SQL16206N. When you do implicit validation, you do not use the `ACCORDING TO XMLSCHEMA` clause. If you do, it becomes explicit validation. DB2 will use the information provided by the `ACCORDING TO XMLSCHEMA` clause, and the schema hints in the XML document are ignored.

In implicit validation, DB2 searches the catalog table for the pair value provided by `xsi:schemaLocation` in order to find the correct schema. In our simple example, DB2 finds the schema `john.person` by searching the catalog tables using the pair value `http://person` and `http://person`.

Explicit validation versus implicit validation

We have discussed both explicit validation and implicit validation with examples. You now must be asking which one to use, explicit validation or implicit validation? It depends on the source of the XML documents, the nature of your applications, and your business rules.

The main difference between explicit validation and implicit validation is where the schema information (hints) are provided. For explicit validation, schema information is provided by the `ACCORDING TO XMLSCHEMA` clause embedded in the insert statement. For implicit validation, schema hints are provided by the inserting XML document. If the XML documents are from a trusted source and the XML document knows its schema, implicit validation can be a good choice. If the XML document's content is unknown, explicit validation is a good choice.

For explicit validation, only one exact schema can be used to validate. For implicit validation, one or more schemas can be used to validate. In some situations, you have to use more than one schema to validate an XML document. For instance, suppose that a SOAP message contains a header and a body. The header may have a schema and the body may have another schema. Implicit validation is required for the SOAP message, because it has to have more than one schema.

In general, explicit validation might have better performance than implicit validation. This is not because the actual explicit validation takes less time than the actual implicit validation. It is because the implicit validation requires DB2 to search the catalog tables to find the correct schemas that match the pair value. Explicit validation does not require searching the catalog tables.

Validation within an application

In some situations, you might want to do validation within the application. For example, suppose that you have an on-line loan application program. Customers can log on and fill out their loan application on-line forms. You have a name field, birthday field, phone number field, and other fields in the on-line form. If the customer enters information that does not satisfy the schema constraints, such as entering character data into a numeric field, you want to validate the XML document on the client/application side. If the XML document is not valid, the application can interactively ask the customer to correct the entered data until the XML document is valid.

Like any computer algorithm, validation takes resources such as CPU and memory. Imagine that there are thousands of clients that insert massive XML documents to a server and the validation is done on the one server. Sometimes, a server does not have enough resources, and the clients have the necessary bandwidth. In this case, it can be a good idea to validate on the client.

No validation

If the XML documents are from a trusted source, there is no necessity to validate. Suppose that a bank develops an application for its branches. The application is guaranteed to generate valid XML documents, therefore no validation is required.

Sometimes, you do not care if XML documents are valid or not. In this case, the validation is also not required.

Note: Even if you do not validate documents, you can only insert well-formed XML documents into an XML column in DB2 9.

3.3 Physical database design

In this section, we discuss the considerations of physical design pertinent to pureXML. For the physical design on relational databases, see *Administration Guide: Planning*, SC10-4223.

Space estimation for XML data type

DB2 9 stores XML data as an XML data type in a hierarchical structure. This pureXML storage model provides the flexibility in storing XML documents with various XML schema in same column. The variation of XML data in each row produces a challenge in estimating the storage required for XML data.

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

► Document structure:

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

► The number of elements:

The pureXML storage model stores all XML data in a hierarchical form as a tree structure, so besides the real data, it requires extra storage, which is used to describe the tree structure. For example, every node in the tree has to store the links to its child nodes and to its parent node. The more complex tree requires more extra storage for the tree structure information. For every element in an XML document, DB2 9 will allocate a small structure to store the element information. There is no data stored in this structure.

► Node names:

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

► Ratio of attributes to elements:

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

► Document code page:

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

► Document validation:

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

To calculate an XML document manually, add up the size of every element, attributes, and actual data. Use this simple XML document as an example:

```
<Customer>
  <Name>John Smith</Name>
  <Phone>408-404-1212</Phone>
</Customer>
```

The size of this document will be the total of the length of <Customer>, </Customer>, <Name>, </Name>, "John Smith", <Phone>, </Phone>, and "408-404-1212". For each element, DB2 uses a few bytes to store element information. Because XML document lengths vary, it is not practical to calculate each document's size and use it as the base to estimate the space required.

A better way to estimate the XML data space required is by sampling your XML documents with a typical size and storing the samples into the database. You can then use the administrative table function `admin_get_tab_info` to check how much space the sample data takes. Following is the `SELECT` statement for the XML object size:

```
SELECT t.xml_object_l_size, t.xml_object_p_size,  
       t.data_object_l_size, t.data_object_p_size,  
       t.index_object_l_size, t.index_object_p_size  
FROM TABLE(admin_get_tab_info('schemaname','tablename')) as t
```

You can project the storage size of the sample to your actual XML document amount. The accuracy of the estimating depends on the sampling and the amount of samples. Note that the monitor element used by the administrative function `admin_get_tab_info` does not take free pages or free page management into account. It just reports the number of pages on disk.

Physical storage design considerations

In pureXML storage, XML is treated as a first-class data type. This means that you can use the XML type in Data Definition Language (DDL) statements, stored procedures, and functions, including XML publishing functions. In this section, we discuss some considerations for designing physical storage for XML type:

► Page size for XML:

When a large XML document is inserted into an XML column, the XML document is split into regions and pages. There is a regions index that keeps track of the XML document parts that split into pages. How many regions or pages will a document split into? It depends on the size of the document and the page size. Given a fixed-size XML document inserted into an XML column, the larger the page size, the less number of regions and pages. For example, the same XML document takes more regions and pages on a 4K-page size than on an 8K-page size. Fewer regions and pages per document are better for performance. You should choose the page size depending on the size of your XML documents. If performance is your only consideration, the larger page size is better.

► Table spaces for XML:

In general, database managed space (DMS) table spaces have better performance than System Managed Space (SMS) table spaces. This is because, unlike SMS table spaces, DB2 can directly access DMS table spaces without going through the operating system. When you create a table with XML columns, you can place XML data and indexes in separate table spaces to use different page sizes and separate configuration parameters, such as prefetch size. Example 3-19 shows creating a table in three different table spaces. The non-long data types are in `tablespace2`. Indexes are in `tablespace3`. XML types are considered to be long data types. XML will go into `tablespace4`.

Example 3-19 Create a table

```
CREATE TABLE xmltable(c1 char(5), c2 int,c3 char(7), c4 XML)
IN tablespace2
INDEX IN tablespace3
LONG IN tablespace4
```

► Buffer pools:

A buffer pool belongs to a single database and can be used by more than one table space. When you assign a buffer pool to a table space, the buffer pool and the table space must have the same page size. If you want to assign a buffer pool to multiple table spaces, all table spaces must have the same page size as the buffer pool page size. Buffer pools reduce disk I/O, therefore buffer pools are crucial for the database performance. The DB2 9 snapshot monitor supports monitoring of XML data in buffer pools. The buffer pool Snapshot™ Monitor has new XML data counters to help you make decisions on how to tune your buffer pools. In order to use the snapshot monitor, you have to turn on the buffer pool switch. Example 3-20 shows the commands to turn on the buffer pool monitoring switch and get the snapshot data.

Example 3-20 Turn on the buffer pool switch

```
UPDATE MONITOR SWITCHES USING bufferpool on
get snapshot for bufferpools on <database name>
```

Example 3-21 shows the counters from the buffer pool snapshot output. For more details on the counters, see the DB2 9 Information Center.

Example 3-21 The new counters for XML data

relational Data Counters	
Buffer pool data logical reads	= 246
Buffer pool data physical reads	= 68
Buffer pool temporary data logical reads	= 132

```

Buffer pool temporary data physical reads  = 0

Relational and XML Index Counters
Buffer pool data writes                    = 16323
Buffer pool index logical reads            = 0
Buffer pool index physical reads           = 0
Buffer pool temporary index logical reads  = 0
Buffer pool temporary index physical reads = 0

XML Data Counters
Buffer pool xda logical reads              = 2921
Buffer pool xda physical reads             = 152
Buffer pool temporary xda logical reads    = 0
Buffer pool temporary xda physical reads   = 0
Buffer pool xda writes                     = 0

```

3.4 Creating a database

XML data is stored in code set UTF-8/code page 1208 in DB2 V9.1. In order to use the XML type, you must create a UTF-8 database. Example 3-22 shows creating a UTF-8 database called xmlrb.

Example 3-22 Create database command

```
create database xmlrb using codeset UTF-8 territory US
```

Through out this book, we discuss the new support for creating database objects such as tables, views, and index. For more information about the features and options for creating database, refer to *Administration Guide: Implementation*, SC10-4221.

DB2 9 has the following new system catalog views for XML indexes and XSR objects:

- ▶ SYSCAT.INDEXXMLPATTERNS:
Each row represents a pattern clause in an index over an XML column.
- ▶ SYSCAT.XDBMAPSHREDTREES:
Each row represents one shred tree for a given schema graph identifier.
- ▶ SYSCAT.XDBMAPGRAPHS:
Each row represents a schema graph for an XDB map (XSR object).

- ▶ SYSCAT.XSROBJECTAUTH:
Each row represents a user or group that has been granted the USAGE privilege on a particular XSR object.
- ▶ SYSCAT.XSROBJECTCOMPONENTS:
Each row represents an XSR object component.
- ▶ SYSCAT.XSROBJECTDEP:
Each row represents a dependency of an XSR object on some other object. The XSR object depends on the object of type BTYPE of name BNAME, so a change to the object affects the XSR object.
- ▶ SYSCAT.XSROBJECTHIERARCHIES:
Each row represents the hierarchical relationship between an XSR object and its components.
- ▶ SYSCAT.XSROBJECTS:
Each row represents an XML schema repository object. *SQL Reference Volume 1*, SC10-4249 has more detailed information about system catalog views.

In Chapter 5, “Managing XML data” on page 173, we discuss in more detail the system catalog view for XML index and XSR objects.

Working with XML

In DB2, you can query XML data in four different ways, using plain SQL, SQL/XML, XQuery, and XQuery with embedded SQL. *XQuery* is a new language supported by DB2. In this chapter, you will learn how to query data stored in XML columns using these languages.

In addition, we introduce the XML Path Language (*XPath*), which is a language used to address portions of an XML document. XPath expressions are often used in XQuery to identify particular parts of XML documents.

In DB2 9, a query can combine expressions from both SQL and XQuery. In this chapter, we discuss each language in detail starting with XPath, followed by XQuery. We then show you the new SQL/XML functions introduced in DB2 9. We discuss each concept and show its use by examples. At the end, we introduce XML full-text search using DB2 Net Search Extender.

We discuss the following topics:

- ▶ XPath
- ▶ XQuery
- ▶ XQuery and SQL/XML
- ▶ When and how to use namespaces
- ▶ Getting XML data in and out of a database
- ▶ XML full-text search

4.1 XPath

XPath 2.0 is an expression language for processing values that conform to the XQuery/XPath Data Model (XDM). XDM provides a tree representation of XML documents. Values in XDM are sequences containing zero or more items, which could be:

- ▶ Atomic values such as integers, strings, or Booleans
- ▶ XML nodes such as documents, elements, attributes, or texts

Example 4-1 shows an XML document, sample.xml. It contains these nodes:

- ▶ **Document node:** This XML document contains one document node, sample.xml.
- ▶ **Comment node:** There is only one comment node, which is a sample XML file.
- ▶ **Element node:** The element nodes are: Customer, Name, FirstName, LastName, Address, Street, City, State, Zip. Both Phone and Email have two occurrences, which make a total of thirteen element nodes.
- ▶ **Attribute node:** The Address and two Phone elements have attribute nodes associated with them. "Country" is the attribute node for Address and "type" is the attribute node for Phone.
- ▶ **Text node:** The ten text nodes and their parent element nodes are:
 - "Steve" - FirstName
 - "Ferrington" - LastName
 - "46 Oak Street" - Street
 - "Los Gatos" - City
 - "CA" - State
 - "95030" - Zip
 - "123-456-7890" - Phone
 - "234-567-8901" - Phone
 - "sfer@yahoo.com" - Email
 - "stevef@gmail.com" - Email.

Example 4-1 sample.xml

```
<!-- sample xml file -->
<Customer>
  <Name>
    <FirstName>Steve</FirstName>
    <LastName>Ferrington</LastName>
  </Name>
  <Address country="US">
    <Street>46 Oak Street</Street>
```

```
<City>Los Gatos</City>
<State>CA</State>
<Zip>95030</Zip>
</Address>
<Phone type="work">123-456-7890</Phone>
<Phone type="home">234-567-8901</Phone>
<Email>sfer@yahoo.com</Email>
<Email>stevef@gmail.com</Email>
</Customer>
```

4.1.1 XQuery/XPath data model

XQuery and XPath expressions transform instances of the XDM and return as results, instances of the same data model. Parsing XML data into the XDM occurs before data is processed by XQuery/XPath. The XML document can be parsed with or without validation.

During model generation, the XML document is converted into an instance of the XDM. An *instance* of the XDM is a sequence. A *sequence* is an ordered collection of zero or more items. An *item* is either an atomic value or a node. Atomic values and nodes can be mixed in any order in a sequence. Sequences cannot be nested. When two or more sequences are combined, the result is a sequence containing all of the items found in the source sequences.

For example, inserting the sequence (`<a/>`, ``, 123) between the two items in sequence ("alpha", 2) results in the sequence ("alpha", `<a/>`, ``, 123, 2). The notation used in the example is consistent with the syntax used to construct sequences in XQuery. The whole sequence is enclosed in parentheses and the items are separated by a comma.

An *atomic value* is an instance of one of the built-in atomic data types that are defined by XML schema. These types are atomic because they cannot be decomposed to simpler types. Atomic data types include integers, decimals, strings, dates, and some other types.

The nodes of a sequences form one or more hierarchies, or *trees*. Each hierarchy consists of a root node and all the nodes that are accessible from it. Every node belongs to only one hierarchy and every hierarchy has exactly one root node.

A node's qualified name (or QName) is composed of two parts: an optional namespace URI and a local name. Lexically it has the following format: `prefix:localName`. In our sample XML document we do not use namespaces. We discuss this topic in 4.4, "When and how to use namespaces" on page 136.

Figure 4-1 shows a simplified representation of the data model for sample.xml. The diagram includes a document node (D), comment node (C), element nodes (E), attribute nodes (A), and text nodes (T). It shows that a node can have other nodes as children forming one or more node hierarchies. For example, the element Address is a child of Customer (or to say this another way, the element Customer is a parent of Address). The Address element has one attribute (country) and four child elements (Street, City, State, and Zip).

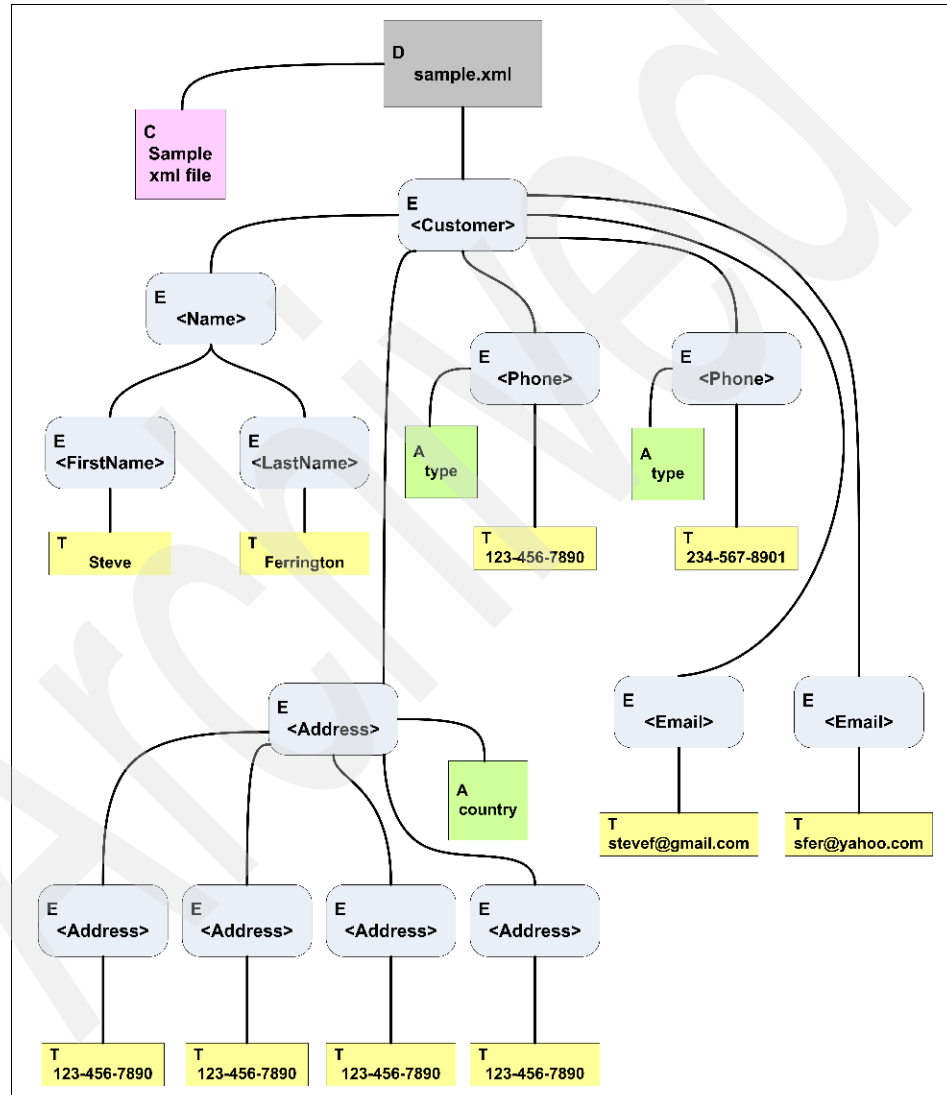


Figure 4-1 Data model diagram for sample.xml

DB2 supports six types of nodes: document, element, attribute, text, processing instruction, and comment. Each type of node has its own set of properties which are different from each other. A node's properties can include its name, its parent node, and its attributes. Table 4-1 lists node properties.

Table 4-1 Node properties

Node property	Description
node-name	The name of the node as a QName
type-name	The dynamic (run-time) type of the node
string-value	A string value that can be extracted from the node
typed-value	A sequence of zero or more atomic values that can be extracted from the node
in-scope namespaces	The in-scope namespaces that are associated with the node
content	The content of the node
attributes	The sequence of attribute nodes that are children of the current node
parent	The node that is parent of the current node
children	The sequence of nodes that are children of the current node
target	The application name as QName

Table 4-2 lists node type supported in DB2 and their properties.

Table 4-2 Node supported in DB2

Node type	Description	Properties
Document node	Encapsulates XML document	string-value typed-value children
Element node	Encapsulates an XML element	node-name type-name string-value typed-value in-scope namespaces attributes parent children

Node type	Description	Properties
Attribute node	Encapsulates an XML attribute	node-name type-name string-value typed-value parent
Text node	Encapsulates XML character content	content parent
Processing instruction node	Encapsulates XML processing instruction	content parent target
Comment node	Encapsulates XML comment	content parent

We can illustrate node types and properties using the sample.xml document. The Address and Names are element type nodes. Here are some properties of the Address node:

- ▶ Node-name: The qualified name of the node is Address.
- ▶ Attributes: This node has one attribute, country.
- ▶ Parent: The parent of the node is the Customer node.
- ▶ Children: This node has four children, Street, City, State, and Zip nodes.

4.1.2 Location paths

Location paths are the most used expressions in XPath. They consist of one or more steps separated by slash(/) or double-slash (//). Each step before the final step produces a sequence of nodes that are used as context nodes for the step that follows. Every step is executed repeatedly, once for every context node that is produced by the previous step. The results of these executions are combined and form the sequence of context nodes for the following step. The value of the location path is the sequence of the items produced from the final step in the path. This sequence can contain only nodes or only atomic values. A location path that generates a mixture of atomic values and nodes results in an error.

The first step defines the starting point of the location path. Often it is a function call or variable reference that returns sequence of nodes. An initial "/" means that the path begins from the root node. An initial "/" means that the path begins with a sequence formed from the root node plus all of its descendants.

Each step can be another axis step or filter expression. An *axis step* consist of three parts: an optional axis that specifies a direction of movement through the XML document or fragment; a node test that defines the criteria used to select

nodes; and zero or more predicates that filter the sequence produced by the step. The result of an axis step is a sequence of nodes, and each node is assigned a context position that corresponds to its position in the sequence. Context positions allow every node to be accessed by its position. Table 4-3 describes the axes supported in DB2.

Table 4-3 Axes supported in DB2

Axis	Description	Direction
self	Returns the context node.	Forward
child	Returns the children of the context node	Forward
descendant	Returns the descendants of the context node	Forward
descendant-or-self	Returns the context node and its descendants	Forward
parent	Returns the parent of the context node	Reverse
attribute	Returns the attributes of the context node	Forward

A *node test* is a condition that must be true for each node that is selected by an axis step. The node test can be either a name test or kind test.

A name test filters nodes based on their names. It consists of a QName or a wildcard and, when used, selects the nodes (elements or attributes) with matching QNames. The QNames match if the expanded QName of the node is equal to the expanded QName in the name test. Two expanded QNames are equal if they belong to the same namespace and their local names are equal. Table 4-4 describes all name tests supported in DB2.

Table 4-4 Name tests supported in DB2

Test	Description
QName	Matches all nodes whose QName is equal to the specified QName
NCName.*	Matches all nodes whose namespace URI is the same as the namespace to which the specified prefix is bound
*.NCName	Matches all nodes whose local name is equal to the specified NCName
*	Matches all nodes

A kind test filters nodes based on their kind. Table 4-5 describes all kind tests supported in DB2.

Table 4-5 Kind test supported in DB2

Test	Description
node()	Matches any node
text()	Matches any text node
comment()	Matches any comment node
processing-instruction()	Matches any processing instruction node
element()	Matches any element node
attribute()	Matches any attribute node
document-node()	Matches any document node

There are two syntaxes for axis steps: unabbreviated and abbreviated. The unabbreviated syntax consist of an axis name and node test that are separated by a double colon (::). In the abbreviated syntax, the axis is omitted by using shorthand notations. Table 4-6 describes abbreviated syntax supported in DB2.

Table 4-6 Abbreviated syntax supported in DB2

Abbreviated syntax	Description
No axis specified	child::, except when the node test is attribute(). In that case omitted axis is shorthand for attribute::.
@	attribute::
//	/descendant-or-self::node()/, except when appears in the beginning of the path expression. In that case the axes step selects the root of the tree plus all nodes that are its descendants
.	self::node()
..	parent::node()

4.1.3 Using location paths to retrieve nodes of an XML document

Here we show you how to use location paths to select different parts of an XML document. We use the sample.xml document as a starting context node for all of our examples. To execute our path expressions, we use DB2.

Setting up DB2 database environment

We use the XMLRB database for our examples (if you have not created it yet, see Chapter 2, “Sample scenario description” on page 21. for instructions):

1. Create an XPS table using the commands in Example 4-2.

Example 4-2 Creating XPS table

```
CONNECT TO xmlrb;  
CREATE TABLE xps(id INTEGER NOT NULL, doc XML);
```

2. Insert our sample XML document into DOC column of XPS table using the command in Example 4-3.

Example 4-3 Inserting sample.xml into xps table

```
INSERT INTO xps (id, doc) VALUES (1, XMLPARSE ( DOCUMENT  
'<!-- sample xml file -->  
<Customer>  
  <Name>  
    <FirstName>Steve</FirstName>  
    <LastName>Ferrington</LastName>  
  </Name>  
  <Address country="US">  
    <Street>46 Oak Street</Street>  
    <City>Los Gatos</City>  
    <State>CA</State>  
    <Zip>95030</Zip>  
  </Address>  
  <Phone type="work">123-456-7890</Phone>  
  <Phone type="home">234-567-8901</Phone>  
  <Email>sfer@yahoo.com</Email>  
  <Email>stevef@gmail.com</Email>  
</Customer>' ));
```

3. We use the template shown in Example 4-4 to execute our path expressions. We will replace the <path_expression> with the actual path expression that is to be executed. The db2-fn:xmlcolumn('XPS.DOC') function returns the sequence of all XML documents stored in the DOC column of the XPS table (this function is discussed in more detail in the next section). We have inserted only one row in the XPS table, so the result of this function call is a sequence containing only sample.xml document, which will be used for all of our examples.

Example 4-4 Template for execution of path expressions

```
XQUERY db2-fn:xmlcolumn('XPS.DOC')<path_expression>;
```

Later in this chapter, we add more data into XPS table. If you have more data in the table and would like to get the same results as shown in our examples, you can replace the `db2-fn:xmlcolumn('XPS.DOC')` with `db2-fn:sqlquery('select DOC from XPS where id = 1')`. The `db2-fn:xmlcolumn` and `db2-fn:sqlquery` functions are described later in this chapter.

Note: XPath is a case-sensitive language. `/Customer` and `/customer` are different.

Executing XQuery

You can execute XQuery using DB2 Command Line Processor (CLP) or DB2 Command Editor. Figure 4-2 shows an XQuery executed using Command Editor.

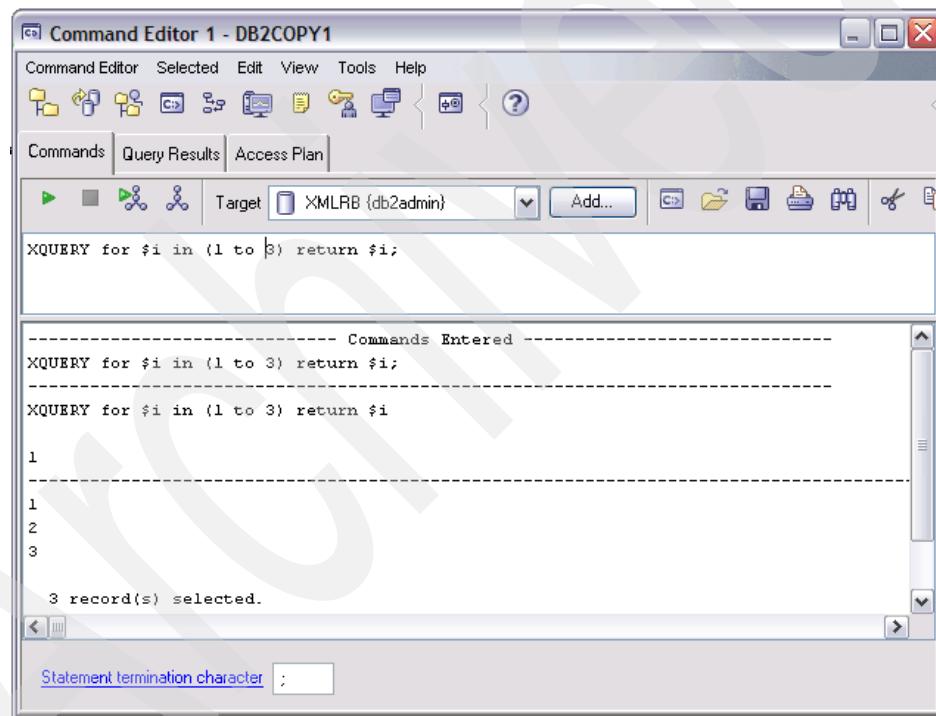


Figure 4-2 Running XQuery using DB2 Command Editor

Example 4-5 shows the same XQuery executed using DB2 CLP.

Example 4-5 Executing XQuery at DB2 CLP

```
E:\SQLLIB\BIN>db2 xquery for $i in (1 to 3) return $i
```

```
1
```

```
-----  
1
```

```
2
```

```
3
```

Because DB2 is not case-sensitive, when executing XQuery using DB2 CLP or Command editor, the key word XQUERY can be lower or upper case. In this chapter, we capitalize the key word XQUERY in our examples for stylistic reasons.

You might have to adjust your DB2 CLP settings in order to display the XML output with all the indents and line spacing.

Location paths examples

In the following sections, we show how location paths can be used for retrieving different parts of an XML document.

Retrieving the whole XML document

Example 4-6 demonstrates how to retrieve the whole XML document.

Example 4-6 Retrieving the whole document (abbreviated syntax)

```
XQUERY db2-fn:xmlcolumn('XPS.DOC')/.;
```

Retrieving the root element of the document

Example 4-7 and Example 4-8 demonstrate how to retrieve the root element of the document. In Example 4-7 we use the name of the root element (Customer). In Example 4-8 we use the node kind test to select the child node of the whole XML document (which is the root element).

Example 4-7 Retrieving the root element using its name

```
XQUERY db2-fn:xmlcolumn('XPS.DOC')/Customer;
```

Example 4-8 Retrieving the root element

```
XQUERY db2-fn:xmlcolumn('XPS.DOC')/node();
```

Retrieving specific elements in a document

Example 4-9 demonstrates how to retrieve the Address element. This type of path expression, where we know the exact path to the element in the document tree, is used very often.

Example 4-9 Retrieving an element using its name and location (abbreviated syntax)

```
XQUERY db2-fn:xmlcolumn('XPS.DOC')/Customer/Address;
```

If we do not know the exact location of the element in the document, instead of specifying the full path (using the child axis), we use the descendant-or-self axis to look at all nodes that are in the hierarchy under the context node.

Example 4-10 demonstrate how to retrieve the Address element without specifying its location in the document.

Example 4-10 Retrieving an element anywhere in an XML document

```
XQUERY db2-fn:xmlcolumn('XPS.DOC')//Address;
```

Retrieving elements with specific locations

It is also possible that we know the location of an element without knowing its name (or want to retrieve all elements with specific location). In this case, we use * name test that matches all elements. Example 4-11 shows how to retrieve all the elements that are children of the Address element.

Example 4-11 Retrieving all child elements of the Address element

```
XQUERY db2-fn:xmlcolumn('XPS.DOC')/Customer/Address/*;
```

Retrieving all elements in a hierarchy

Example 4-12 shows how to retrieve all elements in a document. Here we use // to get all the descendants of the context node (the whole XML document) including the context node itself. Then, using * name test, we select all elements.

Example 4-12 Retrieving all elements

```
XQUERY db2-fn:xmlcolumn('PS.DOC')//*;
```

If we have to retrieve all elements in a hierarchy (instead of in the whole document), we first locate the root of the hierarchy, then retrieve the elements.

We illustrate this with the hierarchy under the Address element as shown in Example 4-13.

Example 4-13 Fragment of sample.xml containing Address element

```
...
  </Name>
  <Address country="US">
    <Street>46 Oak Street</Street>
    <City>Los Gatos</City>
    <State>CA</State>
    <Zip>95030</Zip>
  </Address>
  <Phone type="work">123-456-7890</Phone>
...
```

Example 4-14 shows how to retrieve all elements in a hierarchy including the root of the hierarchy. In Example 4-15 we retrieve all the elements without the root.

Example 4-14 Retrieving hierarchy

```
XQUERY
db2-fn:xmlcolumn('XPS.DOC')//Address/descendant-or-self::element();
```

Example 4-15 Retrieving hierarchy without root element

```
XQUERY db2-fn:xmlcolumn('XPS.DOC')//Address/*;
```

Retrieving text nodes

Often we have to extract text nodes from an XML document. Example 4-16 shows how to retrieve telephone numbers. Text nodes are children of the Phone element.

Example 4-16 Retrieving the phone number

```
XQUERY db2-fn:xmlcolumn('XPS.DOC')/Customer/Phone/text();
```

Example 4-17 and Example 4-18 show how to retrieve text nodes containing the address information for a customer.

Example 4-17 Retrieving the address information (A)

```
XQUERY db2-fn:xmlcolumn('XPS.DOC')//Address//text();
```

Example 4-18 Retrieving the address information (B)

```
XQUERY db2-fn:xmlcolumn('XPS.DOC')//Address/*/text();
```

Retrieving attribute nodes

Example 4-19 shows how to retrieve the attribute node of the Address node (country in our case). Here we use name() function which returns a name of a node. We discuss XPath functions later in this chapter. If we omit it, we will receive an error because in DB2 there is no default rule for serializing an attribute node. We discuss serialization later in this chapter. Note that this example works only when the Address element has no more than one attribute.

Example 4-19 Retrieving the name of attribute node

```
XQUERY  
db2-fn:xmlcolumn('XPS.DOC')/Customer/Address/fn:name(attribute());
```

Example 4-20 shows how to retrieve attribute values. It returns as result values of type attribute for Phone elements in sample.xml. Similar to our previous example, we use a function call. Here the function is string, which transforms a value to string.

Example 4-20 Retrieving the value of attribute node

```
XQUERY db2-fn:xmlcolumn('XPS.DOC')/Customer/Address/fn:string(@type);
```

4.1.4 Predicates

A predicate filters a sequence by keeping only the qualifying items. It consists of a predicate expression that is enclosed in square brackets ([]). The predicate expression is evaluated for each item in the sequence with the selected item as the context item. The result of the evaluation is xs:boolean and is called *predicate truth value*. Only the items for which the predicate truth value is true are retained. All the other items are filtered.

The predicate true value is calculated based on the type of the predicate expression:

- ▶ If the predicate expression returns a numeric value, the predicate truth value is true only for the item that is at the same position in the sequence as the value of the predicate expression.
- ▶ If the predicate returns a nonnumeric value, the predicate truth value is the boolean value of the predicate expression.

The most common use of predicates is to filter the result of path expression based on some criteria. Example 4-21 shows how to retrieve the work phone number. We use a predicate to select only those Phone elements that have a type attribute equal to "work".

Example 4-21 Retrieving the work phone number

```
XQUERY db2-fn:xmlcolumn('XPS.DOC')//Phone[@type="work"]/text();
```

Example 4-22 shows the use of a numeric predicate. It demonstrates how to retrieve the first Email element from sample.xml.

Example 4-22 Usage of numeric predicate

```
XQUERY db2-fn:xmlcolumn('XPS.DOC')/Customer/Email[1];
```

4.2 XQuery

XQuery is a functional language that extends XPath. Its basic building blocks are expressions constructed from keywords, operators (symbols), and operands (that are usually other expressions). Expressions can be nested with full generality. A query is composed of a prologue and a body. The query prologue is optional and consists of declarations that define the execution environment of the query. The query body consists of an expression that provides the result of the query. The input and the output of the query are values (instances) of the XDM.

In Example 4-23 we show a typical XQuery query. It begins with the XQUERY key word followed with a prologue (optional) and a body. The prologue in our example contains default namespace declaration (the second line). The rest of the query is its body. It consists of one or more XQuery expressions.

Example 4-23 Sample XQuery

```
XQUERY
declare default element namespace "http://sample.name.space.com";
for $cust in db2-fn:xmlcolumn('XPS.DOC')
return $cust/Name/LastName;
```

4.2.1 Types, expressions, and functions

In this section, we explain XQuery data types, expressions, and functions. We start with data types because XQuery is a strongly-typed language and every expression and function has its type. We continue with expressions that are the main building blocks of a query. Finally, we describe DB2 XQuery built-in functions.

XQuery data types

XQuery is a strongly-typed language. The operands of every expression, function, or operator must be of their expected types. For example, the 'mod' operator returns a numeric value result and requires that both of its arguments are numeric data type. DB2 XQuery data types include the predefined types of XQuery and built-in types of XML Schema.

The predefined types of XQuery are in the namespace <http://www.w3.org/2005/xpath-datatypes>. This namespace has a predeclared prefix xdt. Some examples of predefined data types are xdt:dayTimeDuration, xdt:yearMonthDuration.

The built-in data types of XML Schema are in the namespace <http://www.w3.org/2001/XMLSchema>. This namespace has a predeclared prefix xs. Some examples of built-in data types are xs:boolean, xs:double, xs:date.

It is expected that xdt data types will be moved in the xs namespace.

Figure 4-3 shows the DB2 XQuery type hierarchy. All the types are derived from xs:anyType. Links in the diagram connect each derived data type with the base type from which it is derived. For example, the xs:long data type is derived from xs:integer data type, which is derived from the xs:decimal data type.

Derived data types can always be used instead of more generic data types. For example, if a function requires an xs:integer type argument, we can use xs:long instead.

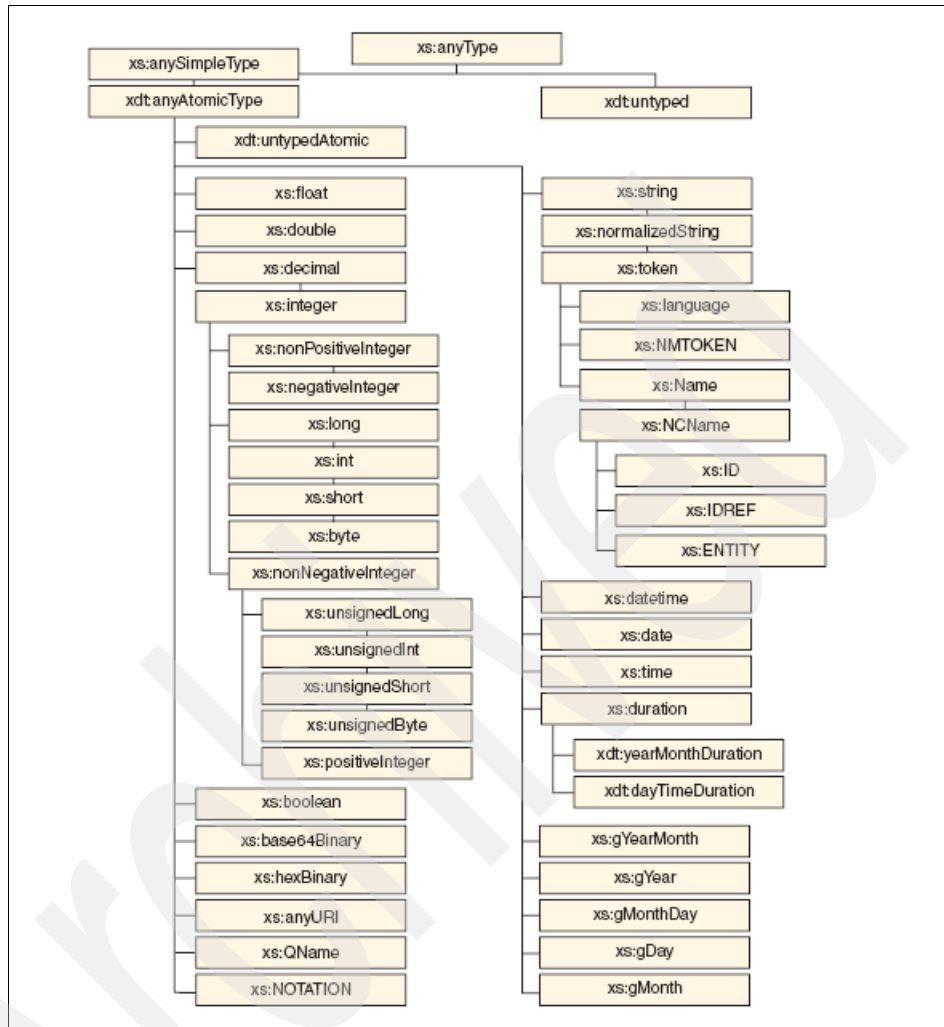


Figure 4-3 DB2 XQuery type hierarchy

There is a constructor function for each of the atomic types which converts a value of one atomic type into an instance of another atomic type. All constructor functions for built-in data types share the same generic syntax:

`type-name(value)`

In this expression, `value` is the value that has to be converted into an instance of the target data type, and the `type-name` is the target data type. Example 4-24 shows an invocation of the constructor function for the `xs:string` atomic data type with an argument of integer data type.

```
xs:string(-123)
```

XQuery expressions

Expressions are the basic building blocks of a query. They can be used alone or in a combination to form complex queries. DB2 supports several kinds of expressions for working with XML data. In our previous section, we discussed path expressions and predicates. Here we present the rest of expression types. The FLWOR expression is presented in the following section.

Primary expressions

Primary expressions are the basic primitive XQuery expressions. They include literals, variable references, parenthesized expressions, context item expressions, constructors, and function calls. Following are some primary expressions:

- ▶ Literals: 12, -134.97, "apple"
- ▶ Variable references: \$i, \$phone, \$seq
- ▶ Parenthesized expressions: (27 + 16) * (43 - 18)
- ▶ Context item expressions: (1, 3, 5, 7, 9)[2]
- ▶ Constructors: xs:date("2006-08-26"), xs:string("a b c")
- ▶ Function calls: fn:true(), fn:abs(-7)

Arithmetic expressions

Arithmetic expressions perform addition, subtraction, multiplication, division, and modulus. Table 4-7 describes the arithmetic operators and lists them in order of operator precedence.

Table 4-7 Arithmetic expressions

Operator	Purpose
- (unary), + (unary)	Negates value of operand.
*, div, idiv, mod	Multiplication, division, integer division, modulus.
+, -	Addition, subtraction.

Comparison expressions

XQuery provides three kinds of comparison expressions: value comparisons, general comparisons, and node comparisons.

Value comparisons compare two atomic values. The operands must be of the same type, or one of them has to be a subtype of the other. The result of value comparison is Boolean. Table 4-8 lists the value comparison operators in DB2.

Table 4-8 Value comparison operators

Operator	Description
eq	Returns true if the first value is equal to the second value.
ne	Returns true if the first value is not equal to the second value.
lt	Returns true if the first value is less than the second value.
le	Returns true if the first value is less than or equal to the second value.
gt	Returns true if the first value is greater than the second value.
ge	Returns true if the first value is greater than or equal to the second value.

Example 4-25 shows an expression containing value comparison.

Example 4-25 Value comparison expression

/Customer/Name[FirstName eq "Steve"]/LastName/text()

General comparisons compare two sequences of any length. The result is Boolean. General comparisons return true if at least one item in the first sequence and one item in the second sequence satisfy the comparison. Each of the items is atomized before comparison. Table 4-9 lists general comparison operators.

Table 4-9 General comparison operators

Operator	Description
=	Returns true if some value in the first sequence is equal to some value in the second sequence.
!=	Returns true if some value in the first sequence is not equal to some value in the second sequence.
<	Returns true if some value in the first sequence is less than some value in the second sequence.
<=	Returns true if some value in the first sequence is less than or equal to some value in the second sequence.
>	Returns true if some value in the first sequence is greater than some value in the second sequence.
>=	Returns true if some value in the first sequence is greater than or equal to some value in the second sequence.

Table 4-10 lists the results of comparing (1, 2) and (2, 3) sequences.

Table 4-10 Results of sample general comparisons

Expression@	Result	Comments
(1, 2) = (2, 3)	true	2 = 2
(1, 2) != (2, 3)	true	1 != 2
(1, 2) < (2, 3)	true	1 < 2
(1, 2) <= (2, 3)	true	1 <= 2
(1, 2) > (2, 3)	false	neither 1 or 2 greater than 2 or 3
(1, 2) >= (2, 3)	true	2 >= 2

Note that if we add 4 to the first sequence, all general comparisons will return true.

Node comparisons compare the position of two nodes in the document order. The result of comparison is Boolean. Table 4-11 lists the node comparison operators.

Table 4-11 Node comparison operators

Operator	Description
is	Returns true if the two nodes have the same identity.
<<	Returns true if the first operand node precedes the second operand node in document order.
>>	Returns true if the first operand node follows the second operand node in document order.

Logical expressions

Logical expressions use *and* and *or* operators. Their arguments are of Boolean type and they return results in boolean value. Table 4-12 lists logical operators. The and operator is with higher precedence than or operator.

Table 4-12 Logical expression operators

Operator	Description
and	Returns true if both arguments are true.
or	Returns true if at least one of arguments is true.

Constructors

Constructors create XML structures inside a query. There are two types of constructors: direct and computed.

Direct constructors create XML structures within query using XML-like notation. Example 4-26 shows direct constructor creating Address element containing an attribute and four child elements. The attribute is country and the child elements are Street, City, State, and Zip. Each of the child elements has an text node as its child.

Example 4-26 Using direct constructor

```
XQUERY
<Address country="US">
  <Street>46 Oak Street</Street>
  <City>Los Gatos</City>
  <State>CA</State>
  <Zip>95030</Zip>
</Address>;
```

```
result:
<Address country="US">
  <Street>
    46 Oak Street
  </Street>
  <City>
    Los Gatos
  </City>
  <State>
    CA
  </State>
  <Zip>
    95030
  </Zip>
</Address>
```

Computed constructors create XML structures within query using enclosed expressions. It starts with a keyword describing the type of node to be created and is followed by the name of the node and its content. Example 4-27 shows creation of the same XML element as in Example 4-26 on page 93 using a computed constructor.

Example 4-27 Using computed constructor

```
XQUERY
element Address {
  attribute country {"US"},
  element Street {"46 Oak Street"},
  element City {"Los Gatos"},
  element State {"CA"},
  element Zip {"95030"}
};

result:
<Address country="US">
  <Street>
    46 Oak Street
  </Street>
  <City>
    Los Gatos
  </City>
  <State>
    CA
  </State>
  <Zip>
    95030
  </Zip>
</Address>
```

Conditional expressions

Conditional expressions evaluate one of two expressions based on whether the value of a test expression is true or false. The structure of a conditional expression is shown in Table 4-13.

Table 4-13 Conditional expression

Expression	Description
if (test_expr) then expr1 else expr2	test_expr is evaluated. If its value is true, the result is the evaluation of expr1, otherwise the result is the evaluation of expr2.

Quantified expressions

Quantified expressions return true or false depending on whether some or every item in one or more sequence satisfies a specific condition. Here are two examples:

```
some $i in (1 to 10) satisfies $i mod 7 eq 0  
every $i in (1 to 5) , $j in (6, 10) satisfies $i < $j
```

The quantified expression begins with a quantifier: *some* or *every*. The quantifier is followed by one or more clauses that bind variables to sequences that are returned by expressions. In our first example, *\$i* is the variable and (1 to 10) is the sequence. In second example, we have two variables *\$i* and *\$j* that are bound to (1 to 5) and (6 to 10). Then we have a test expression in which bound variables are referenced. Test expression is used to determine if some or all of bound variables satisfy a specific condition. In our first example the condition is if *\$i* mod 7 is equal to 0. The qualifier for this expression is *some*, and there is a value for which the test expression is true, so the result is true. In the second example, the condition is if *\$i* is less than *\$j*. The qualifier is *every*, so we check to see if every *\$i* is less than every *\$j*. The result is true.

Cast expressions

Cast expressions are used to transform a value from one type into another type, for example, from string to integer. A cast expression takes two arguments: an input expression and a target data type. It evaluates the expression and attempts to create a new value of the target data type based on the result of the evaluation. Here is an example of cast expression converting string to double:

```
"123.456" cast as xs:double
```

Sequence expressions

Sequence expressions are used to construct, combine, and filter sequence of items. A sequence can be constructed using a comma operator or a range expression.

Using a comma operator, we specify two or more items separated by a comma. Example 4-28 shows using the comma operator to create a sequence containing numbers 1, 3, 5, and 7.

Example 4-28 Creating sequence using comma operator

```
(1, 3, 5, 7)
```

Using the range expression, we create sequences of consecutive integers. We specify the first and the last values, separating them with the *to* operator. Example 4-29 shows a sequence containing numbers 4, 5, 6, and 7 that is created using a range expression.

Example 4-29 Creating sequence using range expression

(4 to 7)

Example 4-30 shows that the comma operator and the range expressions can be combined. The resulting sequence is (1, 2, 3, 1, 7, 9, 11).

Example 4-30 Combining comma operator and range expression

(1 to 3, 1, 7, 9 to 9, 11)

The result of every expression that returns a sequence can be filtered using a predicate. The combination of primary expression and one or more predicates is called a *filtering expression*. Example 4-31 shows creating a sequence containing the numbers 5 and 10 using a filtering expression.

Example 4-31 Filtering expression

(4 to 11)[. mod 5 eq 0]

The information that is available at the time an expression is evaluated is called *dynamic context*. The focus, which consists of the context item, context position, and context size, is an important part of the dynamic context. The focus changes as DB2 processes each item in a sequence. The focus consists of the following information:

- ▶ Context *item* is the atomic value of the node that is currently processed. It can be retrieved using the context item expression, which consists of a single dot (.).
- ▶ Context *position* is the position of the context item in the sequence that is processed. It can be retrieved using the fn:position() function.
- ▶ Context *size* is the number of items in the sequence that is processed.

In Example 4-31, the sequence that is processed is (4, 5, 6, 7, 8, 9, 10, 11). The context size is eight because the sequence contains eight items. The context position of item '5' is two because it is the second item in the sequence. The context position of '10' is seven. During the processing all the nodes in the sequence are iterated one by one and the atomic value of every one of them is processed as a context item. It is referenced in the predicate using a dot (.).

The process of converting the sequence of items into sequence of atomic values is called atomization. Each item in a sequence is converted to an atomic value by applying the following rules:

- ▶ If the item is atomic value, then its value is returned.

- ▶ if the item is a node, then its typed value is returned. The typed value of a node is a sequence of zero or more atomic values that can be extracted from the node. If the node has no typed value, then an error is returned.

Example 4-32 shows the atomization. First, two items, `<a>4` and `5`, in the sequence are nodes. After the atomization they are converted to 4 and 5. The result of the filtering expression is the same as in Example 4-31 on page 96.

Example 4-32 Atomization

```
(<a>4</a>, <b>5</b>, 6 to 11)[. mod 5 eq 0]
```

XQuery functions

DB2 supports a set of built-in functions for working with XML data. These functions include DB2-defined functions and XQuery-defined functions.

DB2-defined functions

There are two DB2-defined functions that are used to access XML data from a DB2 database. They belong to the namespace that is bound to the *db2-fn* prefix.

The *db2-fn:xmlcolumn* function accepts as an argument a name of an XML column in a table or a view and returns as a result a sequence that is the concatenation of the non-null XML values in the specified column. Example 4-33 shows a call to the *db2-fn:xmlcolumn* function. It returns a sequence of all XML documents stored in the column DOC of XPS table.

Example 4-33 Invocation of the db2-fn:xmlcolumn function

```
XQUERY db2-fn:xmlcolumn('XPS.DOC');
```

The *db2-fn:sqlquery* function accepts as an argument string containing fullselect that specify a single-column result set of XML data type and returns as a result a sequence that is the concatenation of the non-null values returned by the specified fullselect. Example 4-34 shows a call to the *db2-fn:sqlquery* function. It returns the same result as Example 4-33.

Example 4-34 Invocation of the db2-fn:sqlquery

```
XQUERY db2-fn:sqlquery('SELECT doc FROM xps');
```

The *db2-fn:sqlquery* function is very important. It allows us to integrate SQL statements inside XQuery. We discuss the usage of *db2-fn:sqlquery* in more detail in 4.3.2, “SQL/XML” on page 127.

XQuery-defined functions

XQuery-defined functions are in the namespace which is bound to the *fn* prefix. This is the default namespace. XQuery defined functions can be invoked without specifying their namespace (unless you want to override the default function namespace).

XQuery-defined functions can be divided into eight different categories based on the type of the data they process, as follows:

- String functions:

Table 4-14 lists the string functions of XQuery-defined functions.

Table 4-14 XQuery-defined string functions

Function	Description
fn:codepoints-to-string	Returns the string equivalent of a sequence of Unicode code points.
fn:compare	Compares two strings. Returns -1, 0, or 1.
fn:concat	Returns a string that is the concatenation of two or more atomic values.
fn:contains	Determines whether a string contains a given substring. Returns true or false.
fn:ends-with	Determines whether a string end with a given substring. Returns true or false.
fn:lower-case	Converts a string to lowercase.
fn:matches	Determines whether a string matches a given pattern. Returns true or false.
fn:normalize-space	Strips leading and trailing whitespace characters from a string and replaces each internal sequence of whitespace characters with a single blank character.
fn:normalize-unicode	Performs Unicode normalization on a string.
fn:replace	Compares each set of characters within a string to a given pattern and then replaces the characters that match the pattern with another set of characters.
fn:starts-with	Determines whether a string begins with a given substring.
fn:string	Returns the string representation of a value.
fn:string-join	Returns a string that is generated by concatenating items separated by a separator character.

Function	Description
fn:string-length	Returns the length of a string.
fn:string-to-codepoints	Returns a sequence of Unicode code points that correspond to a string value.
fn:substring	Returns a substring that occurs in a string.
fn:substring-after	Returns a substring that occurs in a string after the end of the first occurrence of a given search string.
fn:substring-before	Returns a substring that occurs in a string before the first occurrence of a given search string.
fn:tokenize	Breaks a string into a sequence of substrings.
fn:translate	Replaces selected characters in a string with replacement characters.
fn:upper-case	Converts a string to uppercase.

► Boolean functions:

Table 4-15 lists the boolean functions of XQuery defined functions.

Table 4-15 XQuery-defined Boolean functions

Function	Description
fn:boolean	Returns the effective boolean value of a sequence.
fn:false	Returns the xs:boolean value false.
fn:not	Returns true if its argument is false and false if its argument is true.
fn:true	Returns the xs:boolean value true.
fn:zero-or-one	Returns its argument if it is a sequence containing zero or one elements. Otherwise, an error is returned.

► Number functions:

Table 4-16 lists the XQuery-defined number functions.

Table 4-16 XQuery-defined number functions

Function	Description
fn:abs	Returns the absolute value of a numeric value.
fn:avg	Returns the average of the values in a sequence.

Function	Description
fn:ceiling	Returns the smallest integer that is greater than or equal to the argument.
fn:floor	Returns the largest integer that is less than or equal to the argument.
fn:max	Returns the maximum of the values in a sequence.
fn:min	Returns the minimum of the values in a sequence.
fn:number	Converts the value of its argument to xs:double data type.
fn:round	Returns the integer that is closest to the a given numeric value.
fn:round-half-to-even	Returns the integer value with a specified precision that is closest to a given numeric value.
fn:sum	Returns the sum of the values in a sequence.

► Date functions

Table 4-17 lists the XQuery-defined date functions.

Table 4-17 XQuery-defined date functions

Function	Description
fn:current-date	Returns the current date in the implicit time zone of UTC.
fn:current-dateTime	Returns the current date and time in the implicit time zone of UTC.
fn:current-time	Returns the current time in the implicit time zone of UTC.
fn:dateTime	Constructs and returns an xs:dateTime value from an xs:date and an xs:time values.
fn:implicit-timezone	Returns the implicit time zone value of PTOS, which is of type xs:dayTimeDuration. The value PTOS indicates that UTC is the implicit time zone.

► Sequence functions:

Table 4-18 lists XQuery-defined sequence functions.

Table 4-18 XQuery-defined sequence functions

Function	Description
fn:count	Returns the number of values in a sequence.
fn:data	Returns the input sequence after replacing any nodes by their typed values.
fn:deep-equal	Compares two sequences to determine whether they meet requirements for deep equality.
fn:distinct-value	Returns the distinct values in a sequence.
fn:empty	Returns true if its argument is an empty sequence. Otherwise returns false.
fn:exactly-one	Returns its argument if it is a sequence containing exactly one element. Otherwise error is returned.
fn:exist	Returns true if its argument is a sequence with at least one element. Otherwise returns false.
fn:last	Returns the number of values in the sequence that is currently processed.
fn:index-of	Returns a position where an item appears in a sequence.
fn:insert-before	Inserts a sequence before a given position in another sequence.
fn:one-or-more	Returns its argument if it is a sequence containing one or more items. Otherwise returns an error.
fn:position	Returns the position of the context item in the sequence that is currently processed.
fn:remove	Removes an item from a sequence.
fn:reverse	Reverses the order of the items in a sequence.
fn:subsequence	Returns a subsequence of a sequence.
fn:unordered	Returns the items in a sequence in non-deterministic order.

► QName functions:

Table 4-19 lists XQuery-defined QName functions.

Table 4-19 XQuery-defined QName functions

Function	Description
fn:in-scope-prefixes	Returns list of prefixes for all in-scope namespaces of an element.
fn:local-name-from-QName	Returns the local part of an xs:QName value.
fn:namespace-uri-for-prefix	Returns the namespace URI that is associated with a prefix in the in-scope namespaces of an element.
fn:namespace-uri-from-QName	Returns the namespace URI part from of an xs:QName value.
fn:QName	Builds and returns an expanded name from a namespace URI and a string that contains a lexical QName (with an optional prefix).
fn:resolve-QName	Converts a string containing a lexical QName into an expanded QName by using the in-scope namespaces of an element to resolve the namespace prefix to a namespace URI.

► Node functions:

Table 4-20 lists XQuery-defined node functions.

Table 4-20 XQuery-defined node functions

Function	Description
fn:local-name	Returns the local name property of a node.
fn:name	Returns the prefix and local name parts of a node name.
fn:namespace-uri	Returns the namespace URI of the qualified name of a node.
fn:node-name	Returns the expanded QName of a node.
fn:root	Returns the root node of a tree to which a node belongs.

- Other functions:

Table 4-21 lists some other XQuery-defined functions.

Table 4-21 XQuery-defined functions

Function	Description
fn:default-collation	Returns a URI that represents the default collation that is defined for the database.
db2-fn:sqlquery	Retrieves a sequence that is the result of an SQL fullselect in the currently connected DB2 database.
db2-fn:sqlcolumn	Retrieves a sequence from a column in the currently connected DB2 database.

4.2.2 FLWOR and selecting XML data

Very often the FLWOR (for, let, where, order by, and return) expression in XQuery is compared with SELECT-FROM-WHERE block in SQL. They have similar structure and contain multiple clauses denoted by keywords. Table 4-22 lists clauses in a FLWOR expression.

Table 4-22 FLWOR expression clauses

Clause	Description
for	Iterates through an input sequence, binding a variable to each of its items in turn.
let	Declares a variable and assigns it a value. The value could be a sequence containing multiple elements.
where	Specifies criteria for filtering query results.
order by	Specify the sort order of the result.
return	Generates the result to be returned by FLWOR expression.

Next, we discuss and provide examples for each of the clauses.

For and return clauses

We discuss *for* and *return* together in order to show a complete example. Without the return clause, the FLWOR expression will not be complete. A *for* clause iterates through the result of an expression and binds a variable to each item in the sequence. Example 4-35 shows the simplest type of *for* clause containing one variable *\$i* and an expression that results in the sequence (1, 2, 3).

Example 4-35 Using for and return clauses

```
XQUERY
for $i in (1 to 3)
return $i;
```

```
result:
1
2
3
```

The expression in the for clause (1 to 3) is evaluated, and it generates the sequence (1, 2, 3). Each of the values in this sequence binds to the variable `$i`, one at a time. The return clause is evaluated for each of the bindings. The results of these evaluations form a sequence that is returned as a result. In Example 4-35, a return clause is executed for each of the bindings of variable `$i` and returns a sequence of its values.

A for clause can contain multiple variables bound to different expressions. Example 4-36 shows a for clause containing two variables `$i` and `$j`, and two expressions whose results are bind to the variables.

Example 4-36 Using for clause with two variables

```
XQUERY
for $i in (1 to 2), $j in (2 to 3)
return ($i, $j);
```

```
result:
1
2
2
2
1
3
2
3
```

When the *for* clause is evaluated, a tuple of variable bindings is created for each combination of values. `$i` can be bound to 1 and to 2. `$j` can be bound to 2 and to 3. The four possible combinations results in four tuples of variable bindings:

```
$i = 1, $j = 1
$i = 2, $j = 1
$i = 1, $j = 2
$i = 2, $j = 2
```

The return clause executes ones for each tuple of bindings returning the values of \$i and \$j.

If a binding expression evaluates to an empty sequence, no for bindings are generated, and no iterations occur. Example 4-37 shows a for clause where the expression for the variable \$j is an empty sequence. As a result, there are no iterations and the result returned by the return clause is an empty sequence.

Example 4-37 Using for clause with empty sequence

```
XQUERY
for $i in (1 to 2), $j in (3 to 2)
return $i;

result:
```

Note that even if \$j is not part of the return clause, the result is an empty sequence.

When a variable iterates over the items in a sequence, an index or position number is generated for each item in the list. You can declare a position variable for this index with the for clause. The positions are integers starting with 1. The positional variable is defined by the keyword *at*. Example 4-38 shows the usage of positional variables.

Example 4-38 Using for clause with positional variable

```
XQUERY
for $name at $pos in ("Elena", "Maria", "Emma")
return <name pos="{ $pos }">{ $name }</name>;

result:
<name pos="1">
Elena
</name>
<name pos="2">
Maria
</name>
<name pos="3">
Emma
</name>
```

Note that the actual order of the elements in the output stream is not guaranteed unless the FLWOR expression contains an order by clause. Example 4-39 shows that the results produced by the FLWOR expression in Example 4-38 on page 105 could be in a different order.

Example 4-39 Possible results from FLWOR expression

```
result:
<name pos="2">
Maria
</name>
<name pos="1">
Elena
</name>
<name pos="3">
Emma
</name>
```

Let clause

Let clauses bind a variable to the entire result of an expression. The *let* clause does not iterate through a sequence of input, binding each item to a variable, as the *for* clause does. Instead, a *let* clause assigns the whole sequence (or just a value if it is a sequence of one item) to a variable. Example 4-40 shows the differences between *for* and *let* clauses.

Example 4-40 Differences between for and let clauses

```
XQUERY
for $i in (1 to 2)
return <value>{$i}</value>;
```

```
result:
<value>
1
</value>
<value>
2
</value>
```

```
XQUERY
let $i := (1 to 2)
return <value>{$i}</value>;
```

```
result:
<value>
1 2
</value>
```

When the let clause in Example 4-40 on page 106 is executed, a single binding is created for the entire sequence that results from (1 to 2). The return clause executes once.

If the binding expression is an empty sequence, a let binding is created and it contains an empty sequence.

Using for and let clauses in the same expression

When for and let clauses are used in same FLWOR expression, the variable bindings that are generated by let clause are added to the variable bindings that are generated by the for clause. In Example 4-41, the for and let clauses result in the following two tuples of bindings:

```
$i = 1, $j = (2, 3)
$i = 2, $j = (2, 3)
```

Example 4-41 Using for and let clauses in same FLWOR expression

```
XQUERY
for $i in (1 to 2)
let $j := (2 to 3)
return <value>{$i, $j}</value>;
```

```
result:
<value>
1 2 3
</value>
<value>
2 2 3
</value>
```

A variable that is bound in a for or let clause is in scope and can be used in all of the sub-expressions that appear after the variable binding in the FLWOR expression. In Example 4-42 we use \$i to set a value for \$j. Note how the results differ from Example 4-41.

Example 4-42 Variable scope in for or let clause

```
XQUERY
for $i in (1 to 2)
let $j := ($i to 3)
return <value>{$i, $j}</value>;
```

```
result:
<value>
1 1 2 3
</value>
<value>
2 2 3
</value>
```

Where clause

A *where* clause filters the tuples of variable bindings that are generated by for and let clauses in a FLWOR expression. The where clause specifies a condition that is applied to each tuple of variable bindings. If the condition is not true, the tuple is discarded. The return clause is executed only for the remaining tuples, so the where clause effectively filters the results.

Example 4-43 shows the usage of where clause. We keep only the tuples for which \$i is equal to \$j and returning only the values that are in both input sequences.

Example 4-43 Using where clause

```
XQUERY
for $i in (1 to 3)
for $j in (2 to 4)
where ($i eq $j)
return $i;
```

```
result:
2
3
```

Order by clause

An *order by* clause in a FLWOR expression determines the order in which the tuples of variable binding are evaluated by return clause. If no order by clause is specified, the results of a FLWOR expression are returned in a non-deterministic order. An order by clause contains one or more ordering specifications. Each ordering specification consists of an expression and an order modifier, which specifies the sort order (ascending or descending).

In Example 4-44, an order by clause is used to sort the names.

Example 4-44 Using order by clause

```
XQUERY
for $name in ("Elena", "Maria", "Emma", "Antoaneta")
order by $name ascending
return $name;
```

```
result:
Antoaneta
Elena
Emma
Maria
```

Using FLWOR to retrieve XML data

We use the XPS table from Example 4-2 on page 81 to demonstrate how to use FLWOR to retrieve XML data. We already inserted one record in the table (see Example 4-3 on page 81). We use the code in Example 4-45 to insert two more rows.

Example 4-45 Inserting two rows with XML data in XPS table

```
INSERT INTO xps (id, doc) VALUES (2, XMLPARSE ( DOCUMENT
'<Customer>
  <Name>
    <FirstName>Brad</FirstName>
    <LastName>Hunn</LastName>
  </Name>
  <Address country="US">
    <Street>24 Palm Street</Street>
    <City>Los Gatos</City>
    <State>CA</State>
    <Zip>95030</Zip>
  </Address>
  <Phone type="work">123-678-9012</Phone>
  <Phone type="home">123-789-0123</Phone>
  <Phone type="cell">123-890-1234</Phone>
</Customer>'));

INSERT INTO xps (id, doc) VALUES (3, XMLPARSE ( DOCUMENT
'<Customer>
  <Name>
    <FirstName>Domenico</FirstName>
    <LastName>Blefari</LastName>
```

```

</Name>
<Address country="US">
  <Street>68 Cherry Street</Street>
  <City>San Jose</City>
  <State>CA</State>
  <Zip>95134</Zip>
</Address>
<Phone type="work">234-901-2345</Phone>
<Phone type="home">234-012-3456</Phone>
<Email>dom.blefari@yahoo.com</Email>
</Customer>')));

```

We create one more table, CPL, to store the complaints received from customers. SQL statements for creation of this table are shown in Example 4-46.

Example 4-46 Creation of CPL table

```

CONNECT TO xmlrb;
CREATE TABLE cpl(case_id INTEGER NOT NULL,
                  cust_id INTEGER NOT NULL,
                  complain XML);

```

In the CASE_ID column, we store a unique ID for each complaint. In CUST_ID, we store the ID of the complaining customer. In the COMPLAIN column, we store an XML document containing the complaint. A sample complaint is shown in Example 4-47.

Example 4-47 Sample complaint XML document

```

<Complain status="closed">
  <Received>2007-07-01</Received>
  <ReplyTo>sfer@yahoo.com</ReplyTo>
  <Problem>Have not received yet my last order.</Problem>
</Complain>

```

The structure of the XML document is as follows:

- ▶ <Complain> is the root element of the XML document. It has an attribute status with possible values “open” and “closed”. The value of the attribute indicates if the complaint is open or closed. Each other element is a child of this element.
- ▶ <Received> contains a date on which the complaint was received.
- ▶ <ReplyTo> is an e-mail address of the customer as contact information.
- ▶ <Problem> contains a description of the problem submitted by a customer.

We use the code shown in Example 4-48 to insert five records in a CPL table.

Example 4-48 Inserting data into CPL table

```
INSERT INTO cpl (case_id, cust_id, complain) VALUES (1, 1, XMLPARSE
(DOCUMENT
'<Complain status="closed">
  <Received>2006-07-01</Received>
  <ReplyTo>sfer@yahoo.com</ReplyTo>
  <Problem>Have not received yet my last order.</Problem>
</Complain>'));

INSERT INTO cpl (case_id, cust_id, complain) VALUES (2, 1, XMLPARSE
(DOCUMENT
'<Complain status="open">
  <Received>2006-07-06</Received>
  <ReplyTo>stevef@gmail.com</ReplyTo>
  <Problem>One of the items received is broken.</Problem>
</Complain>'));

INSERT INTO cpl (case_id, cust_id, complain) VALUES (3, 1, XMLPARSE
(DOCUMENT
'<Complain status="open">
  <Received>2006-07-11</Received>
  <ReplyTo>sfer@yahoo.com</ReplyTo>
  <Problem>The replacement I received does not work.</Problem>
</Complain>'));

INSERT INTO cpl (case_id, cust_id, complain) VALUES (4, 3, XMLPARSE
(DOCUMENT
'<Complain status="open">
  <Received>2006-07-3</Received>
  <ReplyTo>dom.blefari@yahoo.com</ReplyTo>
  <Problem>Yesterday was put on hold for 4 hours.</Problem>
</Complain>'));

INSERT INTO cpl (case_id, cust_id, complain) VALUES (5, 3, XMLPARSE
(DOCUMENT
'<Complain status="closed">
  <Received>2006-07-12</Received>
  <ReplyTo>sarahb@yahoo.com</ReplyTo>
  <Problem>Have not received my refund.</Problem>
</Complain>'));
```

Selecting all XML documents

We start with a very simple XQuery that selects all XML documents stored in the DOC column of XPS table. The code of the XQuery using a FLWOR expression is shown in Example 4-49.

Example 4-49 Retrieving all XML documents

```
XQUERY
for $doc in db2-fn:xmlcolumn('XPS.DOC')
return $doc;
```

Note that this XQuery returns the whole XML document. Example 4-50 shows a fragment of the query output. In our example, the comment line of our first XML document is included.

Example 4-50 XML document returned from XQuery

```
<!-- sample xml file -->
<Customer>
  <Name>
    <FirstName>
      Steve
    </FirstName>
    <LastName>
      Ferrington
    </LastName>
  </Name>
  <Address country="US">
    <Street>
      46 Oak Street
    </Street>
    <City>
      Los Gatos
    </City>
    <State>
      CA
    </State>
    <Zip>
      95030
    </Zip>
  </Address>
  <Phone type="work">
    123-456-7890
  </Phone>
  <Phone type="home">
    234-567-8901
```

```

    </Phone>
    <Email>
      sfer@yahoo.com
    </Email>
    <Email>
      stevef@gmail.com
    </Email>
  </Customer>
  ...
</Customer>
<Customer>
  ....
</Customer>

```

3 record(s) selected.

If you have only the document's root element without other nodes such as processing instructions, you can use the code shown in Example 4-51.

Note that the document comment is not included in the result because we retrieve the root element. The root element is the child of the document node, not of the root element.

Example 4-51 Retrieve element only

```

XQUERY
for $cust in db2-fn:xmlcolumn('XPS.DOC')/Customer
return $cust;

```

Result:

```

<Customer>
  <Name>
    <FirstName>
      Steve
    </FirstName>
    <LastName>
      Ferrington
    </LastName>
  </Name>
  <Address country="US">
    <Street>
      46 Oak Street
    </Street>
    <City>
      Los Gatos

```

```

    </City>
    <State>
      CA
    </State>
    <Zip>
      95030
    </Zip>
  </Address>
  <Phone type="work">
    123-456-7890
  </Phone>
  <Phone type="home">
    234-567-8901
  </Phone>
  <Email>
    sfer@yahoo.com
  </Email>
  <Email>
    stevef@gmail.com
  </Email>
</Customer>
<Customer>
...
</Customer>
<Customer>
...
</Customer>

```

3 record(s) selected.

If you do not know the name of the root element, you can use `/*` instead of `/Customer`, as shown here:

```

XQUERY
for $cust in db2-fn:xmlcolumn('XPS.DOC')/*
return $cust

```

Selecting parts of XML documents

In this example, we show how to select only parts of XML documents. Suppose that we require only the e-mail addresses. The code in Example 4-52 does the job.

Example 4-52 Retrieving e-mail addresses

```
XQUERY
for $cust in db2-fn:xmlcolumn('XPS.DOC')/Customer
return $cust/Email
```

```
result:
<Email>
sfer@yahoo.com
</Email>
<Email>
stevef@gmail.com
</Email>
<Email>
dom.blefari@yahoo.com
</Email>
```

If we do not require the entire Email element, but only the e-mail address itself, we can use the `text()` function, as shown in Example 4-53.

Example 4-53 Retrieving the text in element

```
XQUERY
for $cust in db2-fn:xmlcolumn('XPS.DOC')/Customer
return $cust/Email/text()
```

```
result:
sfer@yahoo.com
stevef@gmail.com
dom.blefari@yahoo.com
```

Selecting specific XML documents

Suppose that we require only the names of the customers whose zip code is 95030. We can put this restriction using the `where` clause in our FLWOR expression, as shown in Example 4-54. Note that it is very similar to the SQL `WHERE` clause. We also use the concatenation function to form a string containing both first and last names.

Example 4-54 Using where clause

```
XQUERY
for $cust in db2-fn:xmlcolumn('XPS.DOC')/Customer
where $cust/Address/Zip/text()='95030'
return concat($cust/Name/FirstName, " ", $cust/Name/LastName)
```

```
results:
Steve Ferrington
Brad Hunn
```

Very often we can filter data in the path expression instead of in where clause. Example 4-55 produces the same results as the code in Example 4-54 on page 115. It is your choice of how and where to specify the filter. The rule of thumb is to use the one that will make your code more readable and easier to understand.

Example 4-55 Using predicate instead of where clause

```
XQUERY
for $cust in
db2-fn:xmlcolumn('XPS.DOC')/Customer[Address/Zip/text()='95030"]
return concat($cust/Name/FirstName, " ", $cust/Name/LastName)
```

```
results:
Steve Ferrington
Brad Hunn
```

Another criteria for filtering is the existence or number of occurrences of a specific element in an XML document. In Example 4-56, we show how to select only the customers that do not have an e-mail address.

Example 4-56 Retrieving customers without e-mail address

```
XQUERY
for $cust in db2-fn:xmlcolumn('XPS.DOC')/Customer
where not(exists($cust/Email))
return concat($cust/Name/FirstName, " ", $cust/Name/LastName);
```

```
result:
Brad Hunn
```

We use the *exists* and *not* functions. Note that `not($cust/Email)` returns the same result as `not(exists($cust/Email))`.

In Example 4-57 we select only the customers with more than one e-mail address.

Example 4-57 Retrieving customers with more than one e-mail address

```
XQUERY
for $cust in db2-fn:xmlcolumn('XPS.DOC')/Customer
where count($cust/Email) > 1
```

```
return concat($cust/Name/FirstName, " ", $cust/Name/LastName);

result:
Steve Ferrington
```

Some differences between XQuery and SQL

When storing data in relational table, NULL is used to represent a column without a value. For each column, an SQL query will receive one value (possible NULL) from each row. Because XML documents omit missing or unknown data, XQuery does not have a NULL value. The result of an XQuery does not have this row/column relationship.

Examine further the XQuery output in Example 4-53 on page 115. The three e-mail addresses are not corresponding to the three customers (three rows) in the table. One of our documents (customer Brad Hunn) does not contain an Email element and XQuery simply does not return any entry in the output. Another document (customer Steve Ferrington) contains two Email elements and XQuery returns two entries in the output. Based on the results, we cannot determine from which document the e-mail addresses come.

In Example 4-58 we show how to group e-mail addresses by customer. This query also returns the number of e-mail addresses for each customer.

Example 4-58 Grouping the e-mail addresses by customer

```
XQUERY
for $cust in db2-fn:xmlcolumn('XPS.DOC')/Customer
let $name := concat($cust/Name/FirstName, " ", $cust/Name/LastName)
let $n := count($cust/Email)
return (concat($name, " : ", $n), $cust/Email/text());

result:
Steve Ferrington : 2
sfer@yahoo.com
stevef@gmail.com
Brad Hunn : 0
Domenico Blefari : 1
dom.blefari@yahoo.com
```

In Example 4-59, the query outputs only the first e-mail address per customer.

Example 4-59 Selecting the first e-mail address

```
XQUERY
for $cust in db2-fn:xmlcolumn('XPS.DOC')/Customer
```

```
let $name := concat($cust/Name/FirstName, " ", $cust/Name/LastName)
return (concat($name, " : ", $cust/Email[1]/text()));
```

```
result:
Steve Ferrington : sfer@yahoo.com
Brad Hunn :
Domenico Blefari : dom.blefari@yahoo.com
```

Assume that we require a list for customer contact information. We prefer e-mail addresses. For customers without e-mail, we would like to have their phone number. XQuery in Example 4-60 generates this information using conditional expressions. Here we select the last e-mail and phone (assuming that it is most recent).

Example 4-60 Using conditional expression

```
XQUERY
for $cust in db2-fn:xmlcolumn('XPS.DOC')/Customer
let $name := concat($cust/Name/FirstName, " ", $cust/Name/LastName)
let $info := if (exists($cust/Email))
               then($cust/Email[last()]/text())
               else($cust/Phone[last()]/text())
return (concat($name, " : ", $info))
```

```
result:
Steve Ferrington : sfer@yahoo.com
Brad Hunn : 123-678-9012
Domenico Blefari : dom.blefari@yahoo.com
```

You can try to modify the code so that if there is no e-mail address, first look for home phone, then for cell phone, and finally for work phone.

Transforming XML data

We also can produce an XML document using XQuery. See the output in Example 4-60. For each customer, we have one line containing the full name and e-mail address or phone number. Rewrite the code so the output is an XML document with the structure shown in Example 4-61.

Example 4-61 Desired transformed XML document

```
<CustomerContacts>
  <Customer>
    <Name>name</Name>
    <Contact type="phone"|"email">contact info</Contact>
  </Customer>
```



```

    <Customer>
    ...
  </Customer>
  ...
</CustomerContacts>

```

The code in Example 4-62 does the transformation. Note that we include the whole FLWOR body between two `<CustomerContacts>` tags. We also use the curly bracket to instruct DB2 to evaluate the enclosed expression rather than treating it as a literal string.

Example 4-62 Transforming an XML document

```

XQUERY
<CustomerContacts>{
  for $cust in db2-fn:xmlcolumn('XPS.DOC')/Customer
  let $name := concat($cust/Name/FirstName, " ", $cust/Name/LastName)
  let $info := if (exists($cust/Email))
    then($cust/Email[last()]/text())
    else($cust/Phone[last()]/text())
  let $type := if (exists($cust/Email)) then("email") else("phone")
  return
  <Customer>
  <Name>{$name}</Name>
  <Contact type="{ $type }">{$info}</Contact>
</Customer>
}</CustomerContacts>

result:
<CustomerContacts>
<Customer>
  <Name>
    Steve Ferrington
  </Name>
  <Contact type="email">
    stevef@gmail.com
  </Contact>
</Customer>
<Customer>
  <Name>
    Brad Hunn
  </Name>
  <Contact type="phone">
    123-890-1234
  </Contact>

```

```

</Customer>
<Customer>
  <Name>
    Domenico Blefari
  </Name>
  <Contact type="email">
    dom.blefari@yahoo.com
  </Contact>
</Customer>
</CustomerContacts>

```

Joining two or more XML documents in an XQuery

In Example 4-63 we demonstrate how to join XML data from two tables. We want to list all the customers whose e-mail matches the e-mail in the CPL table ReplyTo element. First we extract to \$e all the e-mail addresses that are in complains. Then we eliminate duplicates and store the distinct values in \$de. For every customer, we then check if the e-mail address matches an e-mail address in \$de. Note that the check is done by using general comparison (\$cust/Email/text() = \$de). The result of general comparison is true if some of the elements in first sequence are equal to some of the elements in the second sequence.

Example 4-63 Joining XML data

```

XQUERY
let $e := db2-fn:xmlcolumn('CPL.COMPLAIN')/Complain/ReplyTo/text()
let $de := distinct-values($e)
for $cust in db2-fn:xmlcolumn('XPS.DOC')/Customer
where $cust/Email/text() = $de
return $cust/Name/LastName/text()

```

```

results:
Ferrington
Blefari

```

4.2.3 Updating XML data

You can update an entire XML document, or part of one, stored in a DB2 table.

Updating the whole XML document

We use the SQL UPDATE command to update a whole XML document stored in an XML column. We specify the rows that must be updated using the SQL

WHERE clause. We continue to use the table XPS created in Example 4-2 on page 81 to show how to change an XML document.

Example 4-64 shows the SQL SELECT command you can use to display the content of the XML document before and after update.

Example 4-64 Content of the XML that will be updated

```
SELECT doc FROM xps WHERE id = 1;
```

Example 4-65 shows the SQL UPDATE command that updates the whole XML document.

Example 4-65 Updating an XML document

```
UPDATE xps SET doc = XMLPARSE ( DOCUMENT (
'<!-- sample xml file -->
<Customer>
  <Name>
    <FirstName>Steve</FirstName>
    <LastName>Ferrington</LastName>
  </Name>
  <Address country="US">
    <Street>46 Oak Street</Street>
    <City>Los Gatos</City>
    <State>CA</State>
    <Zip>95030</Zip>
  </Address>
  <Phone type="work">987-654-3210</Phone>
  <Phone type="home">234-567-8901</Phone>
  <Email>sfer@yahoo.com</Email>
  <Email>stevef@gmail.com</Email>
</Customer>'))
WHERE id = 1
```

Updating only a part of an XML document

In Example 4-65 we demonstrated how to update a whole XML document. What if we have to make changes only in a part of an XML document? For example, we just want to add a new e-mail address, to update a phone number, or to change the address. To accomplish a partial update on an XML document using the SQL or XQUERY command, you have to retrieve the entire document, modify it, then use the SQL UPDATE command to replace the document with the modified version.

Another approach is to create an update stored procedure that is capable of updating XML documents stored in the database. In this section, we introduce an as-is XML update stored procedure XMLUPDATE. We show how to use this stored procedure to update a part of an XML document. For full details and more examples about this stored procedure, refer to the following Web site:

<http://www.ibm.com/developerworks/db2/library/techarticle/dm-0605singh/>

The XMLUPDATE stored procedure supports the following partial update functions in an XML document:

- ▶ Change the value of any text or attribute node.
- ▶ Insert a new element.
- ▶ Replace an element node (along with all its children) with another element.
- ▶ Delete a node.

Note that when updating a portion of an XML document with a stored procedure, DB2 writes the entire XML document back to the database under the covers.

Setting up the stored procedure

XMLUPDATE is a Java stored procedure. To install the stored procedure, follow these steps:

1. Make sure the JCC driver is set up for DB2. Run the following command when DB2 is up:

```
db2set DB2_USE_DB2JCCT2_JROUTINE=on
```

2. Increase the default JAVA heap size to allow serializing XML documents in memory. It is done with the following command:

```
db2 update dbm cfg using JAVA_HEAP_SZ 1024
```

3. Download XMLUPDATE stored procedure jar file from the following Web site:

<http://www.ibm.com/developerworks/db2/library/techarticle/dm-0605singh/>

4. Install the stored procedure jar file into DB2 using the following commands:

```
connect to xmlrb
call SQLJ.INSTALL_JAR('file:///d:/work/db2/db2xmlfunctions.jar',
db2xmlfunctions, 0)
```

Replace XMLRB with the name of your database, and d:/work/db2 with your jar file location.

5. Register the stored procedure, as shown in Example 4-66.

Example 4-66 Registering the stored jar file

```
CREATE PROCEDURE db2xmlfunctions.XMLUPDATE(
  IN COMMANDSQL VARCHAR(32000),
```

```
IN QUERYSQL VARCHAR(32000),
IN UPDATESQL VARCHAR(32000),
OUT errorCode INTEGER, OUT errorMsg VARCHAR(32000))
DYNAMIC RESULT SETS 0
LANGUAGE JAVA
PARAMETER STYLE JAVA
NO DBINFO
FENCED
NULL CALL MODIFIES SQL DATA
PROGRAM TYPE SUB
EXTERNAL NAME
'db2xmlfunctions:com.ibm.db2.xml.functions.XMLUpdate.Update'
```

XMLUPDATE procedure parameters

The XMLUPDATE stored procedure has several parameter options. Here we describe some of the frequently used parameters.

We call the stored procedure using the following command in Example 4-67.

Example 4-67 Calling the stored procedure

```
DB2XMLFUNCTIONS.XMLUPDATE (commandXML,
                           querySQL,
                           updateSQL,
                           errorCode,
                           errorMsg)
```

► **commandXML**

This is an XML document that describes the update commands. These commands are applied to the XML document specified by querySQL. The structure of the commandXML document is:

```
<update namespaces="name=prefix:namespace">
<update using="SQL" col="column_number" path="XPathExpression">
update value </update>
</updates>
```

The essential arguments in this XML document are:

- @col: This is the number of the column being modified in the querySQL.
- @path: This is the XPath location of the node in the target XML document.
- @action: This is the action to be performed:
 - replace: Replace the target node with update node.
 - append: Append the update value as child to the target node.
 - delete: Delete the target node.

- update value: This should be a text node or an element.
- querySQ

This is a valid SQL statement for retrieving XML document to be updated.
- updateSQ

This is a parameterized update SQL statement.

Using XMLUPDATE stored procedure

Suppose that we have to add a new e-mail address for a customer (keeping the existing ones). Example 4-68 shows how to add a new element in an XML document using the XMLUPDATE stored procedure.

Look closer at the arguments we submit:

- action="append": We add new element.
- col="1": We work with the first XML column returned by querySQL.
- path="/Customer": The element we add will be a child to this element.
- <Email>newEmail@yahoo.com: The new element.
- 'select doc from xps where id = 1': Our querySQL.
- 'update xps set doc = ? where id = 1': Our updateSQL.

Example 4-68 Adding a new Email element

```
CALL DB2XMLFUNCTIONS.XMLUPDATE(
  '<updates>
  <update action="append" col="1" path="/Customer">
  <Email>newEmail@yahoo.com</Email>
  </update>
  </updates>',
  'select doc from xps where id=1',
  'update xps set doc=? where id=1', ?, ?);
```

Now change the customer address. Example 4-69 shows how to replace an element in an XML document. Here are the arguments we submit:

- action="replace": We replace an element.
- path="/Customer/Address": This is the element we are replacing.

The new element is listed in Example 4-70.

Example 4-69 Replacing the Address element

```
CALL DB2XMLFUNCTIONS.XMLUPDATE(
  '<updates>
  <update action="replace" col="1" path="/Customer/Address">
  <Address>
```

```
<Street>123 Woodstone Road</Street>
<City>Clinton</City>
<State>MS</State>
<Zip>39056</Zip>
</Address>
</update>
</updates>',
'select doc from xps where id=1',
'update xps set doc=? where id=1', ?, ?)
```

Example 4-70 Modified Address element

```
<Address>
  <Street>123 Woodstone Road</Street>
  <City>Clinton</City>
  <State>MS</>
  <Zip>39056</Zip>
</Address>
```

In Example 4-71 we demonstrate how to update a text element using the XMLUPDATE stored procedure. We replace the work phone with a new number. Here are the parameters we submit:

- ▶ action="replace": We replace a text.
- ▶ path="/Customer/Phone[@type='work']/text()": The text we replace. Note that we replaced double quote " with " because we require nested quotes.
- ▶ The new phone number: 601-925-1234

Example 4-71 Changing the work phone number

```
CALL DB2XMLFUNCTIONS.XMLUPDATE(
  '<updates>
<update action="replace" col="1"
path="/Customer/Phone[@type='work']/text() ">
601-925-1234
</update>
</updates>',
'select doc from xps where id=1',
'update xps set doc=? where id=1', ?, ?)
```

4.3 XQuery and SQL/XML

In the previous section, we discussed how to work with XML documents (stored in DB2 XML columns using XQuery). But what should we do if we have to access both XML and relational data? DB2 supports both embedding SQL in XQuery and embedding XQuery in SQL statements. In the following sections we consider both of these possibilities.

4.3.1 XQuery with embedded SQL

The DB2 function *db2-fn:sqlquery* allows us to embed SQL statements in XQuery. It accepts as an argument an SQL full select statement that must return XML data. Refer to Example 4-60 on page 118, where we display the e-mail address or phone number for each customer. We can replace the function *db2-fn:xmlcolumn* with *db2-fn:sqlquery*. The resulting code is shown in Example 4-72.

Example 4-72 Using the db2-fn:sqlquery function

```
XQUERY
for $cust in db2-fn:sqlquery('SELECT DOC FROM XPS')/Customer
let $name := concat($cust/Name/FirstName, " ", $cust/Name/LastName)
let $info := if (exists($cust/Email))
              then($cust/Email[last()]/text())
              else($cust/Phone[last()]/text())
return (concat($name, " : ", $info));

result:
Steve Ferrington : sfer@yahoo.com
Brad Hunn : 123-678-9012
Domenico Blefari : dom.blefari@yahoo.com
```

As you can see, the results are the same. This is because the SQL SELECT statement that we use in the *db2-fn:sqlquery* function returns the same data as the *db2-fn:xmlcolumn* function used in Example 4-60 on page 118. The power of the *db2-fn:sqlquery* function is that we can utilize all the features of the SQL SELECT statement.

Let us suppose that instead of contact information for every customer, we only require contact information for a customer with a specific ID number. The ID field is not stored in the XML document; we can access the data using the SQL statement instead of accessing it from the FLWOR expression. Using the *db2-fn:sqlfunction* allows us to put this restriction in the WHERE clause of the SQL SELECT statement. In Example 4-73, by embedding SQL in XQuery, we combine XML and relational predicates.

Example 4-73 Combining XML and rational predicates

```
XQUERY
for $cust in db2-fn:sqlquery('SELECT doc FROM xps WHERE id=1')/Customer
let $name := concat($cust/Name/FirstName, " ", $cust/Name/LastName)
let $info := if (exists($cust/Email))
              then($cust/Email[last()]/text())
              else($cust/Phone[last()]/text())
return (concat($name, " : ", $info));

result:
Steve Ferrington : stevef@gmail.com
```

Note that in the embedded SQL statement, we are not limited only to the table containing the XML column we are retrieving. In Example 4-74 we retrieve the names of all customers that have complaints.

Example 4-74 Retrieving customers with complaints

```
XQUERY
for $cust in db2-fn:sqlquery('SELECT doc FROM xps
WHERE EXISTS (SELECT case_id FROM cpl WHERE cust_id = id)')/Customer
return $cust/Name/LastName/text();

results:
Ferington
Blefari
```

4.3.2 SQL/XML

SQL/XML is part of the XML language. It defines the XML data type and a set of functions for querying, constructing, validating, and transforming XML data.

Some of the most often used SQL/XML functions supported in DB2 are XMLQUERY, XMLTABLE, and XMLEXISTS. These functions allow us to embed XQuery expressions in SQL. For a complete list of SQL/XML functions, refer to the DB2 manuals, *SQL Reference, Volume 1*, SC10-4249, and *SQL Reference, Volume 2*, SC10-4250.

XMLEXISTS predicate

The XMLEXISTS predicate determines whether an XQuery expression returns a sequence of one or more elements. If the specified XQuery expression returns an empty sequence, XMLEXISTS returns false, otherwise it returns true.

In Example 4-75, we demonstrate the XMLEXISTS predicate. It is commonly used in the WHERE clause to express predicates over XML data. Here we display the IDs of all customers that have an e-mail address.

Example 4-75 Using the XMLEXISTS function

```
SELECT id
FROM xps
WHERE XMLEXISTS('$d/Customer/Email' passing xps.doc as "d");
```

results:

1
3

In the XPS table, if you want to see the customer ID of a customer whose first name is "Brad", the query in Example 4-76 returns this information.

Example 4-76 Selecting the customer ID of a specific customer

```
SELECT XPS.ID FROM XPS
WHERE xmlexists('$i/Customer/Name[FirstName = "Brad"]' passing XPS.DOC
as "i");
```

Result:

ID

2

1 record(s) selected.

Example 4-77 shows an intuitive way of coding a query to get the customer ID of customer Brad. This query returns all the customer IDs in the table, and this is not what is expected.

Example 4-77 Query returns more values

```
SELECT XPS.ID FROM XPS
WHERE XMLEXISTS('$i/Customer/Name/FirstName = "Brad"' PASSING XPS.DOC
AS "i");
```

Result:

```
ID
-----
1
2
3
```

3 record(s) selected.

The XMLEXISTS function tests the existence of XML values and returns TRUE and FALSE. The function returns FALSE only if the XQuery in XMLEXISTS returns an empty sequence. Otherwise, it always returns TRUE. As shown in Example 4-78, the expression `Customer/Name/FirstName="Brad"` returns TRUE or FALSE, not sequences. When this expression is used in the XMLEXISTS function, the function returns TRUE. The condition is met, and DB2 returns all customer IDs in the table.

Example 4-78 XQuery returns TRUE

```
XQUERY for $i in db2-fn:xmlcolumn('XPS.DOC')
/ Customer/Name/FirstName = "Brad"
return $i;
```

Results:

```
1
-----
true
```

1 record(s) selected.

XMLQUERY function

XMLQUERY is an SQL function that allows you to execute XQuery expressions within an SQL statement. The XMLQUERY function returns a sequence.

In Example 4-79 we introduce the XMLQUERY function. It is typically used in a SELECT clause to extract data from an XML column. Here we look for all customers having a zip code equal to 95030. In addition to the ID, we also display the last name extracted from the XML data.

Example 4-79 Using XMLQUERY function

```
SELECT id, XMLQUERY('for $ln in $d/Customer/Name/LastName/text()
                    return $ln'
                    passing xps.doc as "d")
FROM xps
WHERE XMLEXISTS('$d/Customer/Address/Zip[text()="95030"]'
                passing xps.doc as "d");
```

results:

```
1 <LastName>Ferrington</LastName>
2 <LastName>Hunn</LastName>
```

Note that the type of the column containing the last name is XML. In Example 4-80 we use XMLCAST to cast the XML value to VARCHAR(20).

Example 4-80 Using XMLCAST to convert results from XMLQUERY

```
SELECT id, XMLCAST(
XMLQUERY('for $ln in $d/Customer/Name/LastName/text()
        return $ln'
        passing xps.doc as "d") AS VARCHAR(20))
FROM xps
WHERE XMLEXISTS('$d/Customer/Address/Zip[text()="95030"]'
                passing xps.doc as "d");
```

results:

```
1 Ferrington
2 Hunn
```

XMLTABLE function

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

In Example 4-81 we use the XMLTABLE function to retrieve multiple XML element values: customer first name, last name, and zip code.

Example 4-81 Using XMLTABLE function

```
SELECT id, firstname, lastname, zipcode
FROM xps, XMLTABLE(
    'for $cust in $d/Customer
```

```

        return $cust' passing xps.doc as "d"
    COLUMNS firstname VARCHAR(20) path 'Name/FirstName/text()',
               lastname VARCHAR(20) path 'Name/LastName/text()',
               zipcode VARCHAR(10) path 'Address/Zip/text()')
    as nameszip

```

results:

1	Steve	Ferrington	95030
2	Brad	Hunn	95030
3	Domenico	Blefari	95134

XMLTABLE generates tabular output from XML data. It is very useful for providing us with a relational view of XML data.

Suppose that we have to filter the results from Example 4-81 on page 130 in order to display data only for customers with the zip code 95030. We can put the restriction in two different places:

- ▶ In the WHERE clause of the SQL statement
- ▶ in the XQUERY used in the XMLTABLE function call

In Example 4-82 we show how to filter results using the WHERE clause.

Example 4-82 Filtering results using WHERE clause

```

SELECT id, firstname, lastname, zipcode
FROM xps, XMLTABLE(
    'for $cust in $d/Customer
    return $cust' passing xps.doc as "d"
    COLUMNS firstname VARCHAR(20) path 'Name/FirstName/text()',
               lastname VARCHAR(20) path 'Name/LastName/text()',
               zipcode VARCHAR(10) path 'Address/Zip/text()')
    as nameszip
WHERE zipcode = '95030'

```

results:

1	Steve	Ferrington	95030
2	Brad	Hunn	95030

In Example 4-83 we show how to filter results inside XQUERY code.

Example 4-83 Filtering XQUERY

```

SELECT id, firstname, lastname, zipcode
FROM xps, XMLTABLE(
    'for $cust in $d/Customer[Address/Zip/text()="95030"]

```

```

        return $cust' passing xps.doc as "d"
    COLUMNS firstname VARCHAR(20) path 'Name/FirstName/text()',
              lastname VARCHAR(20) path 'Name/LastName/text()',
              zipcode VARCHAR(10) path 'Address/Zip/text()')
    as nameszip

```

results:

1	Steve	Ferrington	95030
2	Brad	Hunn	95030

Joining XML with relational data

SQL/XML is very suitable when you have to join XML data with relational data. In Example 4-84 we create a simple table, TRTIME, with no XML data type column. The TRTIME table stores the delivery days required for a zip code. Three records are inserted into this table.

Example 4-84 Creation of TRTIME table

```

CONNECT TO xmlrb;
CREATE TABLE trtime (zip_code CHAR(10),
                    duration INTEGER);
INSERT INTO trtime (zip_code, duration) VALUES
    ('95030', 5), ('95035', 4), ('95134', 3);

```

To produce a list of all the customers and the delivery days required, we use the zip code to join these two tables. In TRTIME, the zip code is in a relational column; and in XPS, the zip code is in an XML document.

Example 4-85 Joining relational and XML data

```

SELECT fname, lname, duration
FROM trtime, xps, XMLTABLE ( 'for $c in $a/Customer return $c'
    passing xps.doc as "a"
    COLUMNS
        fname varchar (20) path 'Name/FirstName',
        lname varchar (20) path 'Name/LastName',
        zip varchar(10) path 'Address/Zip') as T
WHERE trtime.zip_code = zip

```

result:

Steve	Ferrington	5
Brad	Hunn	5
Domenico	Blefari	3

Aggregating and grouping XML data

SQL/XML can be used for grouping and aggregating XML data. There is no explicit group-by construction in XQuery, and usually it is easier to use SQL/XML for grouping. Topically we use the XMLTABLE function and GROUP BY clause.

If we want to display the number of customers by zip code, we can use the XMLTABLE function to extract the zip code and then use GROUP BY to count the number of customers, as shown in Example 4-86.

Example 4-86 Using SQL/XML for grouping XML data

```
SELECT zip, sum(1)
FROM xps, XMLTABLE ('$a/Customer/Address/Zip' passing doc as "a"
                  COLUMNS zip varchar(10) PATH 'text()') as T
GROUP BY zip
```

result:

95030	2
95134	1

4.3.3 When to use what

With the full support for both SQL and XQuery, DB2 9 gives you several options for querying XML data. Very often you will have to choose which one of them to use. In this section we describe available options and highlight their pros and cons.

In DB2, you can query XML data in four different ways using:

- ▶ Plain SQL
- ▶ SQL/XML
- ▶ XQuery
- ▶ XQuery with embedded SQL

Note that whatever combination of SQL and XQuery is used, DB2 uses a single hybrid compiler to generate and optimize an execution plan for the whole query.

Plain SQL

Using plain SQL, you can only work with full XML documents. It can be used to insert, update, or delete XML documents. When you retrieve a full XML document using plain SQL, your selection of the document has to be based only on relational (non-XML) data.

SQL/XML

SQL/XML allows you to do the following operations:

- ▶ Use predicates on both relational and XML data.
- ▶ Access and extract fragments of XML data.
- ▶ Use aggregation and grouping of XML data on the SQL level.
- ▶ Join relational and XML data.
- ▶ Pass parameters to XQuery expressions.

XQuery

XQuery is a language specifically designed for querying XML data. It is very suitable if you have to work with XML data only. You can easily extract, transform, and join XML data using XQuery.

XQuery with embedded SQL

Embedding SQL into XQuery allows you to use predicates based on non-XML data. This way, you can filter documents from an XML column that are processed by XQuery. You receive additional functionality to call external functions on the XML columns. You can also execute full text search on XML data.

Because SQL/XML is the extension of pure SQL, everything you can do with SQL can also be done with SQL/XML. In the same way that XQuery with embedded SQL is the extension of XQuery, so everything you can do with XQuery can also be done using XQuery with embedded SQL.

Features supported by different languages

In Table 4-23 we list several important features that you might have to use while working with XML data. We also show the level of support that different options provide for these features.

Table 4-23 Features supported by different languages

Feature	SQL	SQL/XML	XPath	XPath w/ SQL
Insert, update, delete XML documents	++	++	--	--
Retrieve full XML documents	++	++	++	++
Retrieve parts of XML documents	--	++	++	++
Transform XML data	--	+/-	++	++
Use relational predicates	++	++	--	+
Use XML predicates	--	++	++	++

Feature	SQL	SQL/XML	XPath	XPath w/ SQL
Join XML data	--	+	++	++
Join XML with relational data	--	++	--	++
Aggregate and group XML data	--	++	+/-	+/-
Call external functions	++	++	--	++
Pass parameter markers	+	++	--	--
Execute full text search	+	++	--	++

The following legends are used in the table:

- ▶ ++ Indicates that the feature is fully supported by the language.
- ▶ + Indicates that the feature is supported. However, using another language might be easier and more efficient.
- ▶ +/- Indicates that feature could be expressed but it is difficult and inefficient.
- ▶ -- Indicates that the feature is not supported.

Following are some guidelines for choosing languages to perform the required activities:

- ▶ Insert, update, and delete XML documents:
INSERT, UPDATE, and DELETE statements can be used to insert, update, or delete XML documents. If we can identify the required documents without accessing XML data, SQL is enough. Otherwise, we have to use SQL/XML.
- ▶ Retrieve full XML documents:
All four languages support retrieval of full XML documents. If identifying the required document is based only on relational data, we can use SQL. If it is based only on the XML data, we can use XQuery. If we have to use both relational and XML data to identify required documents, we have to use SQL/XML or XQuery with embedded XQL.
- ▶ Retrieve parts of XML documents:
Both SQL/XML and XQuery can be used for retrieving parts of XML document.
- ▶ Transform XML data:
The easiest way is to use XQuery. It can be done also with SQL/XML but is usually more difficult to code.

- ▶ Use relational predicates:
Plain XQuery does not support relational predicates. You have to embed SQL in XQuery or just use SQL or SQL/XML.
- ▶ Use XML predicates:
Plain SQL does not support them. You have to use SQL/XML or XQuery.
- ▶ Join XML data:
The easiest way is to use XQuery. It can be done with SQL/XML, but usually is more difficult to code.
- ▶ Join XML with relational data:
Both SQL/XML and XQuery with embedded SQL can be used.
- ▶ Aggregate and group XML data:
The easiest way is to use SQL/XML and integrated SQL functions. Aggregating and grouping XML data can be done with XQuery with embedded SQL but is more difficult to code.
- ▶ Call external functions:
XQuery is the only one not supporting external function call.
- ▶ Pass parameter markers:
XQuery is the only one not supporting parsing parameter markers.

4.4 When and how to use namespaces

In this section, we discuss how to work with namespaces in XQuery. We also describe some of the DB2 XQuery built-in functions related to namespaces.

Namespaces in XML

XML namespaces are defined to avoid naming conflicts. Elements from different documents can have the same name, but completely different content. In this section, the example we use is an Address element for storing customer address information. The structure of the Address element with some sample data is shown in Example 4-87.

Example 4-87 Address element containing customer address

```
<Address country="US">  
  <Street>46 Oak Street</Street>  
  <City>Los Gatos</City>  
  <State>CA</State>  
  <Zip>95030</Zip>
```

</Address>

It is possible that in another XML document, an element named Address is used to store an IP address of a system. In Example 4-88 we show such an element.

Example 4-88 Address element containing IP address

```
<Address>123.123.123.123</Address>
```

There will be a naming conflict if we want to use both Address elements in a single document. This conflict can be avoided by using namespaces.

Namespace declaration

A namespace is defined by associating a prefix to a unique Uniform Resource Identifier (URI). Then this prefix can be used to define element names. The full prefixed name is known as a *qualified name* or *QName*. It consists of two parts: the prefix, known as the namespace prefix; and the element name, known as the local name.

In Example 4-89, we define two namespaces. The URI for the first one is `http://first.sample.address.space.com`, and the URI for the second one is `http://second.sample.address.space.com`. Note that XML interprets URIs as strings, and the URLs do not have to point to real locations.

Example 4-89 Definition of namespaces

```
<sample xmlns:fns="http://first.sample.name.space.com"
        xmlns:sns="http://second.sample.name.space.com">
  <fns:Address fns:country="US">
    <fns:Street>46 Oak Street</fns:Street>
    <fns:City>Los Gatos</fns:City>
    <fns:State>CA</fns:State>
    <fns:Zip>95030</fns:Zip>
  </fns:Address>
  <sns:Address>
    123.123.123.123
  </sns:Address>
</sample>
```

We use the `xmlns` attribute to define a namespace and to bind its prefix to a URI. Then we indicate that an element belongs to that namespace by prefixing it.

We can define a default namespace. It will be applied to all elements that are without a prefix and appear within the element containing the declaration of the default namespace. Example 4-90 shows the use of a default namespace.

```
<sample xmlns="http://first.sample.name.space.com"
  xmlns:sns="http://second.sample.name.space.com">
  <Address fns:country="US">
    <Street>46 Oak Street</Street>
    <City>Los Gatos</City>
    <State>CA</State>
    <Zip>95030</Zip>
  </Address>
  <sns:Address>
    123.123.123.123
  </sns:Address>
</sample>
```

Here we define `http://first.sample.name.space.com` to be the default namespace. Note that we do this by not specifying any prefix. It is the default namespace for the sample element and all of its descendant elements. If any element is defined without a prefix, it belongs to the default namespace. In our example these elements are: sample, Address (first occurrence), Street, City, State, Zip. The second Address element is prefixed by `sns` and it belongs to the other namespace we defined, `http://second.sample.name.space.com`.

We can use namespaces not only with element names, but also with attribute and function names. Note that the default namespace does not apply to attribute names. If an attribute name has no prefix, it does not belong to any namespace. In Example 4-90, if we omit the `fns` prefix of the country attribute of the Address element, it will not belong to any namespace.

Predeclared namespaces

There is a set of predeclared namespaces in DB2. You can use their prefixes directly in your code without any explicit declaration. Table 4-24 lists these namespaces. Note that the URI associated with the `xml` prefix cannot be redefined.

Table 4-24 Predeclared namespaces

Prefix	URI	Description
xml	http://www.w3.org/XML/1998/namespace	XML reserved namespace
xs	http://www.w3.org/2001/XMLSchema	XML schema namespace
xsi	http://www.w3.org/XMLSchema-instance	XML schema instance namespace
fn	http://www.w3.org/2005/xpath-functions	Default function namespace

Prefix	URI	Description
xdtd	http://www.w3.org/2005/ xpath-datatypes	XQuery type namespace
db2-fn	http://www.ibm.com/xmlns/prod/db2/functions	DB2 functions namespace

Using namespaces with XQuery

In addition to the DB2 predeclared namespaces, we can define namespaces in the query prolog. Both namespace declaration and default namespace declaration are supported. The namespaces defined in the XQuery prolog can be used anywhere in the XQuery body. In Example 4-91 we show an XQuery containing namespace declaration in its prolog.

Example 4-91 Declaration of an XML namespace in XQuery prolog

```
declare namespace fns = "http://first.sample.name.space.com";
```

During the query processing, XQuery expands the QNames and replaces the namespace prefix with the URI that is bound to it. Two QNames are equal if both their local names and namespace URLs are equal.

XQuery built-in QName and namespace functions

DB2 supports several built-in functions for working with QNames and namespaces. We demonstrate some of these functions using the sample XML document shown in Example 4-92. We create a table and store this document so we are able to use it in all our examples.

Example 4-92 ns-sample.xml document

```
CONNECT TO xmlrb;
CREATE TABLE nss(id INTEGER NOT NULL, doc XML);
INSERT INTO nss (id, doc) VALUES (1, XMLPARSE ( DOCUMENT
'<sample xmlns="http://namespace.sample.com/one"
  xmlns:two="http://namespace.sample.com/two">
  <eone>
    this is in default namespace
  </eone>
  <two:eone anoone="nons" two:atwo="nstwo">
    this is in namespace two
  </two:eone>
</sample>'));
```

Local-name, name, and namespace-uri functions

The *fn:local-name* function returns the local name of a node. The *name* function returns the prefix and local part of a node name. The *namespace-uri* function returns the namespace URI of the node QName. In Example 4-93, we use these functions to display the local names, QNames, and namespace URIs of all element and attribute nodes from the ns-sample.xml document.

Example 4-93 Using local-name, name, and namespace-uri functions

```
XQUERY
let $d := db2-fn:sqlquery('SELECT doc FROM nss WHERE id = 1')
for $e in $d//element()
return
<element>
  {fn:local-name($e), fn:name($e), fn:namespace-uri($e)}
</element>;
```

```
result:
<element>
sample sample http://namespace.sample.com/one
</element>
<element>
eone eone http://namespace.sample.com/one
</element>
<element>
eone two:eone http://namespace.sample.com/two
</element>
```

```
XQUERY
let $d := db2-fn:sqlquery('SELECT doc FROM nss WHERE id = 1')
for $e in $d//attribute()
return
<attribute>
  {fn:local-name($e), fn:name($e), fn:namespace-uri($e)}
</attribute>;
```

```
result:
<attribute>
anoone anoone
</attribute>
<attribute>
atwo two:atwo http://namespace.sample.com/two
</attribute>
```

Note that there is no namespace URI for the `anoone` attribute because the default namespace is not applied for the attributes.

The in-scope-prefixes function

This function returns a list of all prefixes that are in scope for a given element. In Example 4-94 we use the `in-scope-prefixes` function to display all prefixes that are in scope for the sample element from the `ns-sample.xml` document.

Example 4-94 Using in-scope-prefixes function

```
XQUERY
declare default element namespace "http://namespace.sample.com/one";let
$d := db2-fn:sqlquery('SELECT doc FROM nss WHERE id = 1')
return fn:in-scope-prefixes ($d/sample);
```

result:

```
xml
two
```

The result includes three prefixes. The `xml` prefix is always in-scope. If there is a default namespace, it is presented with a string with length zero. In the result, it is the first record.

QName and node-name functions

The *QName* function returns an expanded name (of `xs:QName` type) that is constructed from a namespace URI and a string that contains a lexical *QName*. In Example 4-95 we show how to use the *QName* function. The returned value is an `xs:QName` value with namespace URI of `"http://sample.name.space.com"`, a prefix of `"sampleprefix"`, and local name `"localname"`.

Example 4-95 Using QName function

```
XQUERY
fn:QName ("http://sample.name.space.com", "sampleprefix.localname");
```

result:

```
sampleprefix.localname
```

The *node-name* function returns the expanded name (of `xs:QName` type) for the given node of an XML document.

local-name-from-QName and namespace-uri-from-QName functions

The *local-name-from-QName* function returns the local part of an `xs:QName` value. The *namespace-uri-from-QName* function returns the namespace URI

part of an `xs:QName` value. In Example 4-96, we demonstrate the use of these two functions. We display the local names and the namespace URIs for all elements in the `ns-sample.xml` document.

Example 4-96 Using local-name-from-QName and namespace-uri-from-QName

```
XQUERY
let $d := db2-fn:sqlquery('SELECT doc FROM nss WHERE id = 1')
for $e in $d//element()
return
<element>
  {fn:local-name-from-QName(fn:node-name($e)),
   fn:namespace-uri-from-QName(fn:node-name($e))}
</element>;

result:
<element>
sample http://namespace.sample.com/one
</element>
<element>
eone http://namespace.sample.com/one
</element>
<element>
eone http://namespace.sample.com/two
</element>
```

4.5 Getting XML data in and out of database

DB2 supports XML as an internal data type. You can define a column as XML data type when creating a table. XML document stored in the column can be accessed like data defined as other data type such as integer using INSERT, UPDATE, DELETE, and SELECT statements.

Inserting XML data

Using SQL INSERT statement to insert data into an XML column is as simple as insert other data type into a relational table column. The value to be inserted can be retrieved from an exiting XML data in another table or created using XML constructor or publishing functions. In Example 4-97, we retrieve an XML document from an XML column and insert it into another XML column.

Example 4-97 Inserting XML data from another table

```
INSERT INTO nss (id, doc)
```



```
VALUES (101, (SELECT doc FROM xps WHERE id = 1));
```

In Example 4-98, we use XML publishing functions to construct the XML value then insert it into an XML column.

Example 4-98 Inserting XML data using constructed data

```
INSERT INTO nss (id, doc)
VALUES (102, XMLDOCUMENT(XMLELEMENT(name "Test", 'Test Element')));
```

Another option of preparing XML data to be inserted into an XML column is using a string representation of an XML document. In any case, DB2 ensures that the provided value is a well-formed XML document.

XML parsing

XML parsing is the process of transforming XML data from the string representation to a hierarchical (tree-like) format. Parsing can be implicit or explicit. XMLPARSE function is used to explicitly check the correctness of an XML value in an SQL statement. If an XMLPARSE function is not specified in an INSERT statement, it will be used implicitly to make sure that the provided value is a well-formed XML document. In Example 4-99, we show the INSERT statement with and without explicit call to XMLPARSE function.

Example 4-99 Implicit and explicit parsing

```
INSERT INTO nss (id, doc) VALUES (103,
'<sample>with no explicit parsing</sample>');
```

```
INSERT INTO nss (id, doc) VALUES (104, XMLPARSE (DOCUMENT
'<sample>with explicit parsing</sample>'));
```

The syntax of the XMLPARSE function is:

```
XMLPARSE (DOCUMENT <string value> [PRESERVE|STRIP WHITESPACE])
```

XMLPARSE provides two options to control whitespace processing:

- ▶ **STRIP WHITESPACE:** Removes the extra whitespace. This is the default option.
- ▶ **PRESERVE WHITESPACE:** Preserves the whitespace in the string value.

In Example 4-100, we illustrate the use of PRESERVE WHITESPACE and STRIP WHITESPACE. The XML document we insert with each of these options contains several spaces that are kept or deleted depending on the option we provide for WHITESPACE.

Example 4-100 PRESERVE and STRIP WHITESPACE options

```
INSERT INTO nss (id, doc) VALUES (105, XMLPARSE (DOCUMENT
'<sample>                                </sample>' PRESERVE WHITESPACE));
INSERT INTO nss (id, doc) VALUES (106, XMLPARSE (DOCUMENT
'<sample>                                </sample>' STRIP WHITESPACE));
XQUERY db2fn-sqlquery('SELECT doc FROM nss WHERE id = 105');
XQUERY db2fn-sqlquery('SELECT doc FROM nss WHERE id = 106');
```

```
results:
<sample>
</sample>

<sample/>
```

XML validation

XML validation is the process of checking whether the structure, data types, and content of an XML document are valid. XML validation adds type annotations to XML elements, attributes, and values. It also strips off ignorable whitespace in an XML document. The ignorable whitespace is whitespace that can be eliminated from an XML document. The XML schema document determines which whitespace is ignorable whitespace.

The XMLVALIDATE function is used to validate an XML document against a schema document. The schema defines the structure of the XML document, including node types, default values, and multiple occurrences. Before using a schema document, it has to be registered (see 5.2, “Schema management” on page 198 for more details). Example 4-101 shows the use of the XMLVALIDATE function.

Example 4-101 Validating XML document using XMLVALIDATE function

```
INSERT INTO nss (id, doc) VALUES (105, XMLVALIDATE (XMLPARSE (DOCUMENT
'<sample>with explicit parsing</sample>')
ACCORDING TO XMLSCHEMA ID schemaid))
```

Here we suppose that schema ID is a registered XML schema. If we pass as an argument a value of character data type, it will be implicitly parsed before validation.

XML encoding

The encoding of XML data can be determined in two ways:

- It can be derived from the data itself, which is known as internally encoded data.

- ▶ It can be derived from external sources, which is known as externally encoded data.

The application data type that is used to exchange the XML data between the application and an XML column determines how the encoding is derived.

- ▶ XML data that is in character or graphic application data types is considered to be externally encoded. XML data that is in these data types is encoded in the application code page.
- ▶ XML data that is in binary application data type or binary data that is in a character data type is considered to be internally encoded. With internal encoding, the content of the data determines the encoding.

DB2 derives the internal encoding from the document content according to the XML standard. It can be derived from three components:

- ▶ Unicode Byte Order Mark (BOM):
BOM is a byte sequence of Unicode character codes. It is in the beginning of XML data. The BOM indicates the byte order of data followed. DB2 recognizes BOM only for XML data type. For XML data that is stored in a column of a non XML data type, it is treated as an ordinary data.
- ▶ XML declaration:
An XML document can contain processing instructions that provide specific information about the XML data.
- ▶ Encoding declaration:
This is an optional part of an XML declaration that specifies the encoding used in the document.

Here, we list some encoding considerations for placing XML data into a database:

- ▶ If you have externally encoded XML data (data sent to the DB2 server using character data types):
 - If its internal and external encoding is not Unicode, be sure that any internally encoded declarations match the external encoding. Otherwise, DB2 rejects the XML document.
 - If both internal and external encoding are Unicode, but the encoding schema does not match, DB2 ignores the internal encoding.
- ▶ If you have internally encoded XML data (data sent to the DB2 server using binary data types):
 - The sending application must ensure that data contains the correct encoding information.

Shredding XML data into relational tables

DB2 provides a functionality to decompose or shred XML documents into columns of relational tables. This feature is very useful when we have or receive XML data that we have to store in relational tables.

DB2 uses an annotated XML schema document to describe decomposition rules. An XML schema document describes the structure of an XML document. In annotated schema decomposition, the process of decomposition (or shredding) is controlled by annotations added to the schema. These annotations provide information such as the name of the target tables and columns where the XML data has to be stored.

In this section, we use a very simple example of annotated schema decomposition to show all the steps that must be completed in order to use this functionality.

Suppose that we have an XML document containing data about the time required to deliver an order to a given zip code. To shred that XML data into a relational table, follow these steps:

1. XML data:

We have to know what content from the XML document is to be stored in relational tables. Example 4-102 shows a simple XML file that we want to shred. We want to decompose and store the information contained in both Zip and Duration elements.

Example 4-102 XML document to be shredded t.xml

```
<DeliveryTimes>
  <Entry>
    <Zip>12345</Zip>
    <Duration>2</Duration>
  </Entry>
  <Entry>
    <Zip>23456</Zip>
    <Duration>4</Duration>
  </Entry>
  <Entry>
    <Zip>34567</Zip>
    <Duration>6</Duration>
  </Entry>
</DeliveryTimes>
```

2. Tables and columns:

We decide on the relational tables into which we will shred the XML data. In our example, we use the TRTIME table defined in Example 4-84 on page 132. This table has two columns: ZIPCODE and DURATION, and we will use them to store the information from Zip and Duration elements from our XML document.

3. Schema:

We require a schema document that describes the structure of the XML data. Example 4-103 here shows a schema for the XML file given in Example 4-102 on page 146. This schema states that the root element of the document is DeliveryTimes, which can contain several Entry elements. Every Entry element contains a Zip element and a Duration element.

Example 4-103 Schema document s.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="DeliveryTimes">
    <xsd:complexType>
      <xsd:sequence maxOccurs="unbounded">
        <xsd:element name="Entry">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="Zip" type="xsd:string"/>
              <xsd:element name="Duration"
type="xsd:integer"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

4. Adding annotations to our schema:

This step describes in which column to store which XML data. It is done by adding annotations to our schema. In our example, the annotations are very simple. We specify that the Zip element will be stored in the ZIP_CODE column of the TRTIME table and that the Duration element will be stored in the DURATION column of the TRTIME table. Annotations can be used for describing much more complex shredding rules. For more information, refer to DB2 XML Guide, SC10-4254.

In Example 4-104 we show our schema with the added annotations. Note that you have to replace the value of `<db2-xdb:defaultSQLSchema>` with the DB2 schema you are using.

Example 4-104 Schema document with annotations sa.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:db2-xdb="http://www.ibm.com/xmlns/prod/db2/xdb1">
  <xsd:annotation>
    <xsd:appinfo>
      <db2-xdb:defaultSQLSchema>ADMIN</db2-xdb:defaultSQLSchema>
    </xsd:appinfo>
  </xsd:annotation>
  <xsd:element name="DeliveryTimes">
    <xsd:complexType>
      <xsd:sequence maxOccurs="unbounded">
        <xsd:element name="Entry">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="Zip" type="xsd:string"
                db2-xdb:rowSet="TRTIME"
                db2-xdb:column="ZIP_CODE"/>
              <xsd:element name="Duration" type="xsd:integer"
                db2-xdb:rowSet="TRTIME"
                db2-xdb:column="DURATION"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

5. Registering the schema to DB2 XML Schema Repository (XSR):

The DB2 XML Schema Repository is described in more detail in 5.2, “Schema management” on page 198. We use the command shown in Example 4-105 to register our annotated XML schema and to specify that it will be used for decomposition.

Example 4-105 Registering sa.xsd schema

```
REGISTER XMLSCHEMA 'd:/work/db2/sa.xsd'
FROM D:\work\db2\sa.xsd
AS ADMIN.TEST_SHR COMPLETE ENABLE DECOMPOSITION;
```

6. Shredding XML data:

Now we are ready for shredding. We use the DECOMPOSE command as shown in Example 4-106.

Example 4-106 Shredding s.xml file

```
DECOMPOSE XML DOCUMENT d:/work/db2/s.xml XMLSCHEMA admin.test_shr
```

Issuing the command shown in Example 4-107 could check that the data from the s.xml file is inserted into the TRTIME table.

Example 4-107 Verifying results of decomposition

```
SELECT * FROM TRTIME
```

results:

ZIP_CODE	DURATION
95030	5
95035	4
95134	3
12345	2
23456	4
34567	6

XML data serialization

XML serialization is the process of converting XML data from its hierarchal representation in the XQuery and XPath data model to the serialized string format.

DB2 can perform serialization implicitly, or it can be explicitly invoked using the XMLSERIALIZE function. The most common use of XML serialization is when transferring XML data from a DB2 server to a DB2 client. Implicit serialization is often preferred because it is simpler and it is more efficient to send data to the client as XML data. However, under the following circumstances, it is better to use explicit serialization:

- ▶ If the XML documents are very large. You can use XMLSERIALIZE to convert XML data to the LOB type. The reason is that there are no XML locators. After converting to LOB, you can use LOB locators.
- ▶ If the client does not support the XML data type. DB2 server implicit serialization converts XML data to BLOB (by default) or CLOB. If you require the retrieved data to be converted to some other data type, you should use explicit serialization.

The most suitable data type for converting XML data is BLOB because retrieval of binary data minimizes encoding problems.

XML publishing functions

XML publishing functions are used to construct XML nodes and documents. Both relational and XML data can be used. Publishing functions are also called *constructor functions*.

XMLELEMENT function

This function creates an XML element node. The arguments include an element name, optional namespace declarations, optional attributes, and zero or more expressions that are the element's content. In Example 4-108, we show how to create an XML element using the XMLELEMENT function. Note that the XMLELEMENT function can be nested.

Example 4-108 Using XMLELEMENT function

```
VALUES (XMLEMENT (name "Name",  
                XMLEMENT (name "FirstName", 'Steve'),  
                XMLEMENT (name "LastName", 'Ferrington')));
```

Result:

```
<Name>  
  <FirstName>Steve</FirstName>  
  <LastName>Ferrington</LastName>  
</Name>
```

XMLATTRIBUTES function

This function creates an attribute node. It can be called only as an argument of the XMLEMENT function. In Example 4-109 we demonstrate how to use the XMLATTRIBUTES function to add an attribute gender to Name element. You can create several attributes using one call to the XMLATTRIBUTES function.

Example 4-109 Using XMLATTRIBUTES function

```
VALUES (XMLEMENT (name "Name", XMLATTRIBUTES ('MALE' as "gender"),  
                XMLEMENT (name "FirstName", 'Steve'),  
                XMLEMENT (name "LastName", 'Ferrington')))
```

results:

```
<Name gender="MALE">  
  <FirstName>Steve</FirstName>  
  <LastName>Ferrington</LastName>  
</Name>
```

XMLFOREST function

This function creates a sequence (forest) of XML element nodes. In Example 4-110 here we use the XMLFOREST function to produce the same result as in Example 4-109 on page 150.

Example 4-110 Using XMLFOREST function

```
VALUES (XMLELEMENT (name "Name",
    XMLATTRIBUTES ('MALE' as "gender"),
    XMLFOREST ('Steve' as "FirstName",
        'Ferrington' as "LastName"))))
```

results:

```
<Name gender="MALE">
  <FirstName>Steve</FirstName>
  <LastName>Ferrington</LastName>
</Name>
```

XMLDOCUMENT function

This function creates an XML document node. Every XML document stored in an XML column must have a document node. It contains the root XML element and optional comments and processing instructions. A document node is not visible in the serialized string representation of XML. In Example 4-111, we show the usage of the XMLDOCUMENT function. We insert an XML document into the NSS table. Note that if we omit the XMLDOCUMENT function (and use only the XMLELEMENT function), there will be an error.

Example 4-111

```
INSERT INTO nss (id, doc) VALUES (999, XMLDOCUMENT(
    XMLELEMENT (name "Name",
        XMLATTRIBUTES ('MALE' as "gender"),
        XMLFOREST ('Steve' as "FirstName",
            'Ferrington' as "LastName"))));
```

XMLNAMESPACES function

This function creates namespace declarations. It can be called only as an argument from the XMLELEMENT, XMLFOREST, and XMLTABLE functions. In Example 4-112, we use the XMLNAMESPACES function to define the default namespace while creating the Name element.

Example 4-112 Using XMLNAMESPACES function

```
VALUES (XMLELEMENT (name "Name",
    XMLNAMESPACES (DEFAULT 'http://sample.default.namespace.com'),
```

```
XMLATTRIBUTES ('MALE' as "gender"),
XMLFOREST ('Steve' as "FirstName",
           'Ferrington' as "LastName")))))
```

results:

```
<Name xmlns="http://sample.default.nspace.com" gender="MALE">
  <FirstName>Steve</FirstName>
  <LastName>Ferrington</LastName>
</Name>
```

XMLCONCAT function

This function creates a sequence of variable number of XML input arguments. In Example 4-113 here, we use the XMLCONCAT function to produce the same result as in Example 4-112 on page 151.

Example 4-113 Using XMLCONCAT function

```
VALUES (XMLELEMENT (name "Name",
  XMLNAMESPACES (DEFAULT 'http://sample.default.nspace.com'),
  XMLATTRIBUTES ('MALE' as "gender"),
  XMLCONCAT(
    XMLELEMENT (name "FirstName", 'Steve'),
    XMLELEMENT (name "LastName", 'Ferrington'))))
```

Results:

```
<Name xmlns="http://sample.default.nspace.com" gender="MALE">
  <FirstName>Steve</FirstName>
  <LastName>Ferrington</LastName>
</Name>
```

XMLCOMMENT function

This function creates a comment node. We show its usage in Example 4-114.

Example 4-114 Using XMLCOMMENT function

```
VALUE(XMLELEMENT (name "Name", XMLATTRIBUTES ('MALE' as "gender"),
  XMLCOMMENT ('Comment line'),
  XMLELEMENT (name "FirstName", 'Steve'),
  XMLELEMENT (name "LastName", 'Ferrington')))
```

Result:

```
<Name gender="MALE">
  <!--Comment line-->
```

```
<FirstName>Steve</FirstName>
<LastName>Ferrington</LastName>
</Name>
```

XMLPI function

This function creates a processing instruction node. We show the usage of the XMLPI function in Example 4-115.

Example 4-115 Using XMLPI function

```
VALUE(XMLELEMENT (name "Name", XMLATTRIBUTES ('MALE' as "gender"),
XMLPI (name "Instruction", 'Do nothing'),
XMLELEMENT (name "FirstName", 'Steve'),
XMLELEMENT (name "LastName", 'Ferrington')))
```

Results:

```
<Name gender="MALE">
  <?Instruction Do nothing?>
  <FirstName>Steve</FirstName>
  <LastName>Ferrington</LastName>
</Name>
```

XMLTEXT function

This function creates a text node. We illustrate its usage in Example 4-116.

Example 4-116 Using XMLTEXT function

```
VALUE(XMLELEMENT (name "Name",
XMLELEMENT (name "FirstName", XMLTEXT ('Steve')),
XMLELEMENT (name "LastName", XMLTEXT ('Ferrington'))))
```

Results;

```
<Name>
  <FirstName>Steve</FirstName>
  <LastName>Ferrington</LastName>
</Name>
```

4.6 XML full-text search

The DB2 9 hybrid database system provides a powerful feature to store the XML document natively. The XQuery support allows users to search data with XML documents. However, in the XQuery Data Model, text is a node and the XQuery function offers only a simple substring match. For XML documents that contain significant portions of text, XQuery cannot search the content easily.

DB2 Net Search Extender (NSE) is a full-text search¹ solution providing an efficient way to search unstructured portions of documents. Using DB2 NSE, a text search requirement such as this can be performed easily:

...find all XML documents with the term 'pureXML' in the ABSTRACT element, the phrase 'DB2 version 9.1' in the TITLE element, where the terms must be in the same sentence.

4.6.1 DB2 Net Search Extender

DB2 Net Search Extender V9 is enhanced to support the new XML Data Type in DB2. All existing text search functionality offered by NSE is also available for natively stored XML documents with DB9. Figure 4-4 illustrates the Net Search Extender architecture.

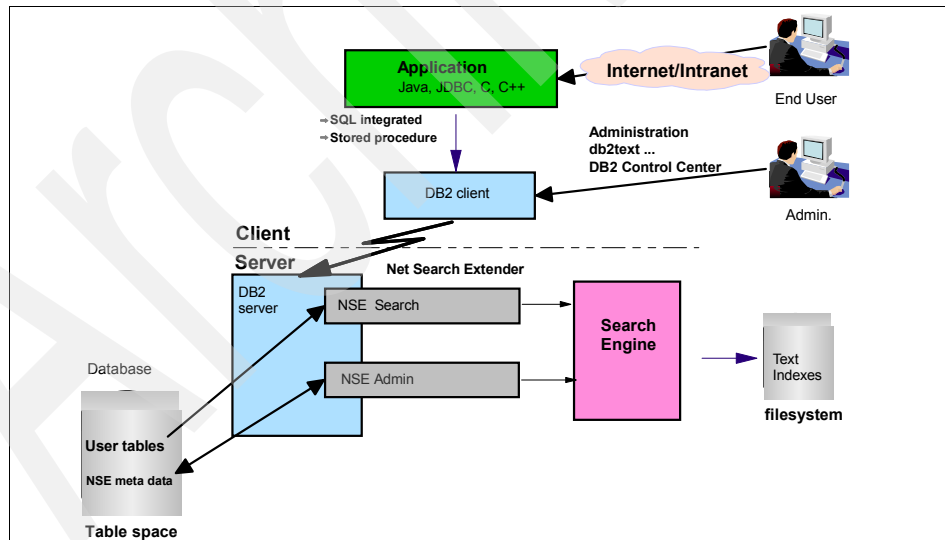


Figure 4-4 Net Search Extender architecture

¹ Portions of this chapter are excerpted from the article *XML full-text search in DB2* by Holger Seubert and Sabine Perathoner, originally published in IBM developerWorks, June 2006.
<http://www.ibm.com/developerworks/db2/library/techarticle/dm-0606seubert/index.html>

DB2 NSE offers an efficient and intelligent method for searching full-text documents stored in a DB2 database using SQL queries. Instead of sequentially searching through the text data using string matching, like the way search is done with the XQuery `contain()` function, DB2 NSE searches textual data that is stored in the column of a DB2 database table using a text index, which typically consists of significant terms that are extracted from the text document. With XML data, the significant terms and their locations in the XML document structure are maintained in a text index.

DB2 NSE is a separately installed feature that is shipped with DB2 9. Its text search capability is integrated into SQL and optimized by the DB2 optimizer for run time. The NSE administration function can be invoked from the DB2 Control Center to prepare administrative tasks such as creating, updating, or deleting text indexes.

4.6.2 Preparing the instance for text search

Once the DB2 Net Search Extender is installed, you have to perform four steps to prepare the DB2 environment for full-text searching on an XML document. You can use the NSE command or GUI interface in the DB2 Control Center to perform these four steps:

1. Starting DB2 Net Search Extender services
2. Enabling the database for full-text search
3. Creating a full-text index
4. Updating the previously created full-text index

We walk you through these steps using a simple table defined in Example 4-117. The `COMMENT` column stores the customer feedback in XML documents.

Example 4-117 Defining FEEDBACK table

```
CREATE TABLE FEEDBACK ("APPL_ID" INTEGER primary key not null,  
"COMMENT" XML);
```

Example 4-118 shows the five sample customer feedback XML documents.

Example 4-118 Sample data for comment column

```
---comment1.xml-----  
<?xml version="1.0" encoding="UTF-8"?>  
<feedback xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
xsi:noNamespaceSchemaLocation="C:\Documents and  
Settings\chnglme.T40-92U-V46\IBM\rational\sd6.0\workspace\aaa\WebContent\WEB-INF\feedback.xsd">  
  <entry>
```

```

        <dateOfEntry> 2005-07-07 </dateOfEntry>
        <rating>4</rating>
        <comment>Excellent on-line application.</comment>
    </entry>
    <entry>
        <dateOfEntry> 2005-09-12 </dateOfEntry>
        <rating>2</rating>
        <comment>I still have not heard from the bank. Not sure the status
of my application. Slow response time.</comment>
    </entry>
</feedback>

```

```

--- comment2.xml -----
<?xml version="1.0" encoding="UTF-8"?>
<feedback xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <entry>
        <dateOfEntry> 2005-08-09 </dateOfEntry>
        <rating>5</rating>
        <comment lan="en">Quick approve time.</comment>
    </entry>
</feedback>

```

```

--- comment3.xml ---
<?xml version="1.0" encoding="UTF-8"?>
<feedback xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <entry>
        <dateOfEntry> 2005-08-09 </dateOfEntry>
        <rating>5</rating>
        <comment lan="en">Good service and quick Response time. I will
recommand to others.</comment>
    </entry>
</feedback>

```

```

--- comment4.xml ---
<?xml version="1.0" encoding="UTF-8"?>
<feedback xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <entry>
        <dateOfEntry> 2005-08-16 </dateOfEntry>
        <rating>3</rating>
        <comment lan="en">There should be more loan prodcuts.</comment>
    </entry>
</feedback>

```

```

--- comment5.xml ---
<?xml version="1.0" encoding="UTF-8"?>

```

```
<feedback xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <entry>
    <dateOfEntry> 2005-08-21 </dateOfEntry>
    <rating>2</rating>
    <comment lan="en" >The interests rate is too high.</comment>
  </entry>
</feedback>
```

Example 4-119 shows the INSERT statement used to insert five customer feedbacks.

Example 4-119 SQL statement to insert data into the table FEEDBACK.

```
INSERT INTO feedback VALUES( 100, xmlparse(document '<comment1.xml>'
Preserve whitespace));
INSERT INTO feedback VALUES( 200, xmlparse(document '<comment2.xml>'
Preserve whitespace));
INSERT INTO feedback VALUES( 300, xmlparse(document '<comment3.xml>'
Preserve whitespace));
INSERT INTO feedback VALUES( 400, xmlparse(document '<comment4.xml>'
Preserve whitespace));
INSERT INTO feedback VALUES( 500, xmlparse(document '<comment5.xml>'
Preserve whitespace));
```

Step 1: Starting DB2 Net Search Extender services

Before you can use DB2 NSE, you must start the DB2 NSE instance services. You can start Net Search Extender instance services by entering the following command in the OS Command Prompt or the DB2 Command Window:

```
DB2TEXT START
```

All text index administrative task commands start with **db2text**. If you require the syntax for text index administrative tasks, you can display a list of **db2text** commands by issue:

```
DB2TEXT ?
```

For syntax of an individual command, you can execute the following command:

```
DB2TEXT ? command
```

For example:

```
DB2 TEXT ? alter index
```

Step 2: Enabling the database for full-text search

After starting Net Search Extender, enable the DB2 database for text search operations. This step creates necessary administration tables and various user defined functions (UDFs) and stored procedures that are required for full-text search on DB2 data, and is executed only once per database. The following command enables DB2 database XMLRB for text search.

```
DB2TEXT ENABLE DATABASE FOR TEXT CONNECT TO xmlrb
```

Step 3: Creating a full-text index

After the database is enabled for text, you can create text indexes on columns storing textual data in various format. The create index command establishes the text index infrastructure by defining and declaring the properties for the text index. In our example, we use a basic text index creation. The create index command creating a full-text index name 'ind1' on the XML documents that are stored natively in XML column COMMENT within FEEDBACK table is:

```
DB2TEXT CREATE INDEX ind1 FOR TEXT ON feedback(comment) CONNECT TO  
xmlrb
```

Note that `CONNECT TO databaseName` is a required clause in all DB2 NSE administrative commands unless the database you are enabling for text search is set in environment variable `DB2DBDFT`.

Step 4: Update the previously created full-text index

After creation, the text index does not contain any data. You must explicitly update the index if no update frequency is specified at index creation time. Index update is the process of adding data to the text index. The initial index update adds all text documents from the text column to the index and it is typically executed after you have created a text index. We use the following command to update the index:

```
DB2TEXT UPDATE INDEX ind1 FOR TEXT CONNECT TO xmlrb
```

The update text index process can be done automatically or manually. An automatic index update can be specified as an index attribute during index creation. It can also be specified after the index has been created using the alter index command.

Now that we have prepared the DB2 database for full-text search, we are ready to move on to full-text search with XML data.

4.6.3 Full-text searching using DB2 NSE

The following items are among the basic search criteria that DB2 NSE features:

- ▶ Boolean operations for conjunction (AND), disjunction (OR), and exclusion (NOT) of search terms.
- ▶ Proximity search searches for words in the same sentence or paragraph
- ▶ Fuzzy search searches for words with similar spellings as the search term.
- ▶ Wildcard search that uses front, middle, and end character masking.

Other advanced search features offered by DB2 NSE include these:

- ▶ Thesaurus search support is provided for broader queries. This means not only searching for a specific term, but also for the additional terms that are related to it with user-defined relations.
- ▶ Numeric attribute search now searches on numeric ranges that could be in a structured document or within additional columns.
- ▶ Stemming search is used to reduce the search term to its word stem before the search is carried out. This is supported for English only.

DB2 NSE also offers section based search with these options:

- ▶ Limiting the search to XML elements
- ▶ Limiting the search to XML attributes
- ▶ Support for mixed content types of XML elements.

Next, we explore each of these capabilities, including examples.

Search using Net Search Extender CONTAIN() function

The CONTAIN() scalar function searches for text in an XML document indexed by DB2 NSE. Using the CONTAIN() function is the most commonly used method for performing full-text search with DB2 NSE where standard SQL would be used. You can combine it with other conditions in an SQL WHERE clause.

The syntax of CONTAIN() function is as follows:

```
CONTAIN(column- name, search-argument)
```

The CONTAIN() scalar function takes two parameters:

- ▶ The *column-name* is the name of a column that must have an associated text index.
- ▶ The search-argument is a string of type VARCHAR containing the terms to be searched.

CONTAIN() returns the INTEGER value 1 if the document contains the text, or any relation indicated in the search argument. Otherwise, it returns 0 (zero).

Example 4-120 shows how to use the scalar CONTAIN() function to perform full-text search.

Example 4-120 The scalar CONTAINS() function

```
SELECT column
FROM table
WHERE CONTAINS(column-name, 'search-argument')=1
```

Text search using SQL and XQuery

Example 4-121 shows a basic text search query that performs a simple text search on the FEEDBACK table that returns all application ID with the term “good” existing somewhere in the document structure of the comment information.

Example 4-121 Text searching using SQL Use of NSE function in combination with XMLQUERY()

```
SELECT appl_id
FROM feedback
WHERE CONTAINS(comment, ' "good" ')=1
```

Example 4-122 shows how to use XMLQUERY in a query. The query returns the “comment” element of the feedback that contains the term “good”.

Example 4-122 Using NSE function in combination with XMLQUERY()

```
SELECT XMLQUERY('$com//comment' passing comment as "com")
FROM feedback
WHERE CONTAINS(comment, ' "good" ')=1
```

The query in Example 4-122 can be written in XQuery as shown in Example 4-123.

Example 4-123 Using text search functionality in XQUERY

```
XQUERY for $com in db2-fn:sqlquery("select comment from feedback
WHERE CONTAINS(comment, ' &quot;good&quot; ')=1")
return $com//comment
```

Limiting the result to XML elements or attributes

In DB2 NSE, limiting text search to specific elements or sub-trees of the XML document is expressed by a fully qualified XPath. This is one of the most frequently used features for XML full-text search. The XPath indicates the part of an XML document that the search should be carried out.

DB2 NSE supports the abbreviated XPath location-step syntax, the use of the child axis (/), and the attribute axis (@). No other XPath expression or functions are supported.

Example 4-124 shows a query to search for the term 'good' and 'service' that is limited to the COMMENT element of the XML document.

Example 4-124 Limit search with XMLQUERY

```
SELECT XMLQUERY('$com//comment' passing comment as "com")
FROM feedback
WHERE CONTAINS(comment, ' section("/feedback/entry/comment")
("good","service") ')=1
```

The SECTION clause used as part of the search argument indicates the part within the XML document where text search occurs. The query in Example 4-124 can be written in XQuery as shown in Example 4-125.

Example 4-125 Limit search with XQuery

```
XQUERY for $com in db2-fn:sqlquery("select comment from feedback
WHERE CONTAINS(comment, 'section(&quot;/feedback/entry/comment&quot;;)
(&quot;good&quot;;&quot;service&quot;;) ')=1")
return $com//comment
```

In addition, text search can be limited to specific XML attributes. Example 4-126 shows how to use the attribute axis in the query.

Example 4-126 Limit search on XML attributes

```
SELECT XMLQUERY('$com//rating' passing comment as "com")
FROM feedback
WHERE CONTAINS(comment, ' section("/feedback/entry/comment/@lan") "en"
')=1;
```

Example 4-127 shows how searching in different sections can be combined using the AND Boolean operator (&).

Example 4-127 searching with AND Boolean operator

```
SELECT XMLQUERY('$com//rating' passing comment as "com")
FROM feedback
WHERE CONTAINS(comment, ' section("/feedback/entry/comment/@lan") "en"
& section("/feedback/entry/rating")"5")=1;
```

4.6.4 Taking advantage of Net Search Extender text search features

DB2 Net Search Extender offers several other features to define text search criteria. For instance, with proximity search, you can choose to limit the text search to terms that match only if they occur in the same sentence.

Proximity search

The query in Example 4-128 finds all the application ID of those feedbacks with the terms “good” and “quick response” not only in the COMMENT element of the XML document, but also in the same sentence.

Example 4-128 Search for the terms “good” and “quick response” in the same sentence

```
select appl_id from feedback where
CONTAINS(comment, ' section("/feedback/entry/comment") "good" in same
sentence as "quick response")=1
```

There are two constraints in Example 4-128. The first one is “CONTAINS(comment, ' section("/feedback/entry/comment")””. It limits the search in the COMMENT element. The second one is “"good" in same sentence as "quick response". It searches for the terms “good” and “quick response” in the same sentence.

Example 4-129 shows how to search using tokens instead of phrases.

Example 4-129 Token search

```
select appl_id from feedback where
CONTAINS(comment, ' section("/feedback/entry/comment") "good" in same
sentence as ("quick", "response")'=1
```

Boolean operators

Using Boolean operators AND (&), OR (|) and NOT, you can combine different search terms with other search terms. Example 4-130 combines several search terms by using the Boolean operators AND and OR.

Example 4-130 Using Boolean operators AND and OR

```
select appl_id from feedback where
CONTAINS(comment, ' section("/feedback/entry/comment") "good" & "quick"
| "response" '=1
```

The query in Example 4-130 returns the application ID of those feedbacks having the term “good” and “quick” or “response” in the element comment.

Using the Boolean operator NOT, you can exclude particular terms from the search result. For example, the following query in Example 4-131 searches for document having the term “good” and “quick” and excluding the term “response” in the COMMENT element.

Example 4-131 Using Boolean operator NOT

```
select appl_id from feedback where  
CONTAINS(comment, ' section("/feedback/entry/comment") "good" & "quick"  
& NOT "response" ')=1
```

Fuzzy search

This search finds documents that contain the search term spelled in a similar way to the specified search term. The match level indicates the desired degree of accuracy. Fuzzy search is normally used when misspellings are possible in the document. You can specify values between 1 to 100 to show the degree of accuracy, where 100 is an exact match.

Example 4-132 is a fuzzy search example. The term response is misspelled purposefully. In the fuzzy search, a document containing the term response is found. The degree of accuracy is 60 in this example.

Example 4-132 Query uses fuzzy search to find similar spelled terms

```
select appl_id from feedback where  
CONTAINS(comment, ' section("/feedback/entry/comment") fuzzy form of 60  
"response" &  
"good" ')=1
```

Stemming search

The stemming search, which is used to search for the stemmed form of a term, causes the term to be reduced to its word stem before the search is carried out. This form of search is not case-sensitive. Currently, only English stemming is supported and the word must follow regular inflection endings.

Example 4-133 shows searching for the fuzzy form of the term response or the stemmed form of "goody". The stemmed search returns documents that have terms such as good and goods.

Example 4-133 Query uses stemmed search

```
select appl_id from feedback where  
CONTAINS(comment, ' section("/feedback/entry/comment") fuzzy form of 60  
"response" | stemmed form of "goody" ')=1
```

Wildcard search

Wildcard search is also known as character masking search. DB2 NSE uses two masking characters:

- ▶ Percent (%) masks any number of arbitrary characters.
- ▶ Underscore (_) masks any single character in a search term.

DB2 NSE uses these masking characters for wildcard search the same way the DB2 predicate LIKE uses them.

A sample query using wildcard search is shown in Example 4-134.

Example 4-134 Query uses wildcard search

```
select appl_id from feedback where  
CONTAINS(comment, ' section("/feedback/entry/comment") "respon_e" &  
"qui%"')=1
```

4.6.5 Full-text search considerations

One very import aspect of full-text search that we want to emphasize when performing search within XML documents is: A search result always returns a complete XML document and not the specific document part or element where the hit encounters. Considering the query searching for the term 'Slow' in Example 4-135.

Example 4-135 Sample query searches for "Slow" in element comment

```
SELECT appl_id FROM feedback WHERE CONTAINS(comment, '  
section("/feedback/entry/comment") "Slow"')=1
```

Example 4-135 returns the application ID of those feedbacks containing the term Slow in the element COMMENT. Example 4-136 is the result of this query.

Example 4-136 Query result

```
APPL_ID  
-----  
100  
  
1 record(s) selected.
```

Suppose we want to discern which date the customer of application ID 100 entered the comment with the term `Slow` in it. Example 4-137 shows the XML query.

Example 4-137 Searching date that customer enters comment with the term "Slow"

```
SELECT XMLQUERY('$com//dateOfEntry' passing comment as "com") FROM
feedback WHERE CONTAINS(comment, ' "Slow" ')=1 and APPL_ID=100
```

Example 4-138 shows the result of the query in Example 4-137. The result shows two dates, 2005-07-07 and 2005-09-12. The customer of application ID 100 has multiple entries of comments. He only enters the term `"Slow"` on date 2005-09-12, but both dates 2005-07-07 and 2005-09-12 show up because the search always returns a complete XML document, as shown in Example 4-138.

Example 4-138 Query result

```
<dateOfEntry xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
2005-07-07 </dateOfEntry>
<dateOfEntry xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  2005-09-12 </dateOfEntry>
```

1 record(s) selected.

Example 4-139 shows another way that the query in Example 4-138 can be written in XQuery. The XQuery also returns both dates 2005-07-07 and 2005-09-12.

Example 4-139 Sample XQuery

```
XQUERY for $com in db2-fn:sqlquery("select comment from
feedback WHERE APPL_ID=100 and CONTAINS(comment, '
section(&quot;/feedback/entry/comment&quot;;)
stemmed form of &quot;Slow&quot;; ')=1") return $com//dateOfEntry
```

You can use full-text search to filter the documents having the term `Slow` in the `COMMENT` element as shown in Example 4-140. The search returns the complete XML documents that satisfied the search condition.

Example 4-140 Filter documents

```
SELECT
XMLQUERY('$com/feedback/entry[contains(comment,"Slow")]/dateOfEntry'
passing comment as "com") FROM feedback WHERE CONTAINS(comment,
'section("/feedback/entry/comment") "Slow")=1 and APPL_ID=100
```

The same query as in Example 4-140 on page 165 can be expressed in XQuery context, as shown in Example 4-141.

Example 4-141 Same query expressed in XQuery

```
XQUERY for $com in db2-fn:sqlquery("select comment from feedback WHERE
APPL_ID=100 and CONTAINS(comment,
'section(&quot;/feedback/entry/comment&quot;)&quot;Slow&quot;')=1")
return $com/feedback/entry[contains(comment,"Slow")]/dateOfEntry
```

Both queries return the same result, as shown in Example 4-142.

Example 4-142 Filtered result

```
<dateOfEntry xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  2005-09-12
</dateOfEntry>
```

XML namespace considerations

When searching XML data, you might encounter XML data that belongs to a specific non-default namespace. This is the case where you must use the fully qualified name when searching in specific elements or sub-trees in the XML document. A fully qualified name consists of the namespace prefix and element name.

Example 4-143 shows that the elements of the element feedback belongs to a namespace with prefix itso.

Example 4-143 An XML document with namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<itso:feedback xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:itso="http://www.itso.org">
  <itso:entry>
    <itso:dateOfEntry> 2005-08-16 </itso:dateOfEntry>
    <itso:rating>3</itso:rating>
    <itso:comment lan="en">There should be more loan
prodcuts.</itso:comment>
  </itso:entry>
</itso:feedback>
```

Example 4-144 shows a query that returns no result because the path in the query is not fully qualified.

Example 4-144 Example of a query returns no result

```
select appl_id from feedback where CONTAINS(comment, '
section("/feedback/entry/comment") "loan")=1
```

Example 4-145 shows a query that returns an application ID because the path in the query is fully qualified.

Example 4-145 Example of a query that returns result

```
select appl_id from feedback where CONTAINS(comment, '
section("/itso:feedback/itso:entry/itso:comment") "loan")=1
```

4.6.6 The NSE document model

Net Search Extender uses a document model to configure the scope of search within structured documents such as XML. The document model defines what information is indexed, how that information is indexed, and by what name you can refer to that information. When creating a text index, you can either use the default document model or supply a custom document model.

These are characteristics of the default document model:

- ▶ All parts of the document are indexed.
- ▶ Search can be refined using XPath syntax.
- ▶ Numeric values are not supported.

These are characteristics of a custom document model:

- ▶ You can define which parts of the XML document are indexed and which parts are excluded.
- ▶ You can give custom names to parts of the XML document (sub-trees, elements, or attributes).
- ▶ You can define an XML element or attribute to be a numeric value allowing for parametric searching.

An NSE document model is itself an XML document. Example 4-146 is a custom document model.

```
<XMLModel>
  <XMLFieldDefinition name="comment" locator="/feedback/entry/comment"
/>
  <XMLAttributeDefinition name="rating" type="NUMBER"
locator="/feedback/entry/rating"/>
</XMLModel>
```

XMLModel

XMLModel is the top-level element. Two XML child elements are allowed for XML Model: XMLFieldDefinition and XMLAttributeDefinition.

XMLFieldDefinition

This element defines the custom name for the specific part, element, or attribute of the XML document that is identified by the 'locator' attribute.

XMLAttributeDefinition

This element defines an NSE attribute based on an XML element or attribute. This attribute can be defined as being of type NUMBER, which allows for numeric operations and ranges to be used during searches.

Locator attribute

The locator attribute is an XPath expression that defines the part of the document that should be indexed. The following subset of XPath expressions are supported for the locator attribute:

- ▶ Child axis (/)
- ▶ Descendent-and-self axis (//)
- ▶ Attribute axis (@)
- ▶ Wildcards (*)
- ▶ Comment node (comment())
- ▶ Processing-instruction node (processing-instruction())
- ▶ Union of elements (A | B)

Example 4-147 is an example of possible locator attribute values

```
locator="/feedback/entry/@lan"
locator="/feedback/entry/*"
locator="//comment"
locator="/feedback/entry/rating"
```

Name attribute

The name attribute defines the name that can be used to refer to the part of the XML document identified by the locator attribute. There are three special values that can be used to automatically generate a name value:

- ▶ name="\$ (NAME)"

Represents the qualified name of the XML element or attribute that is identified by the locator attribute. This would include any namespace associated with the data.

- ▶ name="\$ (LOCALNAME)"

Represents the locale name (no namespace) of the XML element that is identified by the locator attribute.

- ▶ name="\$ (PATH)"

Represents the absolute path to the XML element or attribute identified by the locator attribute. This is equivalent to the value returned from the locator XPath expression.

Type attribute

The 'type' attribute can only be used with the XMLAttributeDefinition element. The only value allowed is "NUMBER". Using this attribute specifies that the underlying data of the XML document can be considered numeric and allows for parametric search of this data.

Default document model

Given the characteristics of the NSE document model format, the default document model can be defined as:

```
<XMLModel>
<XMLFieldDefinition name="$ (PATH)" locator="*" />
</XMLModel>
```

This document model single definition matches everything with its 'locator' attribute and uses a special value for the name attribute to index every match returned by the locator attribute. The end result is that everything in the document is indexed by its corresponding XPath value.

Using a custom document model

After you have created a custom document model, you can use it when creating a new index. Next, Example 4-148 shows creating a new index using a custom document model file model.xml as in Example 4-146 on page 168.

Example 4-148 Create index with a custom document model

```
db2text create index idx2 for text on feedback(comment) format xml
documentmodel XMLModel in C:\model.xml connect to databaseName;
```

You must update the index idx2 before you can use it. Example 4-149 shows the update command.

Example 4-149 Update index idx2.

```
DB2TEXT UPDATE INDEX idx2 FOR TEXT CONNECT TO databaseName
```

The document model parameter specifies the root element and the location of the file. The document model file is only used during the creation of the index, so any later changes to the file would have no affect on existing indexes.

Searching with a custom document model

When you have created an index with your custom document model and updated it, you can begin searching the document using your custom rules.

The part of the document where searching can occur is specified by the name attribute as defined in the document model. Any parts of the document not specified by the document model are not indexed and thus cannot be searched.

In Example 4-150, when using the document model described earlier, the following query returns a result. The comment in the function section is defined in Example 4-146 on page 168.

Example 4-150 query returns result

```
SELECT appl_id FROM feedback WHERE
CONTAINS(comment, ' section("comment") "bank"')=1
```

However, the next query will not return any results even though it references the same part of the document. Example 4-151 shows the query by using path. This is so, because the document model explicitly states only those names that are valid for searching.

Example 4-151 query returns no result.

```
SELECT appl_id FROM feedback WHERE
CONTAINS(comment, ' section("/feedback/entry/comment") "bank"')=1
```

Parametric search

If you define elements using the XMLAttributeDefinition in the document model to be numeric, parametric search is possible. Example 4-152 shows a query that searches by using values specified by the name attribute “rating” as defined in the document model. The query searches for the application IDs that have ratings between 0 and 6.

Example 4-152 Sample parametric search query

```
SELECT appl_id FROM feedback WHERE  
CONTAINS(comment, ' ATTRIBUTE "rating" BETWEEN 0 AND 6')=1
```

Managing XML data

In this chapter, we discuss how to manage XML data stored in XML columns. We introduce the pureXML index features and show you how to use the XML indexes to enhance the performance of XQuery or SQL/XML. After describing schema management, we explain how to move data, including XML documents, in and out of tables using the DB2 9 IMPORT and EXPORT utilities. We also cover how RUNSTATS works with the pureXML features. Finally, we present some security solutions corresponding to the pureXML features.

We cover the following topics:

- ▶ XML index
- ▶ Schema management
- ▶ IMPORT, EXPORT, and RUNSTATS
- ▶ Security

5.1 XML indexes

pureXML in DB2 9 provides intelligent and rich features for storing and working natively with XML documents. One of these is the indexing feature that can index over XML columns. With this indexing feature, retrieving XML node sets is much faster compared to retrieving nodes from XML documents stored in CLOB.

Similar to a relational index, an XML index over XML data indexes an entire column. However, there are some externally visible differences between relational indexes and XML indexes, as follows:

- ▶ Indexes are created on columns of type XML based on path expressions (xmlpattern): The XML pattern expression is similar to the path expression defined in the XQuery language, but it differs in that only a subset of the XQuery language is supported
- ▶ When creating an index, it is possible to specify what paths to index and what types you want to use in your queries.
- ▶ You can only index on a single XML column — composite indexes are not allowed at this time. Elements and attributes inside the document frequently used in predicates and cross-document joins can be indexed.
- ▶ If a node matches the xmlpattern but fails to cast to the specified index type, then no index entry is created for the node, without raising an error.
- ▶ A single document can contain zero, one, or multiple nodes that match the xmlpattern. Thus there can be zero, one, or multiple index entries for a single row in a table (significantly different from indexes on relational columns).

5.1.1 XML index types

DB2 9 pureXML introduces three XML indexes:

- ▶ XML regions index
- ▶ XML column path index
- ▶ Index on an XML column

XML regions index

An *XML regions index* stores the locations of each XML document that is stored in XML storage in DB2 9. If you create a table T1, which has an integer column and an XML column in it as shown in Figure 5-1, the integer column is stored in T1, but DB2 does not store the XML document in T1. Instead, the XML column, XMLDOC, contains an XML data descriptor that has the document ID and version ID for the XML document. The document is stored in the XML Data Area (XDA), which is separate from the base table.

The nodes and subtrees in an XML data page form regions in a document. The XML regions index provides a logical mapping of those regions so that the document data can be retrieved from the XML data pages. The document ID and version ID in the XML Data Descriptor are used to do an index look-up in the regions index. The regions index key entry has the record ID of the root node of the XML document in the XDA.

The XML regions index is automatically created by DB2 9 when the first XML column is created or added to a table. Even though the table has multiple XML columns, just one XML regions index is created. Accessing the XML documents stored in XML storage always goes through the XML regions index.

Figure 5-1 shows how the XML regions index works.

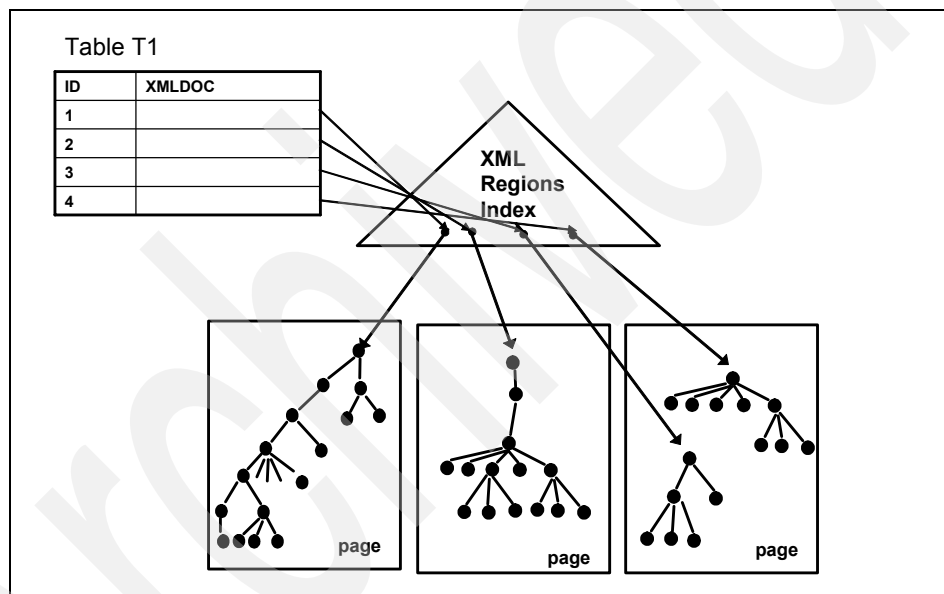


Figure 5-1 XML regions index

XML column path index

The *XML column path index* is system-generated for each XML column created or added to the table. For example, if a table with two XML columns is created, there is one XML regions index, but two XML column path indexes generated by DB2 9.

The XML column path index maps paths to path IDs for each XML column. It is a subset of the paths stored in the global catalog path table and is used to improve index access performance for queries. When an XML document is inserted, every unique path in the XML document is extracted and stored in the XML column path index with a unique path ID.

Index on an XML column (XML index)

An index on an XML column is an index created over an XML column (hereinafter called an *XML index*). This index allows users to enhance the performance of XQuery and SQL/XML. You can index every XML path in an XML column using XPath. Like other relational indexes, the XML index is created as a B-tree structure and stored in the same place as the relational indexes are stored. You can also define multiple XML indexes in one XML column.

Like indexes on relational data, using indexes on an XML column to improve the query performance may have some cost. The performance for INSERT, UPDATE, and DELETE can decrease as the number of indexes defined on XML column increases. Indexes also take space, so you should only create indexes that are really necessary.

5.1.2 Creating XML indexes

In this section, we show you how to create XML indexes by examples. We use one table, LOAN_APPLICATION, for our demonstration. The zip file, ch5sampledata.zip, which includes sample data and script for creating the table and importing data, can be downloaded from the IBM Redbooks Web site. The download instructions are in Appendix B, “Additional material” on page 373.

The LOAN_APPLICATION table stores the loan application XML document in APPL_DOC column. Example 5-1 shows a sample XML document stored in the LOAN_APPLICATION table.

```
<?xml version="1.0"?>
<Application>
  <Customer>
    <Name>
      <FirstName>Ichiro</FirstName>
      <LastName>Ohta</LastName>
    </Name>
    <DateOfBirth>2/11/1999</DateOfBirth>
    <SSN>111-33-3627</SSN>
    <Address country="JP">
      <Street>33 AKEBONO</Street>
      <City>Takatushi-shi</City>
      <State>Osaka</State>
      <Zip>33333</Zip>
    </Address>
    <Phone type="work">201-999-9646</Phone>
    <Phone type="home">039-999-0251</Phone>
    <Email>ichiro.ohta@awagat.com</Email>
    <Employer>
      <Company>My company3</Company>
      <Position>Developer</Position>
    </Employer>
    <FinancialData>
      <Income>76800.00</Income>
      <Debt>44500.00</Debt>
      <Expenses>40000.00</Expenses>
      <Assets>1400.00</Assets>
    </FinancialData>
  </Customer>
  <LoanType>0</LoanType>
  <Campain>1</Campain>
</Application>
```

Creating XML indexes with CREATE INDEX statement

The CREATE INDEX statement has been enhanced to support XML indexing. XML indexes are created on columns of type XML based on path expressions (xmlpattern). Figure 5-2 shows the CREATE INDEX statement structure for an XML index.

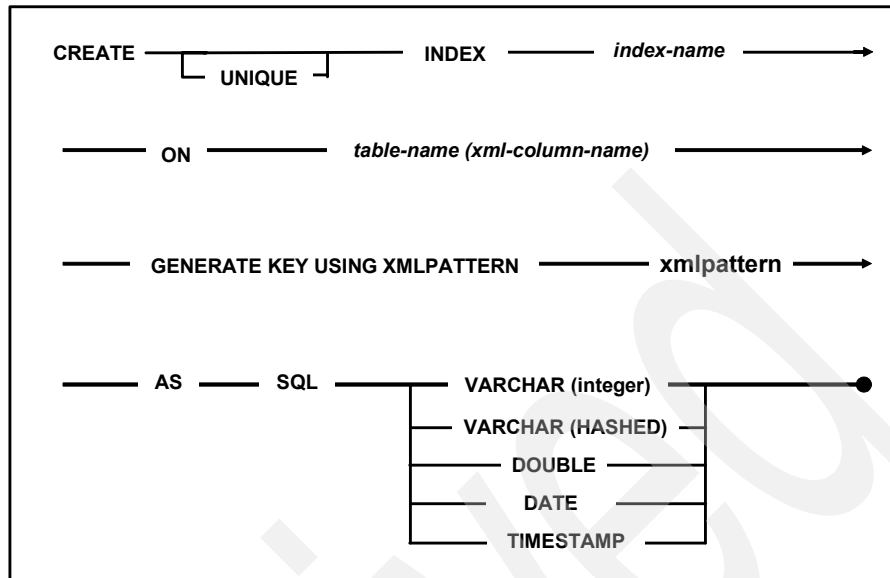


Figure 5-2 *CREATE INDEX* sentence structure for an XML index

When creating an XML index, the following fields are required:

- ▶ Index name: Specify the name of an XML index.
- ▶ Table and column names: Specify which XML column is indexed.
- ▶ XMLPATTERN: Specify the node you want to index.

XMLPATTERN is similar to XPath expression. The difference between XMLPATTERN and XPath is that XMLPATTERN cannot have any conditional expressions. For example, the following expression is valid for XPath but not valid for XMLPATTERN:

```
/Application/Customer/Address[@country="JP"]
```

- ▶ Data type: Specify the SQL data type for the XML index.

Here, we create an XML index for our example XML document. If you require a query for looking up zip codes in the XML column, the XQuery could be similar to the following piece of code:

```

XQUERY
for $Application in
db2-fn:xmlcolumn('LOAN_APPLICATION.APPL_DOC')/Application
return $Application/Customer/Address/Zip;

```

The XML index you might create for this specific query would look similar to the following code sample:

```
CREATE INDEX zipindex ON loan_application(appl_doc) GENERATE KEY USING
XMLPATTERN '/Application/Customer/Address/Zip' AS SQL VARCHAR;
```

Figure 5-3 shows a conceptual structure of this XML index. CREATE INDEX ZIPINDEX statement creates a B-tree index ZIPINDEX, which contains the paths for the Zip node and the value of the Zip nodes.

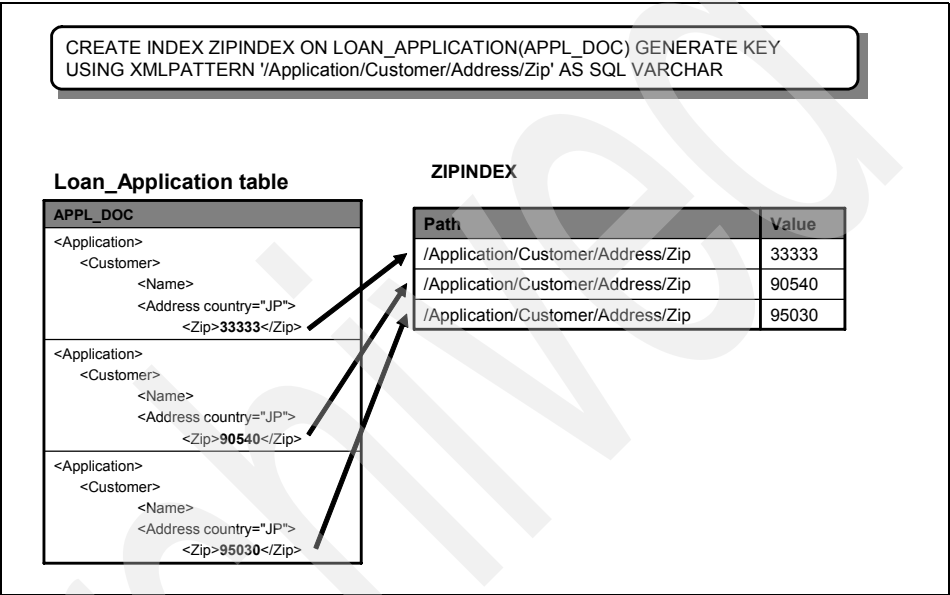


Figure 5-3 Create XML index statement and logical structure of XML index

Data type

When you create an XML index, you must specify the SQL data type for the node value that you want to index so that DB2 9 can convert XML node values specified in the xmlpattern clause to the SQL data type. The values then can be stored in a B-tree index. There are five SQL data types you can use: DOUBLE, VARCHAR(n), VARCHAR HASHED, DATE, and TIMESTAMP.

DOUBLE

The data type DOUBLE should be used to index numeric XML node values. Note that unbounded decimal types and 64-bit integers might lose precision when they are stored as DOUBLE. Following is an example of using the DOUBLE data type:

```
CREATE INDEX zipindexd ON loan_application(appl_doc) GENERATE KEY USING
XMLPATTERN '/Application/Customer/Address/Zip' AS SQL DOUBLE;
```

VARCHAR(n)

This data type is used to index varying-length string node values. The maximum length "n" specified in bytes is a constraint. The index is guaranteed to store complete string values. If you try to insert an XML document which would have an indexed string node value that is longer than the specified maximum length, the insertion will fail. If you try to create an XML index that would have an indexed string node value that is longer than the specified maximum length, the CREATE INDEX statement will fail.

```
CREATE INDEX zipindexv ON loan_application(appl_doc) GENERATE KEY USING
XMLPATTERN '/Application/Customer/Address/Zip' AS SQL VARCHAR(5);
```

Depending on page sizes, the maximum length that you can specify in "VARCAHR(n)" varies:

- ▶ 4K page: Maximum length is 817 bytes.
- ▶ 8K page: Maximum length is 1841 bytes.
- ▶ 16K page: Maximum length is 3889 bytes.
- ▶ 32K page: Maximum length is 7985 bytes.

In the case of indexing on the zip code node in the LOAN_APPLICATION table, both DOUBLE and VARCHAR data types can be used. Note that you can create two XML indexes on same xmlpatterns if they have different SQL data types.

VARCHAR HASHED

VARCHAR HASHED is used to handle indexing of character strings with arbitrary lengths. The VARCHAR HASHED data type can be used in the following cases:

- ▶ If the length of the character string values to be indexed is unknown.
- ▶ When you cannot use VARCHAR(n) because the character string to be indexed exceeds the maximum length, n can specified for the page in which the index is based.

In this case, the system generates an eight-byte hash code over the entire string and there is no limit on the length of the indexed string.

Note that range scans cannot be performed if you specify VARCHAR HASHED data type, because the index contains hash codes instead of the actual character data. Indexes using hashed character strings can be used only for equality lookups. Following is an example using VARCHAR HASHED:

```
CREATE INDEX zipindexvh ON loan_application(appl_doc) GENERATE KEY
USING XMLPATTERN '/Application/Customer/Address' AS SQL VARCHAR HASHED;
```

DATE

DATE data type values will be normalized to Coordinated Universal Time (UTC) or Zulu time before being stored in the index. Note that the XML schema data

type for DATE allows greater precision than the SQL data type. If an out-of-range value is encountered, an error is returned. Following is an example using DATE data type:

```
CREATE INDEX birthdate ON loan_application(appl_doc) GENERATE KEY USING
XMLPATTERN '/Application/Customer/DateOfBirsth AS SQL DATE;
```

TIMESTAMP

TIMESTAMP data type values will be normalized to UTC or Zulu time before being stored in the index. If XML documents have node values as shown in the following example; you can use TIMESTAMP data type to index the node.

```
<date>2006-08-23T07:21:00.000000Z</date>
```

Unique index

You can create a unique index over an XML column. Note that values that you specify for xmlpattern in your CREATE INDEX statement must be unique, not only in one XML document, but also unique in all the XML documents stored in the XML column. In our example, you can create a unique index for SSN because SSN is supposed to be unique. Following is the CREATE INDEX statement:

```
CREATE UNIQUE INDEX ssnindex ON loan_application(appl_doc) GENERATE KEY
USING XMLPATTERN '/Application/Customer/SSN' AS SQL VARCHAR(11);
```

The following statement will fail because zip code is not unique:

```
CREATE UNIQUE INDEX ZIPINDEXD ON LOAN_APPLICATION(APPL_DOC) GENERATE
KEY USING XMLPATTERN '/Application/Customer/Address/Zip' AS SQL DOUBLE;
```

5.1.3 How to look up information for XML indexes

In this section, we show how to look up information about the XML indexes that are stored in the DB2 9 catalog tables.

Logical and physical indexes

When you create an index on an XML column, two indexes are actually created, a *logical index* and a *physical index*. The *logical index* contains the XML pattern information specified in the CREATE INDEX statement. The *physical index* has DB2 generated key columns to support the logical index and contains the actual index values. The user works with an index on an XML column at the logical level for the CREATE INDEX and DROP INDEX statements. Processing of the underlying physical index by DB2 is transparent to the user.

The logical index has the index name specified in the CREATE INDEX statement and has the indextype XVIL. The physical index has a system generated name

and has the indextype XVIP. The logical index is always created and assigned an index ID first. The physical index is created immediately afterwards and is assigned the next consecutive index ID.

SYSCAT.INDEXES

As with relational indexes, index information for the XML indexes is stored in SYSCAT.INDEXES. Even though the XML column path index and the XML regions index are system created indexes, they are visible in SYSCAT.INDEXES.

Four new index types have been added to SYSCAT.INDEXES:

- ▶ XVIL: Index on an XML column (logical)
- ▶ XVIP: Index on an XML column (physical)
- ▶ XPTH: XML paths index
- ▶ XRGN: XML regions index

Here, we create an XML index, APPLNAME, and see what information DB2 stored in SYSCAT.INDEXES.

```
CREATE INDEX applname ON loan_application(appl_doc) GENERATE KEY USING
XMLPATTERN '/Application/Customer/Name' AS SQL VARCHAR(32);
```

After creating that index, issue this SELECT statement to retrieve XML index information.

```
SELECT indname, TABNAME, INDEXTYPE FROM SYSCAT.INDEXES
WHERE TABNAME='LOAN_APPLICATION';
```

The result would be similar to Example 5-2.

Example 5-2 SELECT statement output

INDNAME	TABNAME	INDEXTYPE
-----	-----	-----
SQL060821123323580	LOAN_APPLI	XRGN
SQL060821123323700	LOAN_APPLI	XPTH
APPLNAME	LOAN_APPLI	XVIL
SQL060821154539000	LOAN_APPLI	XVIP
4 record(s) selected.		

The following index types are listed in the INDEXTYPE column:

- ▶ XRGN: XML regions index
- ▶ XPTH: XML column path index
- ▶ XVIL: Logical index for APPLNAME
- ▶ XVIP: Physical index for APPLNAME

The regions index and XML column path index are created by DB2 9 automatically when the LOAN_APPLICATION table is created. The logical and physical indexes are created when the CREATE INDEX statement is executed successfully. Actual index values are stored in the physical index.

SYSCAT.INDEXXMLPATTERNS

In SYSCAT.INDEXXMLPATTERNS, you can find the name, length, and xmlpattern of the XML index that you have created. For the following query:

```
SELECT indname, pindname, datatype, length, pattern FROM
syscat.indexxmlpatterns
```

The result might be similar to this example:

INDNAME	PINDNAME	DATATYPE	LENGTH	PATTERN
APPLNAME	SQL060821154539000	VARCHAR	32	/Application/Customer/Name

db2dart

The DB2 database analysis and reporting tool **db2dart** can be used to examine the architectural correctness of databases and objects within it. You can use this tool to see what values are stored in XML indexes.

To check the XML index using **db2dart**, you have to provide the index object ID and table space ID where the XML index is stored. Example 5-3 shows the query to find the table space ID.

Example 5-3 Get table space ID

```
SELECT tabname, tbspaceid FROM syscat.tables
WHERE tabname = 'LOAN_APPLICATION'
```

TABNAME	TBSPACEID
LOAN_APPLICATION	2

The index object ID can be obtained from the syscat.indexes catalog table INDEX_OBJECTID column using the query shown in Example 5-4.

Example 5-4 Get index object ID

```
SELECT indname, index_objectid FROM syscat.indexes
WHERE indname in ('APPLNAMEW')
```

INDNAME	INDEX_OBJECTID
APPLNAME	4

Now that you know the table space ID and index object ID, you are ready to issue **db2dart**. Specify the database name XMLRB, table space ID 2 for /tsi option, and index object ID 4 for the /oi option as follows:

```
db2dart XMLRB /di /tsi 2 /oi 4 /ps 0 /np 10000 /v y
```

The options used are:

- ▶ /di: dump formatted index data
- ▶ /tsi: table space id
- ▶ /oi: object id
- ▶ /ps: page number to start dumping
- ▶ /np: number of pages
- ▶ /v: verbose option

Example 5-5 shows a clip of the output generated by **db2dart**. You can see the structure of the XML index APPLNAME and the value stored in APPLNAME.

Example 5-5 Output from db2dart

Key 1:

```
Offset Location = 3546 (xDDA)
Record Length   = 39 (x27)
Key Part 1:
  Long Integer
  Value = 105
Key Part 2:
  Variable Length Character String
  Actual Length = 10
  49636869 726F4F68 7461                Ichiro0hta
Key Part 3:
  Big Integer
  Value = 22799473113613056
Key Part 4:
  Variable Length Binary String
  Actual Length = 2
  2222                                ""
Key Part 5:
  Fixed Length Character String
  00                                  .
Key Part 6:
  Fixed Length Character String
  31                                  1
Table RID: x(0000 0083 000B) r(00000083;000B) d(131;11)
ridFlags=x2 Punc
```

db2cat

The system catalog analysis command **db2cat** can analyze the contents of packed descriptors. Given a database name, schema name, and table name, this command will query the system catalogs for table information and format the results. The same index statistics are collected (such as nlevels and nleafs) for XML indexes as relational indexes when using RUNSTATS. You can use db2cat to check the statistical collection for an XML column. db2cat has these options:

-d : database name
-s : schema name
-n : table name
-o : output file

Example 5-6 shows the **db2cat** command and the output it generates with information about the XML column statistics of the APPL_LOAN table section.

Example 5-6 A part of the output file from db2cat

```
db2cat -d XMLRB -s db2admin -n loan_application -o db2catoutput.txt
+++++
XML column statistics
+++++
Column ID              = 1
No. NULL XML docs      = 0
No. non-NULL XML docs  = 102
-----
Catch All Pathid Bucket
-----
Distinct Pathid count = 44
Sum Node Counts        = 4794
Sum Doc Counts          = 4488
-----
Top-k Pathid node counts
-----
Max no. of path counts = 44
Cur no. of path counts = 44
Cnt( /root()/Application/Customer/Phone ) = 204
Cnt( /root()/Application/Customer/Phone/type ) = 204
.....
-----
PathID                  = /root()/Application/Customer/Address/Street/text()
Distinct Value Cnt      = 3
2nd Highest Key         = 19:46 East Main Street
2nd Lowest Key          = 14:1-1-1 AINOKAWA
Sum Node Cnt            = 102
Sum Doc Cnt             = 102
```

5.1.4 Access plan

An *access plan* is a plan specifying an order of operations for accessing DB2 data. The DB2 9 optimizer creates the best access plan whenever SQL or XQuery statements are compiled. You can see statistics for tables, indexes, and columns in the access plan. DBAs can check the access plan for a specific XQuery or SQL/XML query to discern if XML indexes that are expected to be used are actually used.

New operators: XSCAN, XISCAN, XANDOR

In DB2 9, the new operators, XSCAN, XISCAN, and XANDOR, are added to the access plan for accessing XML columns.

XSCAN

DB2 uses the XSCAN operator to traverse XML document trees and if required, to evaluate predicates and extract document fragments and values. XSCAN can appear in an execution plan after a base table scan to process each of the documents retrieved from a table.

For example, if you want to search XML documents that have Ichiro in the FirstName element, XQuery can be similar to this:

```
db2-fn:xmlcolumn('LOAN_APPLICATION.APPL_DOC')/Application/Customer/Name  
[FirstName="Ichiro"]
```

If there is no index on the FirstName element, DB2 9 has to read all of the XML documents stored in the APPL_DOC column to check each XML document that satisfies those XPath conditions.

Figure 5-4 shows how DB2 9 gets results for this XQuery. DB2 first accesses the LOAN_APPLICATION table to get the first XML column and then accesses the XML regions index to get the address of the first node of the first XML document.

DB2 then traverses nodes from the first node to see if the document satisfies the condition /Application/Customer/Name[FirstName="Ichiro"]. This continues until DB2 finds the XML document which has a customer with last name Ichiro. This might be the last document in the database, as in our example.

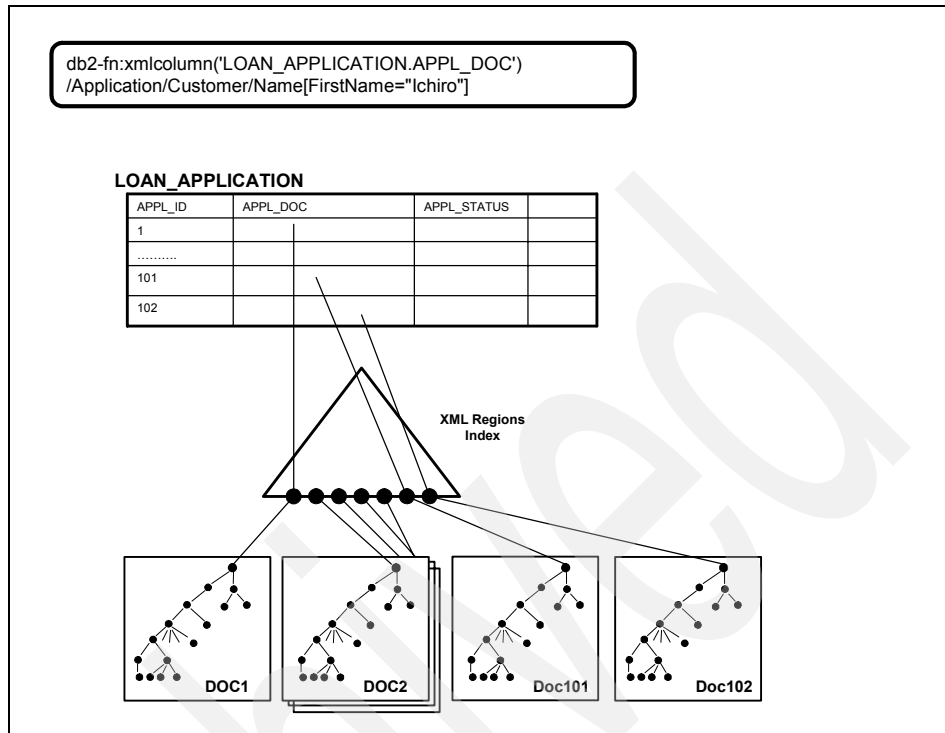


Figure 5-4 Logical model for accessing XML column without an index

XISCAN

XISCAN scans an XML index. This operator is likely to be used if an adequate XML index for the XQuery has already been created.

Assume that an XML index APPLFIRSTNAME for FirstName element has been created using the following command:

```
CREATE INDEX applfirstname ON loan_application(appl_doc)
GENERATE KEY USING XMLPATTERN '/Application/Customer/Name/FirstName' AS
SQL VARCHAR(32)
```

Now that we have an XML index for the FirstName element, DB2 no longer has to read all XML documents to find the record. Instead, DB2 only accesses the XML documents that are required.

Figure 5-5 shows conceptually how DB2 accesses XML documents via XML indexes. When the same query is issued, since the FirstName element has already been indexed, DB2 checks the values in the index APPLFIRSTNAME. Once DB2 finds it, DB2 uses the RID to access the base table descriptor.

Through the XML region index, DB2 accesses the XML document to construct the result sequence.

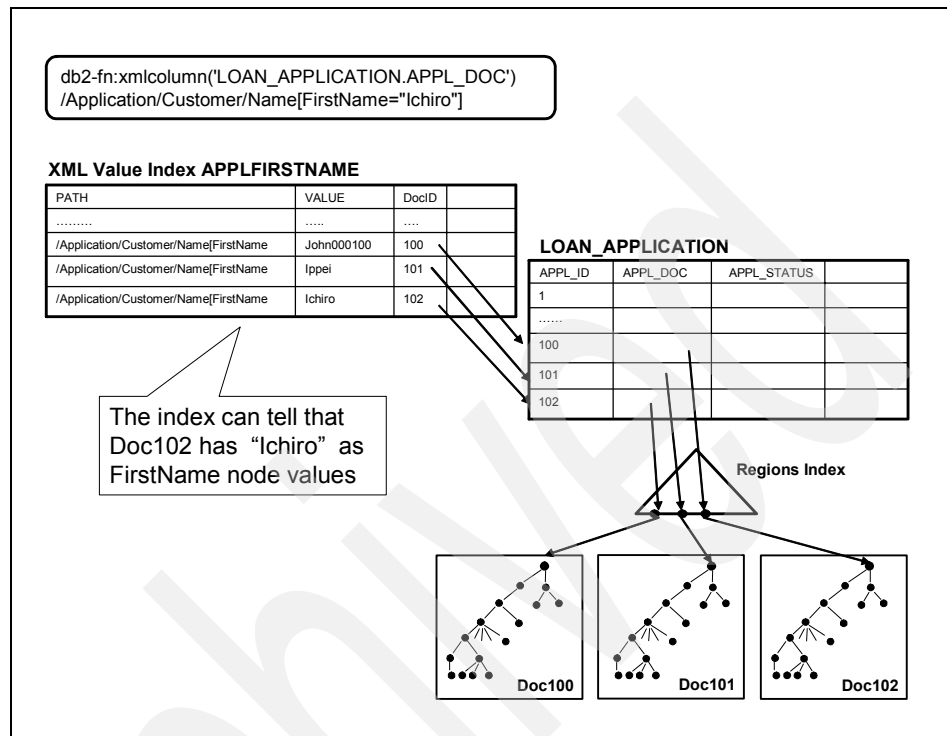


Figure 5-5 Logical model for accessing XML column with an index

XANDOR

The XANDOR operator merges multiple results from the XISCAN operators. In DB2 9, XANDOR supports only ANDing.

You define such an XML index for the Zip element as described here:

```
CREATE INDEX applzip ON loan_application(appl_doc) GENERATE KEY USING
XMLPATTERN '/Application/Customer/Address/Zip' AS SQL VARCHAR(5)
```

Now you have two XML indexes on the LOAN_APPLICATION — one is APPLFIRSTNAME for the FirstName element, another is APPLZIP for the Zip element. DB2 probably uses the XANDOR operator for a query similar to this:

```
XQUERY for $i in
db2-fn:xmlcolumn('LOAN_APPLICATION.APPL_DOC')/Application/Customer[Name
/FirstName="Ichiro" and Address/Zip="33333"]/Name return $i
```

The index APPLFIRSTNAME is used for resolving the condition /Application/Customer/Name[FirstName="Ichiro"] and the index APPLSIZP is used for resolving the condition /Application/Customer/Address[Zip="33333"]. XANDOR is used to merge the input from both XML indexes.

How to get an access plan

We have discussed three operators for accessing XML columns in an access plan. Here we show you how to acquire an access plan for XQuery and SQL/XML statements. There are several ways to get an access plan. You can use the **db2exfmt** command, the **db2expln** command, or Visual Explain. Visual Explain is a graphical tool. **db2exfmt** and **db2expln** can be used in the environment without graphical representation. In this section, we introduce **db2exfmt** and Visual Explain.

db2exfmt

To get the access plan, you have to create Explain tables. DB2 provides a script EXPLAIN.DDL to create Explain tables. This script is in the directory %SQLLIB%\misc. To run the script, issue the following command from DB2 CLP:

```
db2 -tvf EXPLAIN.DDL
```

If you want to follow our example using the data we provide, drop the XML indexes created in the previous section and refresh the table statistics using the following command:

```
DROP INDEX APPLFIRSTNAME;  
DROP INDEX APPLZIP;  
RUNSTATS ON TABLE DB2ADMIN.LOAN_APPLICATION;
```

The case where XSCAN is used

In this example, we show how XSCAN is used in an access plan. Follow these steps to get the access plan for the query we would like to examine:

1. Set explain mode to YES using the following command:

```
db2 set current explain mode yes;
```
2. Issue the XQuery with which you want to get the access plan:

```
XQUERY for $i in  
db2-fn:xmlcolumn('LOAN_APPLICATION.APPL_DOC')/Application/Customer/N  
ame[FirstName="Ichiro"] return $i;
```
3. Set explain mode back to NO:

```
db2 set current explain mode no;
```
4. Issue the db2exfmt command to format the output to a file:

```
db2exfmt -d XMLRB -o plan1.txt -1;
```

Example 5-7 shows the formatted output. After the XSCAN operator is chosen, all XML documents in XML column are read for checking if an XPath specified in XQuery or SQL/XML is matched or not.

Example 5-7 Access plan where XSCAN operator is used

Access Plan:

```

-----
Total Cost: 805.299
Query Degree:1

      Rows
      RETURN
      ( 1)
      Cost
      I/O
      |
      1
      NLJOIN
      ( 2)
      805.299
      106
      /---\
102      0.00980392
TBSCAN      XSCAN
( 3)      ( 4)
30.8137      7.59299
4              1
|
102
TABLE: DB2ADMIN
LOAN_APPLICATION

```

The case where XISCAN is used

Here we show an example where XISCAN is used. We create an index on the FirstName element using the following command and check the access plan for the same query:

```

CREATE INDEX applfirstname ON loan_application(appl_doc) GENERATE KEY
USING XMLPATTERN '/Application/Customer/Name/FirstName' AS SQL
VARCHAR(32)

```

Repeat the steps in “The case where XSCAN is used” on page 189. Example 5-8 shows the formatted output. From the access plan, we see that XISCAN is used; the total cost of this access plan is much lower than the one without an index.

Example 5-8 Access plan where XISCAN operator is used

Access Plan:

```

-----
Total Cost: 15.3607
Query Degree:1

      Rows
      RETURN
      ( 1)
      Cost
      I/O
      |
      1
      NLJOIN
      ( 2)
      15.3607
      2
      /-+-\
      1      1
      FETCH  XSCAN
      ( 3)   ( 7)
      7.76775 7.59299
      1      1
      /-----\
      1      102
      RIDSCN  TABLE: DB2ADMIN
      ( 4)   LOAN_APPLICATION
      0.175903
      0
      |
      1
      SORT
      ( 5)
      0.173137
      0
      |
      1
      XISCAN
      ( 6)
      0.166633
      0
      |
      102
      XMLIN: DB2ADMIN
      APPLFIRSTNAME
  
```

The case where XANDOR is used

In order to see if XANDOR is used, we can create another XML index on the APPL_DOC column Zip element using the following command:

```
CREATE INDEX applzip ON loan_application(appl_doc) GENERATE KEY USING
XMLPATTERN '/Application/Customer/Address/Zip' AS SQL VARCHAR(5)
```

Issue the following commands as shown in Example 5-9.

Example 5-9 Using the XANDOR operator

```
db2 set current explain mode yes;

db2 XQUERY for $i in db2-fn:xmlcolumn('LOAN_APPLICATION.APPL_DOC')
/Applycation/Customer[Name/FirstName="Ichiro" and
Address/Zip="33333"]/Name
return $i;

db2 set current explain mode no;

db2exfmt -d xmlrb -o plan3.txt -1;
```

Example 5-10 shows the access plan of this query. The access plan shows that two defined indexes APPLFIRSTNAME and APPLZIP are used. The XANDOR operator is used to merge multiple XISCAN output to produce the result set.

Example 5-10 Access plan where XANDOR operator is used

Access Plan:


Total Cost: 8.0112
Query Degree:1

```

      Rows
      RETURN
      (   1)
      Cost
      I/O
      |
0.00970874
      NLJOIN
      (   2)
      8.0112
      1.0098
      /--+--\
0.00980392  0.990291
```

FETCH	XSCAN
(3)	(9)
0.418209	7.59299
0.00980392	1
/-----+-----\	
0.00980392	102
RIDSCN	TABLE: DB2ADMIN
(4)	LOAN_APPLICATION
0.341843	
0	
0.00980392	
SORT	
(5)	
0.339077	
0	
0.00980392	
XANDOR	
(6)	
0.333267	
0	
/-----+-----\	
1	1
XISCAN	XISCAN
(7)	(8)
0.166633	0.166633
0	0
102	102
XMLIN: DB2ADMIN	XMLIN: DB2ADMIN
APPLFIRSTNAME	APPLZIP

Visual Explain

Visual Explain allows you to view the access plan for the explained SQL or XQuery statements as a graph. Using Visual Explain is probably the easiest way to get an access plan. If the Explain tables do not exist, Visual Explain automatically creates those tables for you. Visual Explain can be invoked from the Command Editor by entering the query for which you want to see an access plan and clicking the access plan icon . See Figure 5-6.

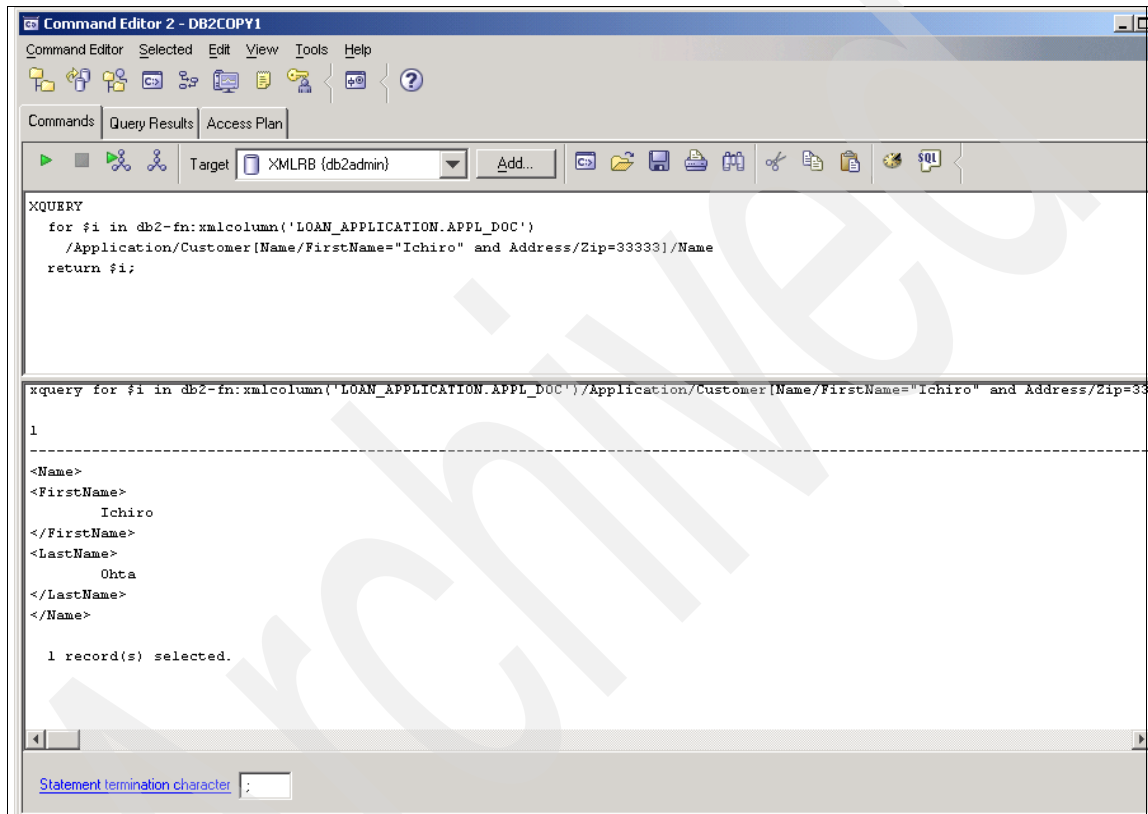


Figure 5-6 Command Editor

Figure 5-7 shows an access plan graph on Visual Explain. If you click any box on the pane, you can see further information.

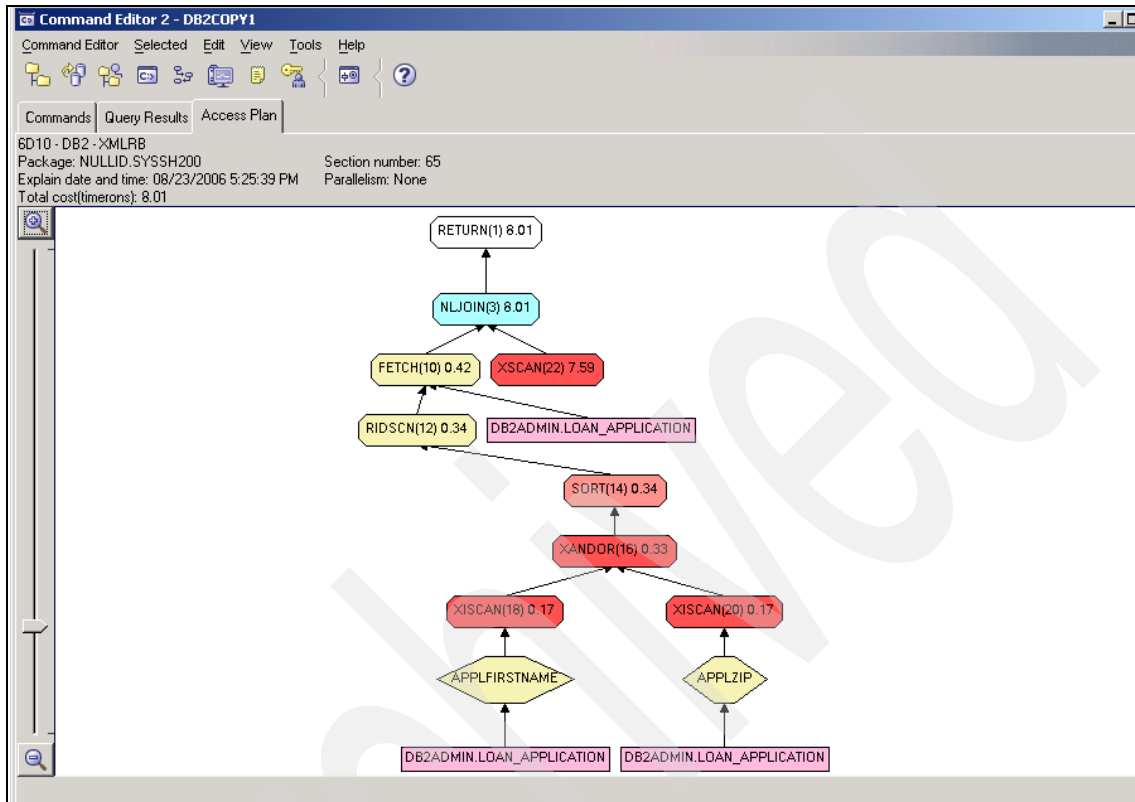


Figure 5-7 Access plan graph from Visual Explain

5.1.5 Best practices

This section describes some best practices we have used or learned along the way.

When to use an XML index

In this section, we provide a few guidelines for you to consider when creating XML indexes.

CASE 1: Predicate

An XML index can be used if the XML index contains the query predicate, for example, it is equally or less restrictive than the predicate.

Consider the following index and query on the LOAN_APPLICATION table:

```
CREATE INDEX applname ON loan_application(appl_doc) GENERATE KEY USING
XMLPATTERN '/Application/Customer/Name' AS SQL VARCHAR(32)
```

```
XQUERY for$i in db2-fn:xmlcolumn('LOAN_APPLICATION.APPL_DOC')
/Applycation/Customer/Name[FirstName="Ichiro"]
return $i
```

In this example, the XML index is on the Name element whose value is a combination of the node values from FirstName element and LastName element, for example "IchiroOhta". The XQuery will not be able to utilize the index APPLNAME because the query predicate /Applycation/Customer/Name[FirstName="Ichiro"] does not match the string node values in the XML index.

To use an XML index, the XML index should include FirstName element. Defining the XML indexes in either of the following ways allows DB2 to use the index:

```
CREATE INDEX applname ON loan_application(appl_doc) GENERATE KEY USING
XMLPATTERN '/Application/Customer/Name/FirstName' AS SQL VARCHAR(32)
```

```
CREATE INDEX applname ON loan_application(appl_doc) GENERATE KEY USING
XMLPATTERN '/Application/Customer/Name/*' AS SQL VARCHAR(32)
```

CASE 2: Data type

The data type should be matched between a query predicate and an XML index.

If the LOAN_APPLICATION table has one XML index, APPLZIP, which defines an XML index for the Zip element as a VARCHAR type, as follows:

```
CREATE INDEX applzip ON loan_application(appl_doc) GENERATE KEY USING
XMLPATTERN '/Application/Customer/Address/Zip' AS SQL VARCHAR(5)
```

The following query will be able to use the index APPLZIP because the query predicate specifies a text string and the type matches.

```
XQUERY $i in db2-fn:xmlcolumn('LOAN_APPLICATION.APPL_DOC')
/Applycation/Customer[Address/Zip="33333"]/Name
return $i
```

The following query has the value in the condition specified as a numeric value. This query will not use the XML index APPLZIP.

```
XQUERY $i in db2-fn:xmlcolumn('LOAN_APPLICATION.APPL_DOC')
/Applycation/Customer[Address/Zip=33333]/Name
return $i
```

CASE 3: Multiple XMLEXISTS in one query

You can expect better performance if you use a single XMLEXISTS function, rather than multiple XMLEXISTS functions in an SQL statement.

The following two SELECT statements in Example 5-11 return the same result. However, the first query separates two conditions into two XPath expressions with the EMLEXISTS function. The second query combines two conditions into one XPath expression. Each EMLEXISTS function is interpreted as an XQuery. Therefore, the cost of the first query will be higher than the second one.

Example 5-11 SELECT statements

```
SELECT l.appl_id FROM loan_application l
WHERE
XMLEXISTS('$i/Application/Customer/Name[FirstName = "Ippei"]'
PASSING l.appl_doc AS "i") AND
XMLEXISTS('$i/Application/Customer/Address[Zip = "22222"]'
passing l.appl_doc AS "i");

SELECT l.appl_id FROM loan_application l
WHERE
xmlexists('$i/Application/Customer[Name/FirstName="Ippei"]/Address[Zip=
"22222"]' PASSING l.appl_doc AS "i");
```

Note that the two XQueries in Example 5-12 cost almost the same.

Example 5-12 Two XQueries

```
XQUERY
for $i in db2-fn:xmlcolumn('LOAN_APPLICATION.APPL_DOC')/Application
where $i/Customer/Name/FirstName = "Ippei"
and $i/Customer/Address/Zip = "22222"
return $i/Customer/Name;

XQUERY
for $i in
db2-fn:xmlcolumn('LOAN_APPLICATION.APPL_DOC')/Application/Customer[Name
/FirstName="Ippei" and Address/Zip="22222"]/Name
return $i
```

CASE 4: Wild cards on an XML index

Be careful when you use wild cards (*) in an xmlpattern. Figure 5-8 shows an XML index, APPLALL, which indexes every single node of each XML document, and should be avoided.

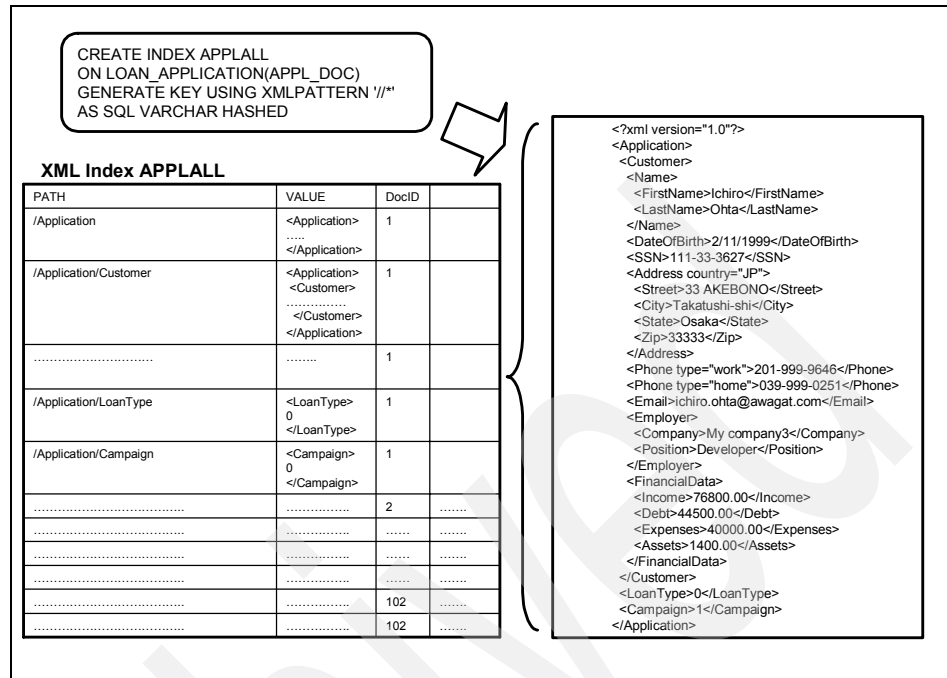


Figure 5-8 APPLALL indexes all XML nodes

5.2 Schema management

XML schemas are used to define the structures and data type constraints of the elements of XML documents. XML schemas are also used to validate XML documents. If an XML document satisfied the structures and data type constraints of an XML schema, the XML document is valid to the XML schema. There are other languages that can be used to define XML document, for instance, DTD (Document Type Definition). Comparing DTD with XML schema, XML schemas are richer and more powerful. XML schemas are written in XML and are extensible to future additions. It also supports data types and namespaces.

DB2 9 supports DTDs and external entries for entity resolution, but not for validation. DB2 9 supports only XML schemas to validate XML documents. All DTDs can be converted to XML schemas without any information loss. You can convert your existing DTDs to XML schemas. All XML schemas must be successfully registered in the XML schema repository in DB2 9 before you can use them. The registered XML schema should be managed in the same way as other database objects.

5.2.1 XML Schema Repository

DB2 9 keeps registered XML schemas in the XML schema repository (XSR). The registered XML schemas are used for validation. XML documents stored in XML columns usually have a reference or Uniform Resource Identifier (URI) that contains the information of their XML schemas, DTDs, or other external entities. The URI is required to validate XML documents. DB2 manages dependencies on such externally referenced XML artifacts with the XSR without requiring changes to the URI location reference. The XSR also removes the additional overhead required to locate external documents, along with the possible performance impact. An XML schema repository is located in the database catalog and comprises catalog tables, catalog views, and some system defined stored procedures to enter data into these catalog tables.

An XML schema can contain one or more XML schema documents. For instance, consider an XML schema document A.xsd that imports and includes other schema documents, B.xsd, C.xsd, and D.xsd. Here, A.xsd, B.xsd, C.xsd, and D.xsd are a collection of an XML schema. A.xsd is at the top of the hierarchy of the schema documents. A.xsd is the primary schema document, since it imports/includes all of the other schema documents, B.xsd, C.xsd, and D.xsd.

All XML schemas, DTDs and external entities must be registered before you can use them. A DB2 XSR object representing the XML schema is created when the first schema document is registered.

5.2.2 XML schema registration/dropping

We use examples to demonstrate how to register XML schemas and to manipulate an XSR object.

pets.xsd in Example 5-13 is a primary schema document. It defines an XML element "pets". The element "pets" has a sequence of two elements. The first element "CAT" has data type "ca:CAT" which is defined in cat.xsd. The second element "DOG" has data type "do:DOG" which is defined in dog.xsd.

pets.xsd imports both dog.xsd and cat.xsd, which are shown in Example 5-14 and Example 5-15 respectively.

Example 5-13 XML schema pets.xsd

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.itso.org/pets"
xmlns:pe="http://www.itso.org/pets" xmlns:ca="http://www.itso.org/cat"
xmlns:do="http://www.itso.org/dog">
```

```

<xs:import namespace="http://www.itso.org/cat" schemaLocation="cat.xsd"
/>
<xs:import namespace="http://www.itso.org/dog" schemaLocation="dog.xsd"
/>
  <xs:element name="PETS">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="DOG" type="do:DOG"/>
        <xs:element name="CAT" type="ca:CAT"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>

```

Example 5-14 XML schema cat.xsd

```

<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.itso.org/cat">
  <xsd:complexType name="CAT">
    <xsd:sequence>
      <xsd:element name="NAME" type="xsd:string" />
      <xsd:element name="AGE" type="xsd:integer" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

```

Example 5-15 XML schema dog.xsd

```

<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.itso.org/dog">
  <xsd:complexType name="DOG">
    <xsd:sequence>
      <xsd:element name="NAME" type="xsd:string" />
      <xsd:element name="AGE" type="xsd:integer" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

```

pets.xsd by itself is incomplete to validate a document containing elements "dogs" and "cats". In order to validate such a document, you also have to register dogs.xsd and cats.xsd. Use the following steps to register a schema:

1. Register the primary schema. The information required to register a primary schema document is:
 - The fully-qualified file name for the primary XML schema document:
The fully-qualified name means the file name plus the path to where the XML schema document is located. The fully-qualified file name is used in the REGISTER XMLSCHEMA command.
 - The SQL identifier:
You can choose any valid SQL two-part name to identify the XML schema. You should choose a meaningful name. The SQL identifier will be used for explicitly validating XML instance documents. You also require the SQL identifier when you drop the registered XML schema document.

Some additional information such as the schema location of the primary schema document can also be used in registration. Just as its name suggests, a schema location indicates where a schema document is located. The schema location can be a URL, FTP address, or a fully-qualified file name on the local machine. The schema location is stored in the catalog table after registration. The information can be used for implicitly validating an XML instance document.

Example 5-16 shows the command to register the primary schema document pets.xsd.

Example 5-16 Registering a primary schema document

```
register xmlschema http://sample from c:\pets.xsd as sample.pets
```

In this example, c:\pets.xsd is the full-qualified name. pets.xsd is located in root path of C drive in the local directory. The schema location using is the URL http://sample. SQL identifier is sample.pets.

2. Add schema documents.

You must add all the XML schema documents that the primary XML schema document directly or indirectly imports/includes if the primary XML has any imports/includes. You can skip this step if the primary XML has no imports/includes. The schema location must match the schemaLocation attribute from the import/include declaration in the importing/including schema document. Example 5-17 shows the command to add XML schema documents cat.xsd and dog.xsd.

Example 5-17 Adding schema documents

```
add xmlschema document to sample.pets add cat.xsd from c:\cat.xsd
add xmlschema document to sample.pets add dog.xsd from c:\dog.xsd
```

In this example, sample.pets is the SQL identifier of the registered primary XML schema. c:\cat.xsd and c:\dog.xsd are the schema locations that match the import tags for cat.xsd and dog.xsd in the primary XML schema document pets.xsd. C:\cat.xsd and C:\dog.xsd are the full-qualified file names on the local machine. No specific order is required when you add an XML schema document. In the example, we could have added dog.xsd first, then cat.xsd.

3. Complete the registration:

After you register the primary schema document and add all the involved schema documents, you can complete the registration process. This step checks the schemaLocation values in import and include tags in XML schema documents and the schema locations provided by **add xmlschema document** commands. If there is any mismatch, the completing registration would fail with an error. This step also checks if the XML schema documents are well-formed. If one or more schema documents are not well-formed, the completing registration would also fail with an error. Example 5-18 shows the command to complete the registration.

Example 5-18 Completing schema with success

```
complete xmlschema sample.pets
```

In the example above, sample.pets is the SQL identifier we used to register the primary XML schema document pets.xsd.

Example 5-19 is an example of completing registration that failed with error.

Example 5-19 Completing schema fails with error

```
complete xmlschema sample.pets
SQL20329N The completion check for the XML schema failed because one
or more XML schema documents is missing. One missing XML schema
document is identified by "NAMESPACE" as "http://www.itso.org/cat".
SQLSTATE=428GI
```

In this example, we did not add cat.xsd. Registration failed with error code SQL20329N.

After an XML schema is registered in XSR, it is a XSR object. You can remove a XSR object by dropping it. The schema repository does not have a notion for each XML schema document. When you drop an XML schema, all XML schema documents that belong to the XML schema are dropped.

If one or more of the XML schema documents have to be changed, you must drop the XML schema and recreate the XSR object with the new XML schema documents. You can drop an XML schema whether it is completed or not. Example 5-20 is an example of dropping an XML schema.

Example 5-20 Dropping a schema

```
drop xsrobject sample.pets
```

In this example, sample.pets is the SQL identifier that we used to previously to register the primary XML schema document.

5.2.3 Querying XSR

You can query the system catalog view SYSCAT.XSROBJECTS for the XSR object information. Example 5-21 shows the table description of the view SYSCAT.XSROBJECTS.

Example 5-21 Table description of the view SYSCAT.XSROBJECTS

Column name	Type schema	Type name	Length	Scale	Nulls
<hr/>					
OBJECTID	SYSIBM	BIGINT	8	0	No
OBJECTSCHEMA	SYSIBM	VARCHAR	128	0	No
OBJECTNAME	SYSIBM	VARCHAR	128	0	No
TARGETNAMESPACE	SYSIBM	VARCHAR	1001	0	Yes
SCHEMALOCATION	SYSIBM	VARCHAR	1001	0	Yes
OBJECTINFO	SYSIBM	XML	0	0	Yes
OBJECTTYPE	SYSIBM	CHARACTER	1	0	No
OWNER	SYSIBM	VARCHAR	128	0	No
CREATE_TIME	SYSIBM	TIMESTAMP	10	0	No
ALTER_TIME	SYSIBM	TIMESTAMP	10	0	No
STATUS	SYSIBM	CHARACTER	1	0	No
DECOMPOSITION	SYSIBM	CHARACTER	1	0	No
REMARKS	SYSIBM	VARCHAR	254	0	Yes

Suppose we require information about the target namespace, schema location, object schema, and object name. Example 5-22 shows the query and its output. Each row in the view SYSCAT.XSROBJECTS represents an XML schema.

Example 5-22 query view SYSCAT.XSROBJECTS

```
select SCHEMALOCATION, TARGETNAMESPACE, OBJECTSCHEMA, OBJECTNAME from  
SYSCAT.XSROBJECTS
```

SCHEMALOCATION	TARGETNAMESPACE	OBJECTSCHEMA	OBJECTNAME
http://sample	http://www.itso.org/sample	SAMPLE	ORDER
http://sample	http://www.itso.org/pets	SAMPLE	PETS
http://person	http://person	JOHN	PERSON

3 record(s) selected.

If you require information about XML schema documents, you can query the system catalog SYSCAT.XSROBJECTCOMPONENTS. Example 5-23 shows the table description of the view SYSCAT.XSROBJECTCOMPONENTS.

Example 5-23 Query view SYSCAT.XSROBJECTCOMPONENTS

Column name	Type schema	Type name	Length	Scale	Nulls
OBJECTID	SYSIBM	BIGINT	8	0	No
OBJECTSCHEMA	SYSIBM	VARCHAR	128	0	No
OBJECTNAME	SYSIBM	VARCHAR	128	0	No
COMPONENTID	SYSIBM	BIGINT	8	0	No
TARGETNAMESPACE	SYSIBM	VARCHAR	1001	0	Yes
SCHEMALOCATION	SYSIBM	VARCHAR	1001	0	Yes
COMPONENT	SYSIBM	BLOB	31457280	0	No
CREATE_TIME	SYSIBM	TIMESTAMP	10	0	No
STATUS	SYSIBM	CHARACTER	1	0	No

Example 5-24 shows the query of SYSCAT.XSROBJECTCOMPONENTS and its output. Unlike SYSCAT.XSROBJECTS, each row in the view SYSCAT.XSROBJECTCOMPONENTS represents an XML schema document.

Example 5-24 Query view SYSCAT.XSROBJECTCOMPONENTS

```
select TARGETNAMESPACE, SCHEMALOCATION, OBJECTSCHEMA, OBJECTNAME FROM
SYSCAT.XSROBJECTCOMPONENTS
```

TARGETNAMESPACE	SCHEMALOCATION	OBJECTSCHEMA	OBJECTNAME
http://www.itso.org/sample	http://sample	SAMPLE	ORDER
http://person	http://person	JOHN	PERSON
http://www.itso.org/dog	dog.xsd	SAMPLE	PETS
http://www.itso.org/pets	http://sample	SAMPLE	PETS

4 record(s) selected.

5.2.4 XSR support on the Control Center

You can also use the Control Center to manage an XSR object. DB2 9 Control Center supports adding and dropping XML schemas.

To view XML schemas, highlight the folder **XML Schema Repository (XSR)** under database in **Object View**. XML is displayed at the upper right window of the Control Center.

Figure 5-9 shows three schemas with SQL identifiers: SAMPLE.ORDER, JOHN.PERSON and SAMPLE.PETS

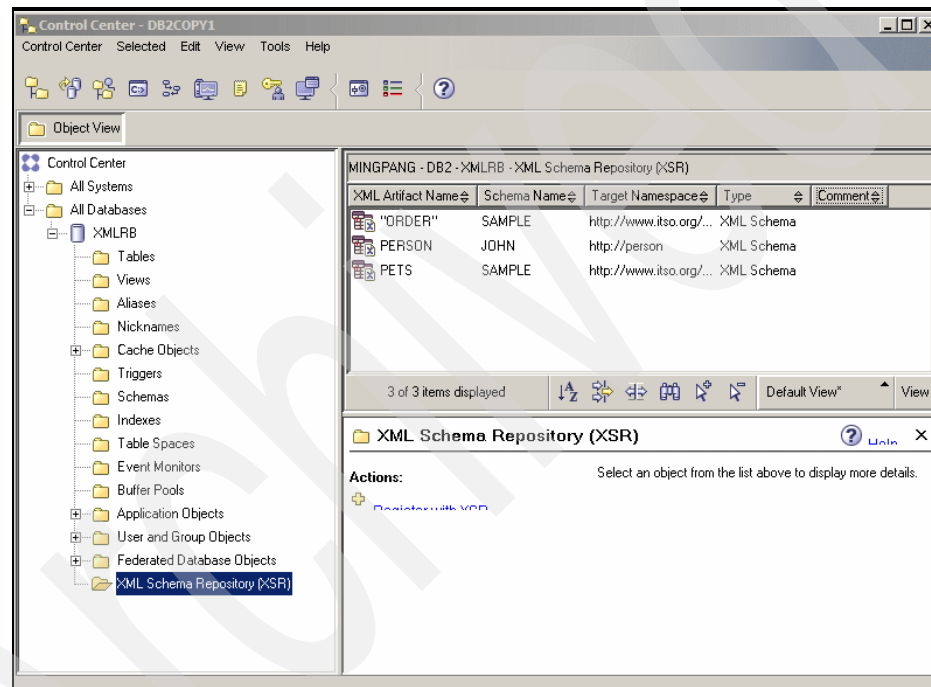


Figure 5-9 XML Schema Repository (XSR) in control center

You can create a filter to filter out the schema you do not want to see. Go to **Selected** → **Filter** → **Create**. You can see a pop-up window as shown in Figure 5-10. You can filter out schemas based on the predicates that you set on XML Artifact Name, Schema Name, Target Namespace, Type, and Comment.

Column	Comparison	Values
XML Artifact Name	LIKE	
Schema Name	=	SAMPLE
Target Namespace	LIKE	
Type	LIKE	
Comment	LIKE	

☒ Meet all conditions
 ☐ Meet any conditions

Clear

OK Cancel Delete Help

Figure 5-10 Creating a filter

5.2.5 Schema evolution

Sometimes, business rules change and an XML schema must be changed in order to reflect the new rules. For example, a chain store has an XML schema order. The order element has name, quantity, and price as elements and weight and color as attributes. Example 5-25 shows the XML schema order.xsd.

Example 5-25 order.xsd

```

<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="item">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="quantity" type="xsd:integer"/>
      <xsd:element name="price" type="xsd:integer"/>
    </xsd:sequence>
    <xsd:attribute name="weight" type="xsd:integer"/>
    <xsd:attribute name="color" type="xsd:string"/>
  </xsd:complexType>
  <xsd:element name="order" >
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="order" type="item" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>

```



```

        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
</xsd:schema>

```

The chain store is expanding and starting to offer free store memberships to the customers. Store member customers receive special discounts on some items. The chain store decides to let customers know how much they are saving when buying as store members. The customer will see the savings on each purchased item in the receipts. The schema `order.xsd` must be changed to reflect the new business rules. The change is called schema *evolution*. Example 5-26 shows the new XML Schema Definition, `order_new.xsd`. The `order_new.xsd` is compatible with the `order.xsd` in Example 5-25 on page 206 . The change is adding a new element `discount`. The element `discount` has attribute `minOccurs` set to 0 because not every item has a discount price.

Example 5-26 order_new.xsd

```

<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:complexType name="item">
        <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="quantity" type="xsd:integer"/>
            <xsd:element name="price" type="xsd:integer"/>
            <xsd:element name="discount" type="xsd:integer" minOccurs="0"/>
        </xsd:sequence>
        <xsd:attribute name="weight" type="xsd:integer"/>
        <xsd:attribute name="color" type="xsd:string"/>
    </xsd:complexType>
    <xsd:element name="order" >
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="order" type="item" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>

```

You should register the new schema file `order_new.xsd`. The application should validate the new document instance with the new registered schema. If you do not want to change the application and you use the explicit validation, you can drop the old schema and register the new schema with information that is identical with the old schema.

5.3 IMPORT, EXPORT, and RUNSTATS

In this section we discuss DB2 9 utilities that were enhanced to support the XML data type. We cover IMPORT, EXPORT, and RUNSTATS.

5.3.1 IMPORT

The IMPORT utility can be used to move data into DB2 database tables. In DB2 9, the IMPORT utility was enhanced to support the XML data type. With this enhancement, you can use the IMPORT utility to insert one or more XML data files to DB2 relational tables with XML data type columns.

You can only import well-formed XML documents because the columns defined as XML data types can only contain complete XML documents. If you are importing a data file with a row containing a document that is not well-formed, it will be rejected by DB2. The IMPORT utility will treat an XML document as Unicode unless the importing document contains a declaration tag specifying the encoding attribute.

XML data can be imported into a DB2 table with or without XML schema validation. Without XML schema validation, IMPORT inserts well-formed XML documents into the database without checking if the data in your XML documents is valid. In the following sections we explore how importing XML data into a DB2 9 table can be achieved with and without XML schema validation.

In this section, we show you, by examples, how to import data into XML columns with or without validating the XML data. We discuss in detail the new IMPORT utility options. In our examples, we use a CUSTOMER table for demonstration. The CUSTOMER table can be created using the following command:

```
CREATE TABLE customer(id INT NOT NULL PRIMARY KEY, name VARCHAR(20),  
customer_info XML);
```

Importing XML data

Importing XML data into a table with an XML data type column without validation is simple and straightforward. There is not much difference from importing data into a relational table without the XML data column. This is a simple IMPORT command example:

```
IMPORT FROM "C:\XmlRedbook\PROG\import\customer.del" OF DEL  
XML FROM "C:\XmlRedbook\PROG\import"  
INSERT INTO db2admin.customer;
```

Preparing a data file

For XML data, DB2 supports ASC, DEL, and IXF input file formats. You can create the delimited (DEL) ASCII file using the editor of your choice. Each line of the data file represents a row for inserting into the target table. For XML data stored in a file, you must specify an additional parameter, XML Data Specifier (XDS), in the data file to instruct the IMPORT utility to read the XML documents from the file.

Example 5-27 shows an XML document to be imported into the XML column CUSTOMER_INFO.

Example 5-27 XML file ContactInfo1.xml for CUSTOMER column

```
<?xml version="1.0"?>
<ContactInfo>
  <Address>
    <Street>22 Willow Street </Street>
    <City>Los Gatos </City>
    <State>CA </State>
    <Zip>95030</Zip>
  </Address>
  <Phone>
    <work>408-677-8888 </work>
    <home>408-588-9999 </home>
    <mobile>408-345-6666 </mobile>
  </Phone>
</ContactInfo>
```

Example 5-28 shows the import data file customer.del we prepared. The XDS specifies the document file name.

Example 5-28 Delimited ASCII file for input to DB2 IMPORT

```
10000,"Sarah Young","<XDS FIL='contactInfo1.xml' />"
```

XML Data Specifier (XDS)

An XML data specifier (XDS) is used in the IMPORT input file to describe the information about the actual XML data in the column; such information includes the name of the file that contains the actual XML data, and the offset and length of the XML data within that file.

The XDS can have up to four attributes. One of the attributes is mandatory and the rest of the attributes are optional. Where there is no XDS specified for an XML column in the data file, a NULL value will be inserted into the corresponding column, unless you specified otherwise. The IMPORT utility will treat a blank line in your data file as a record. You must make sure that you have no blank line in your data file or the IMPORT utility will insert a NULL row into your table.

The following XDS attributes are included:

- ▶ *FIL* is the name of the system file in which the XML document is stored. This attribute is required.
- ▶ *OFF* is the byte offset of the XML data in the file specified in the FIL attribute. The offset starts from zero.
- ▶ *LEN* is the length of the XML data in the file specified in the FIL attribute.
- ▶ *SCH* is the fully qualified XML schema name that is used for validating the XML documents.

For each row in the delimited input data file, the number of XDSs must be equal to or less than number of XML columns in a table. For our sample table, we have one XML column, so we can have zero to one XDS for each row. If we have a row in data file with two or more XDSs, that row will be rejected by the DB2 IMPORT utility. Example 5-29 shows an import data file with two XDSs in a row for a table with two XML columns.

Example 5-29 Input file with multiple XDSs

```
10000,"Sarah Young","<XDS FIL='contactInfo1.xml' />","<XDS FIL='file1.xml' />"
10002,"Jadan Phillips","<XDS FIL='contactInfo2.xml' />","<XDS FIL='file2.xml' />"
```

Importing data

With the data file, XML source file, and the database table created, we can import the data using the new XML FROM option as shown in the following IMPORT command:

```
IMPORT FROM "C:\XmlRedbook\PROG\import\customer.del" OF DEL
XML FROM "C:\XmlRedbook\PROG\import"
INSERT INTO db2admin.customer;
```

If you have XML files that reside in more than one location, you can specify the IMPORT command to look into more than one path as follows:

```
IMPORT FROM "C:\XmlRedbook\PROG\import\customer.del" OF DEL
      XML FROM "C:\XmlRedbook\PROG\import", "C:\XmlRedbook\data\xml\"
INSERT INTO db2admin.customer;
```

The sample output of the DB2 IMPORT command is shown in Example 5-30.

Example 5-30 IMPORT without validation sample output

```
IMPORT FROM "C:\XmlRedbook\PROG\import\customer.del" OF DEL
      XML FROM "C:\XmlRedbook\PROG\import"
INSERT INTO db2admin.customer
```

```
SQL3109N The utility is beginning to load data from file
"C:\XmlRedbook\PROG\import\customer.del".
```

```
SQL3110N The utility has completed processing. "1" rows were read
from the input file.
```

```
SQL3221W ...Begin COMMIT WORK. Input Record Count = "1".
```

```
SQL3222W ...COMMIT of any database changes was successful.
```

```
SQL3149N "1" rows were processed from the input file. "1" rows were
successfully inserted into the table. "0" rows were rejected.
```

```
Number of rows read      = 1
Number of rows skipped   = 0
Number of rows inserted  = 1
Number of rows updated   = 0
Number of rows rejected  = 0
Number of rows committed = 1
```

```
SELECT * FROM DB2ADMIN.CUSTOMER WHERE ID=10000
```

ID	NAME	CONTACT_INFO
10000	Sarah Young	<ContactInfo><Address><Street>22 Willow Street </Street><City>Los Gatos </City><State>CA </State><Zip>95030</Zip></Address><Phone><work>408-677-8888 </work><home>408-588-9999 </home><mobile>408-345-6666 </mobile></Phone></ContactInfo>

If you have XML files that reside in a different path than the one you specified in the IMPORT command, DB2 IMPORT will raise an error SQL3229N with reason code '1' stating that the file name cannot be found.

Import with XML schema validation

You can import XML data into a DB2 table with XML data type column with or without validation. To validate your XML data when importing into DB2 relational table, in addition to a data file, you also require an XML schema to be used for validation. The XDS parameter in the main data file requires additional attributes.

We continue using the CUSTOMER table to show how to import XML data with validation. Example 5-31 shows another XML document for our example.

Example 5-31 Content of contactinfor2.xml

```
<?xml version="1.0"?>
<ContactInfo>
  <Address>
    <Street>555 Lincoln Blvd</Street>
    <City>San Jose</City>
    <State>CA</State>
    <Zip>95136</Zip>
  </Address>
  <Phone>
    <work>408-677-8888</work>
    <home>408-588-9900</home>
    <mobile>408-345-7777</mobile>
  </Phone>
</ContactInfo>
```

XML schema

To validate XML documents during import, you must have an XML schema that specifies the acceptable XML elements, the order of the elements, the minimum and maximum occurrences of elements, data types, and required or optional elements. For our sample XML schema, we use IBM Rational® Software Development to generate an XML schema, but you can use any text editor or tool to create an XML schema. The IBM Rational Software Development was previously known as WebSphere Application Development Studio.

The schema we create checks every imported XML document for the following information:

- For every group of customer information, the order is ID, NAME, then CONTACT_INFO.

- ▶ For every group of contact information, the order is ADDRESS and then PHONE.
- ▶ For every group of address information, the order is Street, City, State, and Zip.
- ▶ For Address, provide at least one address; you can have up to two of them.
- ▶ For Phone, there must be at least one phone number and you can have up to three Phone elements per group.
- ▶ For Zip information, the value must be at least a five-digit number or nine-digit number in one of these formats: xxxxx or xxxxx-yyyy.

For the XML schema `contactInfo.xsd`, see A.2, “`contactInfo.xsd`” on page 366.

Before you can validate against a specific schema, you must register it with DB2. The following command will register `customerInfo.xsd` with DB2:

```
REGISTER XMLSCHEMA 'C:\XmlRedbook\PROG\import/'
FROM C:\XmlRedbook\PROG\import\contactInfo.xsd
AS DB2ADMIN.CONTACTINFO;
```

The fully qualified SQL identifier of the XML schema will be used in the XSD file to instruct DB2 to validate data using the schema during import. In our example, it is `DB2ADMIN.CONTACTINFO`.

XDS attributes for importing data with validation

The SCH attribute in the XDS specifies the schema used to perform schema validation. Example 5-32 shows what the XDS parameter should look like with the schema attribute SCH specified:

Example 5-32 Sample XDS with SCH attributes

```
<XDS FIL='contactInfo1.xml' SCH='DB2ADMIN.CONTACTINFO' />
<XDS FIL='contactInfo2.xml' SCH='DB2ADMIN.CONTACTINFO' />
```

Importing data with validation

The XDS information is included in the main data file to provide the information about the XML files to be imported for each row. Our sample data file, `customer2.del`, with XDS specified, is shown in Example 5-33.

Example 5-33 Sample data file for import XML data with validation

```
10001,"Sarah Young","<XDS FIL='contactInfo1.xml'
SCH='DB2ADMIN.CONTACTINFO' />",
10002,"Jadan Phillips","<XDS FIL='contactInfo2.xml'
SCH='DB2ADMIN.CONTACTINFO' />"
```

Following is an IMPORT command example with the new option XMLVALIDATE indicating that the data will be validated during importing:

```
IMPORT FROM "C:\Import\DATA\customer2.del" OF DEL
      XML FROM "C:\Import\DATA"
      XMLVALIDATE USING XSD
INSERT INTO db2admin.customer;
```

If we modify our contactInfo1.xml document and remove all three phone elements, the modified file looks as shown in Example 5-34:

Example 5-34 Modified contactInfo1.xml file

```
<?xml version="1.0"?>
<ContactInfo>
  <Address>
    <Street>22 Willow Street </Street>
    <City>Los Gatos </City>
    <State>CA </State>
    <Zip>95030</Zip>
  </Address>
  <Phone>
  </Phone>
</ContactInfo>
```

We also modified the zip code in the file contactInfo2.xml by adding a dash(-) after the fifth digit. The modified file is shown in Example 5-35.

Example 5-35 Modified contactInfo2.xml

```
<?xml version="1.0"?>
<ContactInfo>
  <Address>
    <Street>555 Lincoln Blvd</Street>
    <City>San Jose</City>
    <State>CA</State>
    <Zip>95136-</Zip>
  </Address>
  <Phone>
    <work>408-677-8888</work>
    <home>408-588-9900</home>
    <mobile>408-345-7777</mobile>
  </Phone>
</ContactInfo>
```

Validation against the registered schema DB2ADMIN.CONTACTINFO will raise error SQL16123N for contactinfo1.xml because our schema defined that at least one contact phone is required in an input XML document. For the second file contactInfo2.xml, we should see error SQL16210N because the input data violates the defined format for zip code, which was xxxxx or xxxxx-xxxx, where x is a number from 0 to 9.

The output of the IMPORT command is shown in Example 5-36.

Example 5-36 IMPORT command output

```
----- Commands Entered -----
IMPORT FROM "C:\Import\DATA\customer.del" OF DEL XMLVALIDATE USING XDS
INSERT INTO DB2ADMIN.CUSTOMER;
SELECT * FROM DB2ADMIN.CUSTOMER;
-----

IMPORT FROM "C:\Import\DATA\customer.del" OF DEL
      XMLVALIDATE USING XDS
INSERT INTO DB2ADMIN.CUSTOMER

SQL3109N The utility is beginning to load data from file
"C:\Import\DATA\customer.del".

SQL3148W A row from the input file was not inserted into the table.
SQLCODE "-16123" was returned.

SQL16123N XML document contains an element "((work,home),mobile)" with
empty content where the content model requires content for this
element.
SQLSTATE=2200M

SQL3185W The previous error occurred while processing data from row
"1" of the input file.

SQL3148W A row from the input file was not inserted into the table.
SQLCODE "-16210" was returned.

SQL16210N XML document contained a value "95136-" that violates a
facet constraint. Reason code = "13". SQLSTATE=2200M

SQL3185W The previous error occurred while processing data from row
"2" of the input file.

SQL3110N The utility has completed processing. "2" rows were read
from the input file.
```

```
SQL3221W ...Begin COMMIT WORK. Input Record Count = "2".
```

```
SQL3222W ...COMMIT of any database changes was successful.
```

```
SQL3149N "2" rows were processed from the input file. "0" rows were  
successfully inserted into the table. "2" rows were rejected.
```

```
Number of rows read           = 2  
Number of rows skipped        = 0  
Number of rows inserted       = 0  
Number of rows updated        = 0  
Number of rows rejected       = 2  
Number of rows committed     = 2
```

```
SELECT * FROM DB2ADMIN.CUSTOMER
```

ID	NAME	CONTACT_INFO
----	------	--------------

0 record(s) selected.

IMPORT command and options

DB2 9 IMPORT utility features the following new options to support the XML data type. In this section, we look at each of the options and give some examples of how to use each of these options with the DB2 9 IMPORT command.

New DB2 9 IMPORT options include:

- XML FROM pathname:

This option indicates the path that contains the XML files you want to import. If this option is not specified, the DB2 IMPORT utility will assume that the imported XML file resides in the same path as the input relational data file.

```
IMPORT FROM "C:\Import\DATA\customer.del" OF DEL XML FROM . INSERT  
INTO db2admin.customer;
```

If you have XML file locates in more than one path, use a comma(,) to separate the path. For example, in the following command:

```
IMPORT FROM "C:\Import\DATA\customer.del" OF DEL  
XML FROM "C:\Import\XML", "C:\Import\DATA"  
INSERT INTO db2admin.customer;
```

The IMPORT utility will look for the input XML files in C:\Import\XML and C:\Import\Data paths.

► **MODIFIED BY:**

Two new MODIFIED BY file-type mode options are added for XML data, MODIFIED BY XMLCHAR and MODIFIED BY XMLGRAPHIC.

These two options specify that the incoming XML data is encoded in the character or graphic code page. Most commonly the code page is ASCII or UTF-8. The MODIFIED BY XMLCHAR is valid for delimited and nondelimited ASCII file types. Following is an IMPORT command with MODIFIED BY XMLCHAR options:

```
IMPORT FROM "C:\XmlRedbook\PROG\sample_loan_app.del" OF DEL
XML FROM "C:\XmlRedbook\PROG"
MODIFIED BY XMLCHAR INSERT INTO db2admin.customer;
```

The MODIFIED BY XMLGRAPHICs option is useful when incoming XML documents are encoded in a specific graphic code page but have no encoding declaration at the beginning of the XML document. The MODIFIED BY XML GRAPHIC is available to use with delimited and non-delimited ASCII data file types.

If you import an XML document that contains an encoding attribute, the encoding must match the character or graphic code page value or the row is rejected. The code page value is the value specified by the CODEPAGE file type modifier or the graphic component of the application code page. If the modifier is not specified for the IMPORT command, the application default character code page is used. For more information about character code pages, see the DB2 9 publication, *XML Guide for DB2 Version 9*, SC10-4254:

ftp://ftp.software.ibm.com/ps/products/db2/info/vr9/pdf/letter/en_US/db2xge90.pdf

If the XML file is encoded with the ASCII character code page, you can use the CODEPAGE file type modifier as shown in the following command:

```
IMPORT FROM "C:\Import\DATA\customer.del" OF DEL
XML FROM "C:\Import\XML"
MODIFIED BY CODEPAGE=367 XMLCHAR INSERT INTO db2admin.customer;
```

If the XML file is encoded with UTF-16, you can specify the CODEPAGE file type modifier with matching graphic code page for the encoding UTF-16 as shown in the following example:

```
IMPORT FROM "C:\XmlRedbook\PROG\import\graphic.del" OF DEL
XML FROM "C:\XmlRedbook\PROG\import\"
MODIFIED BY CODEPAGE=1204 XMLGRAPHIC
INSERT INTO db2admin.customer;
```

Note: If you specify the MODIFIED BY XMLGRAPHIC option with the IMPORT command, the XML document to be imported must be encoded in the UTF-16 code page, and the file type modifier CODEPAGE value must match the UTF-16 code page, or the row is rejected.

► XMLPARSE STRIP/PRESERVE WHITESPACE:

This option specifies to remove or not to remove white space when XML documents are parsed. When the XMLPARSE option is omitted, the parser behavior for XML documents will be determined by the value of the CURRENT XMLPARSE OPTION special register.

Example 5-37 shows the IMPORT command using the XMLPARSE STRIP WHITESPACE option and the data inserted.

Example 5-37 IMPORT with XMLPARSE STRIP WHITESPACE option

```
IMPORT FROM "C:\Import\DATA\customer.del" OF DEL
XML FROM "C:\Import\XML" XMLPARSE STRIP WHITESPACE
INSERT INTO db2admin.customer;
```

```
SELECT * FROM db2admin.customer WHERE id=10000;
```

ID	NAME	CONTACT_INFO
10000	Sarah Young	<ContactInfo><Address><Street>22 Willow Street </Street><City>Los Gatos </City><State>CA </State><Zip>95030 </Zip></Address><Phone><work>408-677-8888 </work><home>408-588-9999 </home><mobile>408-345-6666 </mobile></Phone></ContactInfo>

Example 5-38 shows the output of the CUSTOMER table where the XML file is imported with the whitespace preserved.

Example 5-38 IMPORT command using XMLPARSE PRESERVE WHITESPACE.

```
IMPORT FROM "C:\Import\DATA\customer.del" OF DEL
XML FROM "C:\Import\DATA\" XMLPARSE PRESERVE WHITESPACE
INSERT INTO db2admin.customer
```

```
SELECT * FROM db2admin.customer WHERE id=10001
```

ID	NAME	CONTACT_INFO
10001	Jay Martins	<ContactInfo> <Address>

```

        <Street>555 Lincoln Blvd</Street>
        <City>San Jose</City>
        <State>CA</State>
        <Zip>95136</Zip>
    </Address>
    <Phone>
        <work>408-677-8888</work>
        <home>408-588-9900</home>
        <mobile>408-345-7777</mobile>
    </Phone>
</ContactInfo>

```

1 record(s) selected.

► **XMLVALIDATE USING XDS:**

This option indicates that XML documents are validated against a schema. The schema used for validation is determined by the SCH attribute of the XML Data Specifier (XDS) for each row within the main data file. The USING XDS is a default option when the XMLVALIDATE option is invoked.

In the case where the SCH attribute is omitted, no schema validation will occur unless you specified that a default schema is to be used by the DEFAULT schema_qualifier.schema_name clause. The following example shows an example of the XMLVALIDATE USING XDS option in the IMPORT command:

```

IMPORT FROM "C:\Import\DATA\customer.del" OF DEL
    XML FROM "C:\Import\XML"
    XMLVALIDATE USING XDS
INSERT INTO db2admin.customer;

```

► **XMLVALIDATE USING XDS DEFAULT schema_sqlid:**

This option can only be used when the USING XDS parameter is specified to modify the schema determination behavior.

The schema specified through the DEFAULT clause indicates the schema to be used for validation when an SCH attribute of the XDS is omitted. This DEFAULT clause takes precedence over the IGNORE and MAP clause to be addressed in the following paragraph. The DEFAULT, IGNORE, and MAP clauses apply to the specifications of the XDS and not to each other. In the following example, the schema name to be used as default for validation in the above IMPORT command is DB2ADMIN.CONTACTINFO.

```

IMPORT FROM "C:\Import\DATA\customer.del" OF DEL
    XML FROM "C:\Import\XML"
    XMLVALIDATE USING XDS DEFAULT DB2ADMIN.CONTACTINFO
INSERT INTO db2admin.customer;

```

► XMLVALIDATE USING XDS IGNORE schema_sqlid:

Similar to XMLVALIDATE USING XDS DEFAULT schema_sqlid, this option can only be used when the USING XDS option is specified to modify the schema determination behavior. You can use this option to indicate one or more schemas to ignore if they are identified by an SCH attribute.

In the case where an SCH attribute exists in your XDS, and the schema identified by your SCH attribute is included in the list of schemas to IGNORE, no schema validation will take place for the imported XML documents. The command in Example 5-39 tells the IMPORT utility to ignore schema DB2ADMIN.LOAN_APP.

Example 5-39 IMPORT: ignore XML validation

```
IMPORT FROM "C:\Import\DATA\customer.del" OF DEL
XML FROM "C:\Import\XML"
XMLPARSE PRESERVE WHITESPACE
XMLVALIDATE USING XDS DEFAULT DB2ADMIN.CONTACTINFO
IGNORE ( DB2ADMIN.LOAN_APPL )
INSERT INTO db2admin.customer;
```

► XMLVALIDATE USING XDS MAP (schema_sqlid, schema_sqlid):

This option can be used when the USING XDS parameter is specified. You can use the MAP clause to indicate alternate schemas to be used for validation in place of the schemas identified by the SCH attribute of the XML Data Specifier (XDS). The MAP clause indicates a list of one or more schema pairs, where each pair represents a mapping of the original schema to a substitute schema. The original schema is specified by the SCH attribute in an XDS, and the substitute schema is the one that should be used for schema validation.

The IMPORT command in Example 5-40 tells the IMPORT utility to use DB2ADMIN.CUSTOMER for validation instead of the original schema specified by the SCH attribute in the input data file, which is DB2ADMIN.CONTACTINFO.

Example 5-40 IMPORT validation using DB2ADMIN.CUSTOMER

```
IMPORT FROM "C:\Import\DATA\customer.del" OF DEL
XML FROM "C:\Import\XML"
XMLPARSE PRESERVE WHITESPACE
XMLVALIDATE USING XDS DEFAULT DB2ADMIN.CONTACTINFO
IGNORE ( DB2ADMIN.LOAN_APPL )
MAP ( ( DB2ADMIN.CONTACTINFO, DB2ADMIN.CUSTOMER ) )
INSERT INTO db2admin.customer;
```

► XMLVALIDATE USING SCHEMA schema_sqlid:

This option is used to indicate that all XML documents are to be validated against a specific XML schema. In this case, the SCH attribute of the XDS will be ignored for all XML columns.

The following sample command in Example 5-41 tells the DB2 IMPORT utility to ignore whatever schema name is specified in the input data file and just use the schema DB2ADMIN.CUSTOMER for validation.

Example 5-41 Using DB2ADMIN.CUSTOMER for validation

```
IMPORT FROM "C:\Import\DATA\customer.del" OF DEL
XML FROM "C:\Import\XML"
XMLPARSE PRESERVE WHITESPACE
XMLVALIDATE USING SCHEMA DB2ADMIN.CUSTOMER
INSERT INTO db2admin.customer;
```

► XMLVALIDATE USING SCHEMALOCATION HINTS:

This option indicates that documents are validated against the schemas identified by XML schema location hints in the source XML documents. If a schemaLocation (SCH) attribute in the source XML document is not found, no validation will take place. When this option is specified, the SCH attribute of the XDS within the main data file will be ignored for all XML columns. Example 5-42 shows a sample command for XMLVALIDATE USING SCHEMALOCATION HINTS.

Example 5-42 SCHEMALOCATION HINTS

```
IMPORT FROM "C:\Import\DATA\customer.del" OF DEL
XML FROM "C:\Import\XML"
XMLPARSE PRESERVE WHITESPACE
XMLVALIDATE USING SCHEMALOCATION HINTS
INSERT INTO db2admin.customer;
```

For detailed information about these options, refer to the *Data Movement Utilities Guide and Reference*, SC10-4227.

5.3.2 EXPORT

The EXPORT utility extracts data from DB2 tables to one or more files on your system. The exported files can be used to import to tables in a different database on the same server or a different server. DB2 9 supports exporting XML data in delimited (DEL) and integrated exchanged format (IXF).

Exporting XML data

The EXPORT utility can generate DEL and IXF data files that contain the validation information in the XML Data Specifier (XDS). With DB2 9, XML becomes a first-class data type; the EXPORT utility introduces several new options to support exporting XML data.

Just as with LOBs, the exported XML data is always stored in a separate file from the main data file containing your exported relational data. The main exported data file contains one or more XML data specifiers (XDSs) for each row where XML data is present. The XDS holds the information that points to the XML data and information about XML schema that is already saved. One XDS is required for every XML column with XML data in the table. The XML data can be exported with or without XML schema information.

EXPORT command and options

Before we use the new export options for XML data type, we take a look at each of the options and give some examples to illustrate how to use them in the EXPORT command:

► XML TO pathname:

This option indicates one or more paths to the directory in which the exported XML files are to be stored. If the option XML TO is omitted, the XML file will be written to the same path to which the exported relational data is written. The following sample command shows how the XML TO option is used.

```
EXPORT TO <path/main_data_file_name> OF DEL
      XML TO <path/output.xml>
SELECT * FROM db2admin.customer;
```

The main data file is written to C:\Export\DATA\customer.del, and all XQuery data model (QDM) instances are written to files C:\Export\XML\customer.del.001.xml.

When more than one path is specified with XML TO option in the EXPORT command, DB2 export utility will cycle between the paths to write each successive XQuery Data Model (QDM) instance to the appropriate XML file. Example 5-43 shows the EXPORT command with the XML TO option to direct the XML file to be written to separate paths.

Example 5-43 Export with XML TO option specified output more than one path

```
EXPORT TO "C:\Export\DATA\customer.del" OF DEL
      XML TO "C:\Export\DATA", "C:\export\XML"
SELECT * FROM db2admin.customer;
```

This command generates the main data file customer.del in C:\export\DATA. The QDM instances are written to two file locations as specified in the

EXPORT command. They are C:\Export\DATA\customer.del.001.xml and C:\Export\XML\customer.del.002.xml.

► **XMLFILE filename:**

This option indicates the base file names to be used for XML data. This option works similarly to export LOB data. The following example shows how this option is used in the EXPORT command:

```
EXPORT TO "C:\Export\DATA\customer.del" OF DEL
XML TO "C:\Export\DATA", "C:\export\XML"
XMLFILE "CUSTINFO"
SELECT * FROM db2admin.customer;
```

This command generates the main data file customer.del under the C:\Export\DATA directory. The QDM instances are written to two locations as specified in the EXPORT command. They are C:\Export\DATA\CUSTINFO.001.xml and C:\export\XML\CUSTINFO.002.xml.

As we have learned from DB2 8, the LOBINFILE option provides a means to specify the base name of the LOB file generated by the EXPORT utility. Similarly, in DB2 9, the XMLFILE option specifies the name of the XML file generated by the export utility.

By default, the XML file base name is the name of the exported data file with an .xml extension. The full name of the XML file consists of the base name, followed by a number extension that is padded to three digits, and the .xml extension. For LOB, the full name would consist of the base name, followed by a number extension that is padded to three digits and a .lob extension.

► **MODIFIED BY XMLNODEDECLARATION:**

This option in Example 5-44 indicates that QDMs are written without an XML declaration tag. QDM instances are exported with an XML declaration tag that includes an encoding attribute at the beginning of the XML file by default.

Example 5-44 Modifying by XMLNODEDECLARATION

```
EXPORT TO "C:\Export\DATA\customer.del" OF DEL
XML TO "C:\Export\DATA", "C:\export\XML"
XMLFILE "CUSTINFO"
MODIFIED BY XMLNODEDECLARATION
SELECT * FROM db2admin.customer;
```

► **MODIFIED BY XMLCHAR:**

This option in Example 5-45 on page 224 indicates that QDM instances are written in the character code page. The character code page is the value that can be controlled by specify the CODEPAGE file type modifier. If it is not specified, the application code is applied. QDM instances are written out in Unicode (UTF-8) by default.

Example 5-45 Modifying BY XMLCHAR

```
EXPORT TO "C:\Export\DATA\customer.del" OF DEL
XML TO "C:\Export\DATA", "C:\export\XML"
XMLFILE "CUSTINFO"
```

MODIFIED BY XMLCHAR

```
SELECT * FROM db2admin.customer;
```

► MODIFIED BY XMLGRAPHIC:

This options indicates that the encoding to be used for the exported XML document is XMLGraphic codepage. When this modifier is used in the EXPORT command, your exported XML document will be encoded in UTF-16 regardless of the CODEPAGE file type modifier or the application code page. See, Example 5-46.

Example 5-46 Modifying by XMLGRAPHIC

```
EXPORT TO "C:\Export\DATA\customer.del" OF DEL
XML TO "C:\Export\DATA", "C:\export\XML"
XMLFILE "CUSTINFO"
```

MODIFIED BY XMLGRAPHIC

```
SELECT * FROM db2admin.customer;
```

► MODIFIED BY XMLINSEPFILLES:

When this option is specified, each QDM instance is written to a separate file. By default all exported XML documents are concatenated together in the same file. When you specify MODIFIED BY XMLINSEPFILLES, each of the exported XML document will be placed in a separate file. See Example 5-47.

Example 5-47 Modifying by XMLINSEPFILLES

```
EXPORT TO "C:\Export\DATA\customer.del" OF DEL
XML TO "C:\Export\DATA", "C:\export\XML"
XMLFILE "CUSTINFO"
```

MODIFIED BY XMLINSEPFILLES

```
SELECT * FROM db2admin.customer;
```

► MODIFIED BY LOBINSEPFILLES:

Like the XMLINSEPFILLES option, this means each LOB value is to be written into a separate file. By default, multiple values are concatenated together in the same exported LOB file. Example 5-48 and Example 5-49 show the EXPORT command with the MODIFIED BY LOBSINSEPFILLES option.

In Example 5-48, all LOBs values are written to the file C:\Export\LOB\CUST_INFO.001.lob and all XML data are written to the file C:\Export\XML\CUST_INFO.001.xml.

Example 5-48 Using MODIFIED BY LOBSINSEPFIL option - all LOBs are in one file

```
EXPORT TO "a" OF DEL
  LOBS TO "C:\Export\LOB" LOBFILE "CUST_INFO"
  XML TO "C:\Export\XML"
  XMLFILE "CUST_INFO"
  MODIFIED BY LOBSINFIL
SELECT * FROM db2admin.customer;
```

In Example 5-49, each LOB value is written to a separate file under the specified directory C:\Export\LOB. The base name for the exported LOB file is CUST_INFO.00X.lob, where X increases with each LOB value to be exported. Each QDM instance is written to C:\Export\XML\CUST_INFO.001.xml.

Example 5-49 Using MODIFIED BY LOBSINSEPFIL option - separated LOB files

```
EXPORT TO "C:\Export\DATA\customer.del" OF DEL
  LOBS TO "C:\Export\LOB"
  LOBFILE "CUST_INFO"
  XML TO "C:\Export\XML"
  XMLFILE "CUST_INFO"
  MODIFIED BY LOBSINFIL LOBSINSEPFILS
SELECT * FROM db2admin.customer;
```

► **XMLSAVSCHEMA:**

This option indicates that XML schema information should be saved for all XML columns. For each exported XML document that was validated against an XML schema at the time it was inserted, this option specifies that the name of the XML schema used to validate the XML document be written to the corresponding XML Data Specifier (XDS) as an SCH attribute.

In the case where the exported document was not validated against an XML schema or that XML schema no longer existed in the database, the SCH attribute will be omitted in the XDS.

The EXPORT command with XMLSAVSCHEMA option is shown in this example:

```
EXPORT TO "C:\Export\DATA\customer.del" OF DEL XMLSAVSCHEMA
SELECT * FROM db2admin.customer;
```

The main data file is written to file C:\Export\DATA\customer.del, and the QDM instance is written to XML file C:\Export\DATA\customer.del.001.xml. The content of the main data file looks as follows:

```
10000,"Sarah Young", "<XDS FIL='customer.del.001.xml' OFF='0'
LEN='273' SCH='DB2ADMIN.CONTACTINFO' />"
10001,"Jay Martins", "<XDS FIL='customer.del.001.xml' OFF='273'
LEN='266' SCH='DB2ADMIN.CONTACTINFO' />"
```

Example 5-50 shows how to specify an XML file location and file name prefix.

Example 5-50 Specify XML file location and prefix

```
EXPORT TO "C:\Export\DATA\customer.del" OF DEL
XML TO "C:\export\XML"
XMLFILE "CUSTINFO"
MODIFIED BY XMLINSEPFILES XMLSAVESCHEMA
SELECT * FROM db2admin.customer;
```

Each XML file is written to a separate file with the default file names
C:\export\XML\CUSTINFO.001.xml and C:\export\XML\CUSTINFO.0002.xml.

Export command examples and outputs

Now that we have seen all the new EXPORT options that support the XML data type, we can apply some of these options with our sample CUSTOMER table to export the XML data.

Example 5-51 shows a sample EXPORT command with XML TO, XMLFILE, and MODIFIED BY XMLINSEPFILES options and the output the command produces.

Example 5-51 EXPORT command and output

```
----- Commands Entered -----
CONNECT TO DEMO;
SELECT * FROM DB2ADMIN.CUSTOMER;
EXPORT TO "C:\Export\DATA\customer.del" OF DEL
XML TO "C:\Export\XML"
XMLFILE "CUST_INFO"
MODIFIED BY XMLINSEPFILES XMLSAVESCHEMA
SELECT * FROM DB2ADMIN.CUSTOMER;
-----

Database Connection Information

Database server          = DB2/NT 9.1.0
SQL authorization ID     = DB2ADMIN
Local database alias     = DEMO

SELECT * FROM DB2ADMIN.CUSTOMER

ID          NAME          CONTACT_INFO
-----
10000      Sarah Young    <ContactInfo><Address><Street>22
Willow Street </Street><City>Los Gatos </City><State>CA
```

```
</State><Zip>95030</Zip></Address><Phone><work>408-677-8888  
</work><home>408-588-9999 </home><mobile>408-345-6666  
</mobile></Phone></ContactInfo>
```

```
10001      Jay Martins      <ContactInfo><Address><Street>555  
Lincoln Blvd</Street><City>San  
Jose</City><State>CA</State><Zip>95136</Zip></Address><Phone><work>408-  
677-8888</work><home>408-588-9900</home><mobile>408-345-7777</mobile></  
Phone></ContactInfo>
```

2 record(s) selected.

```
EXPORT TO "C:\Export\DATA\customer.del" OF DEL  
XML TO "C:\Export\XML"  
XMLFILE "CUST_INFO"  
MODIFIED BY XMLINSEFILES XMLSAVESCHEMA  
SELECT * FROM DB2ADMIN.CUSTOMER
```

SQL3104N The Export utility is beginning to export data to file
"C:\Export\DATA\customer.del".

SQL3105N The Export utility has finished exporting "2" rows.

Number of rows exported: 2

Two XML files produced as the result of the EXPORT command are
C:\export\XML\CUSTINFO.001.xml and C:\export\XML\CUSTINFO.002.xml.
The main data file C:\Export\DATA\customer.del produced with content is shown
in Example 5-52.

Example 5-52 Main data file created by EXPORT

```
10000,"Sarah Young","<XDS FIL='CUST_INFO.001.xml'  
SCH='DB2ADMIN.CONTACTINFO' />"  
10001,"Jay Martins","<XDS FIL='CUST_INFO.002.xml'  
SCH='DB2ADMIN.CONTACTINFO' />"
```

When you specify the XML TO option for the EXPORT command, be sure that the path name is correct or the EXPORT command will fail. If you specify the EXPORT command with the XMLTO option to a directory or path that does not exist, the EXPORT command raises the error SQL3235N stating that the EXPORT utility cannot use the specified path. Example 5-53 illustrates the error received when you specify the bad path name.

Example 5-53 EXPORT command with XMLTO option to non-existing path

```
----- Commands Entered -----
EXPORT TO "C:\Export\DATA\customer.del" OF DEL
      XML TO "C:\export\BADFOLERNAME\"
      XMLFILE "CUSTINFO"
      MODIFIED BY XMLINSEPFILLES
SELECT * FROM DB2ADMIN.CUSTOMER;
SELECT * FROM DB2ADMIN.CUSTOMER;
-----
```

```
EXPORT TO "C:\Export\DATA\customer.del" OF DEL
      XML TO "C:\export\BADFOLERNAME\"
      XMLFILE "CUSTINFO"
      MODIFIED BY XMLINSEPFILLES
SELECT * FROM DB2ADMIN.CUSTOMER
```

SQL3235N The utility cannot use the "XML" path
"C:\export\BADFOLERNAME\
parameter as specified. Reason code: "3".

SQL3235N The utility cannot use the "XML" path
"C:\export\BADFOLERNAME\
parameter as specified. Reason code: "3".

Explanation:

One of the following reason codes may apply:

1 Either the path "<path-name>" is not a valid sqlu_media_list
or the values provided are not valid. The media_type must be
SQLU_LOCAL_MEDIA and all path names must be terminated with a
valid path separator.

2 There is not enough space on the paths provided for the EXPORT
utility to hold all the data of type "<type>".

3 The path "<path-name>" cannot be accessed.

User Response:

Determine which reason code applies above, correct the problem,
and resubmit your command.

5.3.3 RUNSTATS

The RUNSTATS command updates statistics about physical characteristics of table columns and associated indexes. These statistics include information such as number of records, number of pages, and average record length. These statistics are used by the DB2 optimizer to determine the optimal access paths to the data.

As XML becomes a native data type in DB2 9, the RUNSTATS utility has been updated to collect XML column statistics. The statistics collected are at both the XML document and node level, and can be used for cost estimation when selecting execution plans. This information was not available on the catalog for the user to view or update in the first release of DB2 9.

XML column statistics

DB2 9 collects XML statistics by using sampling technique. The DB2 RUNSTATS command calls an XML statistics collection routine, which invokes the XML runtime routine to traverse through the XML document in each row of an XML column to gather statistics.

The statistics collected for XML columns are composed of path distribution data and path-value distribution data at the document and node levels. This data is collected by sampling every XML document in the table and determining all the various XPath expressions contained in the documents. For every XPath expression, DB2 collects path distribution data and path-value distribution data.

- ▶ Path distribution data contains the number of rows that contain the XPath and the number of times the XPath is encountered within each XML document. This data can be described in terms of <pathid, value, docCount, nodeCount> where:
 - pathid is the XPath expression.
 - value is result of the XPath expression.
 - docCount is the number documents that contains a matching value for the XPath.
 - nodeCount is the number of XML element has the matching value for the XPath.
- ▶ Path-Value distribution data contains the value of the XPath expression, the number of rows that contain the same value for the XPath and the number of times in each XML document the same value for the XPath was encountered. This data can be represented as <pathid, value, docCount, nodeCount>.

Statistics collection on XML type columns is governed by two DB2 database system registry values:

- ▶ DB2_XML_RUNSTATS_PATHID_K
- ▶ DB2_XML_RUNSTATS_PATHVALUE_K

These two parameters are similar to the NUM_FREQVALUES parameter in that they specify the number of frequency values to collect. If not set, a default of 200 will be used for both parameters.

RUNSTATS considerations with XML columns and indexes

Keeping the statistics up-to-date is required for the most efficient access to the data. You should consider running the RUNSTATS utility to collect statistics on XML columns in the following situations:

- ▶ When XML data has been imported into a table and indexes have been created
- ▶ When new XML indexes are created
- ▶ When the XML documents in the table have been extensively updated
- ▶ When the XML indexes have been updated extensively, where in this case, extensively might mean 10 to 20 percent of the XML indexes
- ▶ When a table has been reorganized with the REORG utility
- ▶ When you want to compare current and previous XML statistics

In DB2 8, RUNSTATS is a stateless command that wipes out previously collected statistics when a new RUNSTATS command is issued. In DB2 9, when the RUNSTATS utility is used to collect statistics for XML columns only, all existing statistics for non-XML columns previously collected are retained. For the XML columns with previously collected statistics, they will be merged or updated with the XML columns statistics collected by the current RUNSTATS command.

Automatic statistics collection by default will include all XML columns and indexes. The automatic statistics collection feature analyzes differences between statistics over time to determine when it must execute RUNSTATS again. These decisions are currently based solely on relational statistics. Therefore, any changes on XML columns would not affect the autoRunstats decision. In DB2 9, the autoRunstats cannot be configured to exclude XML columns.

The performance of RUNSTATS is directly related to the size and the complexity of your XML documents. One thing to keep in mind is that the performance of RUNSTATS on XML columns depends heavily on the traversal operation. If you have a very large table that contains a significant number of complex XML documents, it can take time for RUNSTATS to traverse through each document.

RUNSTATS command support

In DB2 9, the RUNSTATS utility has been updated to support the collection of statistics on a DB2 table that contains XML columns and indexes over XML data.

Other RUNSTATS options that are supported for non-XML columns in V8 are kept unchanged for non-XML columns, but they are not applicable or not supported for XML columns in DB2 9. When you supply these options for XML columns in the RUNSTATS command, they are simply ignored.

Because the DB2 9 RUNSTATS utility collects very basic statistics for XML data, there are no extra command line options for RUNSTATS over XML columns introduced in DB2 91.

The following RUNSTATS command formats are supported in DB2 9:

► **RUNSTATS ON TABLE** *schemaName.tableName*

This command collects columns statistics on all columns of the table. For each XML column in the table, the basic XML statistics are collected.

Assuming that we have an EMPLOYEE table defined with these columns:

```
CREATE TABLE db2admin.employee (c1 int, c2 int, xcol1 XML, xcol2 XML);
```

The following RUNSTATS command will collect statistics for all columns, including XML statistics for both xcol1 and xcol2:

```
RUNSTATS ON TABLE db2admin.employee;
```

► **RUNSTATS ON TABLE** *schemaName.tableName* **ON COLUMNS** *columnList*

This command collects basic column statistics for all columns included in the *columnList*. For each XML column in the *columnList*, the basic XML statistics are collected.

Assuming that we have the same EMPLOYEE table, the following command collects XML statistics for xcol1 and xcol2:

```
RUNSTATS ON TABLE db2admin.customer ON COLUMNS(xcol1,xcol2);
```

If you want to collect XML statistics for xcol1 only, the format for the command is written this way:

```
RUNSTATS ON TABLE db2admin.employee ON COLUMN(xcol1);
```

- ▶ **RUNSTATS ON TABLE** *schemaName.tableName* **ON COLUMNS** *columnList1* **WITH DISTRIBUTION ON COLUMNS** *columnList2*

This command collects basic column statistics on all columns included in *columnList1* plus distribution statistics for all columns included in *columnList2*. For each XML column that is included in either *columnList1* or *columnList2*, the basic column statistics are collected, because the basic XML column statistics are the same as the distribution statistics for any XML column.

Using the **EMPLOYEE** table, the following two commands collect XML statistics for both *xcol1* and *xcol2*:

```
RUNSTATS ON TABLE db2admin.employee ON COLUMN(c1) WITH DISTRIBUTION;
```

```
RUNSTATS ON TABLE db2admin.employee ON COLUMNS(c1) WITH DISTRIBUTION  
ON COLUMNS (xcol2, xcol2);
```

If you only want to collect XML statistics for column *xcol1*, the **RUNSTATS** command would look similar to this:

```
RUNSTATS ON TABLE DB2ADMIN.EMPLOYEE ON COLUMNS(xcol1) WITH  
DISTRIBUTION(xcol1);
```

- ▶ **RUNSTATS ON TABLE** *schemaName.tableName* **EXCLUDING XML COLUMNS**

For the convenience of users, DB2 9 supports a new clause for a **RUNSTATS** utility called **EXCLUDING XML COLUMNS**. You can specify the **EXCLUDING XML COLUMNS** clause in the **RUNSTATS** command to exclude all XML columns from statistics collection if statistics for XML columns are not required or if you want to have XML columns statistics collected at another time.

The **EXCLUDING XML COLUMNS** clause takes precedence over all other clauses that specify XML columns for **RUNSTATS**, so be aware that the **EXCLUDING XML COLUMNS** can be ambiguous at times.

For example, in the following **RUNSTATS** command:

```
RUNSTATS ON TABLE db2admin.employee ON COLUMNS(c1, xcol2) WITH  
DISTRIBUTION ON ALL COLUMN EXCLUDING XML COLUMN
```

No statistics for XML column *xcol2* are collected even though you explicitly specify the *xcol2* in the column list, because you have the **EXCLUDING XML COLUMNS** clause specified. The **RUNSTATS** command simply omitted all of the XML columns from statistics collection in this case.

Note: In DB2 9, **RUNSTATS** does not support the **KEY COLUMNS** clause for the XML data type because an XML type column cannot be a key column.

5.4 XML data security

In this section, we introduce two ways to achieve access control on data stored in an XML column:

- ▶ Access control at row-level and column-level
- ▶ Access control at XML nodes-level

Row-level access control means that you can control which users are allowed to access which rows. *Column-level* access control means that you can control user access at the column-level. To achieve these controls, we can use the new DB2 9 feature label-based access control (LBAC).

An access control at the XML node level means that you can control user access level on the XML elements or attributes inside one XML document. To achieve this, we can use the VIEW and XMLTABLES function.

5.4.1 LBAC

LBAC is a new DB2 9 security feature that provides a configurable capability to control access on individual rows and columns. A security administrator who is granted with SECADM authority performs the LBAC security setup. The security administrator configures the LBAC system by creating security policies. A *security policy* describes the criteria that are used to decide who has access to specific data. Under the security policy, the security administrator creates security labels and associates the label with the rows and columns to be protected. The security administrator also associates labels with users. The LBAC policies compare the data labels with the user's label to determine if the user has access to the specific row or column.

Security labels have a new data type, DB2SECURITYLABEL. A new SECURED WITH option is added to the CREATE TABLE and ALTER TABLE statements to associate the security label with the rows or columns.

SECADM

SECADM authority is a brand new authority in DB2 9. SECADM is aimed to centralize security privileges. The abilities given by SECADM are not given by any other authority, not even SYSADM. Functions that only SECADM is allowed to perform are as follows:

- ▶ Create and drop security label components.
- ▶ Create and drop security policies.
- ▶ Create and drop security labels.
- ▶ Grant and revoke security labels.
- ▶ Grant and revoke LBAC rule exemptions.

- ▶ Grant and revoke SETSESSIONUSER privileges.
- ▶ Execute the SQL statement TRANSFER OWNERSHIP on objects that you do not own.

5.4.2 Row and column-level access control

In this section we show you how to implement Label-based access control (LBAC) to control the user access in row and column level. LBAC is a new DB2 9 security feature that provides the capability to control read and write access in more granular level in the row and column. As a result, data security is greatly increased.

A tutorial for learning document-level security using DB2 9 pureXML and LBAC can be found at the following Web site:

<http://www.ibm.com/developerworks/edu/dm-dw-dm-0607williams-i.html>

Employee records scenario

In this scenario we have a small company that stores all the employee information in a database. The employee information is categorized into general information and confidential information. Only the personnel in the Human Resource Department can access the confidential personal information. The general employee information can be accessed by all employees. The data security policy further prohibits the regular employee from viewing managers' general information.

Figure 5-11 illustrates the data security requirements. USERA in the Human Resource Department can access all the employee data, including both confidential and general information. The regular employee, USERB, can only see the general information of employees with non-management roles, such as engineer and architect.

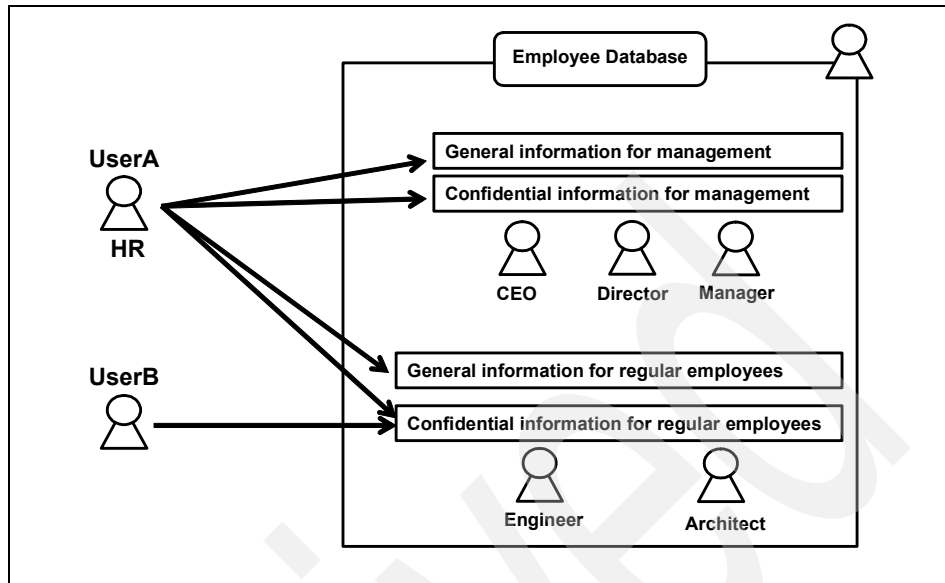


Figure 5-11 New security policy

In order to adopt this security policy, we separate the employee information into two parts, general and confidential. Each of them is described as an XML document and stored into two XM columns. Example 5-54 shows a sample of general information in XML format.

Example 5-54 General information

```
<Employee>
  <Name>John Smith1</Name>
  <EmpNo>001</EmpNo>
  <Title>Manager</Title>
  <Phone type="work">312-964-0001</Phone>
  <Email>john.smith1@my.com</Email>
</Employee>
```

Example 5-55 is a sample of employee confidential information in XML format.

Example 5-55 Confidential information

```
<Employee>
  <Name>John Smith1</Name>
  <EmpNo>001</EmpNo>
  <DateOfBirth>2/21/1967</DateOfBirth>
  <SSN>892-76-0001</SSN>
```

```

<Address country="US">
  <Street>1 East Main Street</Street>
  <City>Los Gatos</City>
  <State>CA</State>
  <Zip>95034</Zip>
  <Phone type="home">678-181-0001</Phone>
</Address>
<Salary>10000</Salary>
</Employee>

```

The employee table has an ID column, two columns for XML documents, and a column for LBAC. Figure 5-12 illustrates the data model for the employee table EMP. The general information of each employee is stored in EMP1 column and the confidential information is stored in EMP2 column. USERA from HR is authorized to access all the data in the EMP table. The regular employee USERB can only access column EMPID and EMP1 in row#2 and row#3 where row#2 is a record of an engineer and row#3 is a record of an architect. The columns USERB can access are highlighted in gray.

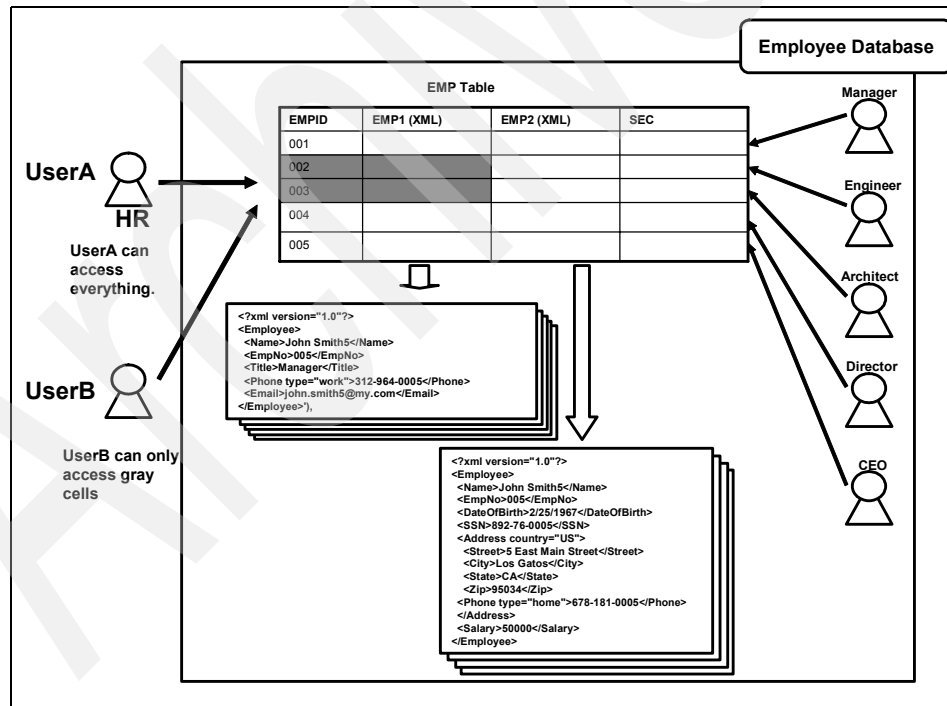


Figure 5-12 Data model of EMP table and access scope of USERA and USERB

Implementing LBAC

In this section, we show the procedure of implementing data security using LBAC. Following are the steps required to set up the LBAC:

1. Appointing a security administrator
2. Creating a security label component
3. Creating a security policy
4. Creating a security label
5. Adding security policy to table
6. Assigning security labels to users
7. Verifying the security setting

Appointing a security administrator for LBAC

To implement LBAC, the administrator ID must have security administration authority SECADM. No other DB2 authorities, including SYSADM, can implement LBAC. In our scenario, we create a user ID *db2lbac* to be used as security administrator. We grant SECADM authority to the db2lbac user by using the following commands:

```
CONNECT to XMLRB user db2admin using db2admin;  
GRANT secadm on database to user db2lbac;
```

Creating an LBAC security label component

An LBAC security label component is used to model a security structure of an organization. In this scenario, we use an array type for the LBAC security label component. Within the array, the sequence of the element represents the level of trust. The first element has the highest trust level and the last element has the lowest trust level. In our example, the SECRET element is higher than the NONCLASSIFIED element.

Log in to the database as db2lbac to create an LBAC security label component using the following commands:

```
CONNECT TO XMLRB USER db2lbac USING db2lbac  
CREATE SECURITY LABEL COMPONENT emp_comp ARRAY  
['SECRET', 'NONCLASSIFIED']
```

Creating an LBAC security policy

An LBAC security policy defines the security label component to be used and the rules. We create an LBAC security policy EMP_POLICY to attach to the EMP table by using the following command:

```
CREATE SECURITY POLICY emp_policy COMPONENTS emp_comp WITH DB2LBACRULES  
RESTRICT NOT AUTHORIZED WRITE SECURITY LABEL
```

The **RESTRICT NOT AUTHORIZED WRITE SECURITY LABEL** clause indicates that the insert or update operation will fail if the user is not authorized to write the explicitly specified security label that is provided in the **INSERT** or **UPDATE** statement.

Creating an LBAC security label

An LBAC security label describes a certain set of security criteria for protecting data. Once defined, a security label can be granted to allow users to access protected data. When a user tries to access data protected by LBAC, DB2 checks the user's authorized security label with the security label of the data and determines if the user has the authority to access that data.

The syntax of security label is:

```
CREATE SECURITY LABEL [label] COMPONENT [security label component name]
[element name]
```

If you want to create two security labels for **USERA** and **USERB** with the **EMP_COMP** security label component, those should be similar to the following:

```
CREATE SECURITY LABEL emp_policy.hr_only COMPONENT emp_comp 'SECRET'
CREATE SECURITY LABEL emp_policy.public COMPONENT emp_comp
'NONCLASSIFIED'
```

You can grant each LBAC security label to users by using the commands shown in Example 5-56.

Example 5-56 Granting LBAC security labels

```
GRANT SECURITY LABEL emp_policy.hr_only TO USER usera FOR ALL ACCESS
GRANT SECURITY LABEL emp_policy.public TO USER userb FOR ALL ACCESS
GRANT SECURITY LABEL emp_policy.hr_only TO USER db2admin FOR ALL ACCESS
GRANT SECURITY LABEL emp_policy.public TO USER db2admin FOR ALL ACCESS
GRANT EXEMPTION ON RULE DB2LBACWRITEARRAY WRITEDOWN FOR emp_policy TO
USER db2admin
```

The **GRANT EXEMPTION** statement gives **DB2ADMIN** an access rule exception for the security policy. With this exception, **DB2ADMIN** can create a table and insert data into tables that are associated with **EMP_POLICY**.

Adding security policy to the table

The security policy can be added to a table while creating a table or by altering the table. In this example, we show you how to add a security policy in the **CREATE TABLE** statement. Example 5-56 shows the statements used to create the **EMP** table.

Example 5-57 Adding the security policy

```
CONNECT TO XMLRB USER db2admin USING db2admin
CREATE TABLE "EMP" (
  ID char(3) NOT NULL PRIMARY KEY,
  EMP1 XML,
  EMP2 XML SECURED WITH HR_ONLY,
  SEC DB2SECURITYLABEL) SECURITY POLICY EMP_POLICY;
```

The EMP2 column must be protected and it is defined with a SECURED WITH clause with the security label HR_ONLY. We also must protect those rows that store the managers' information. For this, we have the SEC column defined as a DB2SECURITYLABEL type for storing the LBAC information. When a row is inserted, the HR_ONLY label will be added. The SECURITY POLICY clause associates the security policy with the table.

After the table is created, we grant SELECT, INSERT, UPDATE, DELETE privileges to USERA and USERB on the EMP table using the following commands:

```
GRANT select, insert, update, delete ON db2admin.emp TO USER userb
GRANT select, insert, update, delete ON db2admin.emp TO USER usera
```

Verifying the LBAC security setting

We add some data into the table so we can verify the LBAC security setting we just implemented. Example 5-58 shows a sample INSERT statement for John Smith1, whose title is manager. Since his title is manager, this row should be protected by the HR_POLICY label. The SECLABEL_BY_NAME ('EMP_POLICY', 'HR_ONLY') places the HR_ONLY label into the SEC column so that this row is protected and only an authorized user can access the data. The INSERT statements for John Smith2 to John Smith5 are given in Appendix A, "Sample data" on page 361.

Example 5-58 Insert protected data

```
INSERT INTO EMP VALUES (
  '001',
  XMLPARSE( DOCUMENT
    '<?xml version="1.0"?>
    <Employee>
      <Name>John Smith1</Name>
      <EmpNo>001</EmpNo>
      <Title>Manager</Title>
      <Phone type="work">312-964-0001</Phone>
      <Email>john.smith1@my.com</Email>
    </Employee>'),
```

```

XMLPARSE( DOCUMENT
  '<?xml version="1.0"?>
    <Employee>
      <Name>John Smith1</Name>
      <EmpNo>001</EmpNo>
      <DateOfBirth>2/21/1967</DateOfBirth>
      <SSN>892-76-0001</SSN>
      <Address country="US">
        <Street>1 East Main Street</Street>
        <City>Los Gatos</City>
        <State>CA</State>
        <Zip>95034</Zip>
        <Phone type="home">678-181-0001</Phone>
      </Address>
      <Salary>10000</Salary>
    </Employee>'),
SECLABEL_BY_NAME('EMP_POLICY', 'HR_ONLY')));

```

The security setting can be verified by trying to access the data as USERA or USERB using the following SELECT statement:

```

SELECT ID,
xmlquery('$c/Employee/Name' passing emp1 as "c") as NAME,
xmlquery('$c/Employee/Title' passing emp1 as "c") as TITLE
FROM DB2ADMIN.EMP

```

Example 5-59 shows the result set from USERA and USERB using the same query. The result sets are different due to the security restriction. USERA who has authorized to use the HR_ONLY security label received all the records in the EMP table. USERB can only get two rows that contain the personnel records of non-management employees. The DB2 LBAC mechanism detects that USERB has an authority of PUBLIC label only and does not allow USERB to access data labeled with HR_ONLY, confirming that row-level protection is successfully implemented.

Example 5-59 Query result from USERA

```

-- Result set from USERA
NAME                                TITLE
-----
001 <Name>John Smith1</Name>        <Title>Manager</Title>
002 <Name>John Smith2</Name>        <Title>Engineer</Title>
003 <Name>John Smith3</Name>        <Title>Architect</Title>
004 <Name>John Smith4</Name>        <Title>Director</Title>
005 <Name>John Smith5</Name>        <Title>CEO</Title>

```

```
-- Result set from USERB
```

NAME	TITLE
002 <Name>John Smith2</Name>	<Title>Engineer</Title>
003 <Name>John Smith3</Name>	<Title>Architect</Title>

To check the column-level protection, the following query is run by USERA and USERB:

```
SELECT ID,  
xmlquery('$c/Employee/Name' passing emp1 as "c") as NAME,  
xmlquery('$c/Employee/Salary' passing emp2 as "c") as SALARY  
FROM DB2ADMIN.EMP
```

Example 5-60 shows the result from USERA and USERB. USERA can issue the XQuery on the EMP2 column because it is labeled as HR_ONLY, and USERA is authorized on this label. USERB receives an error message indicating that he is not authorized to read data from the EMP2 column. This confirms that the column-level access control is also successfully implemented.

Example 5-60 Result of accessing protected data

```
-- Result from USERA --  
001 <Name>John Smith1</Name> <Salary>10000</Salary>  
002 <Name>John Smith2</Name> <Salary>20000</Salary>  
003 <Name>John Smith3</Name> <Salary>30000</Salary>  
004 <Name>John Smith4</Name> <Salary>40000</Salary>  
005 <Name>John Smith5</Name> <Salary>50000</Salary>  
  
-- Result from USERB --  
SQL20264N For table "EMP", authorization ID "USERB" does not have  
"READ"  
access to the column "EMP2".  SQLSTATE=42512
```

5.4.3 Node-level access control

VIEW is a virtual table that can present data from several tables or from a subset of a table. Since DB2 provides access control on a view like a table, we can utilize this DB2 feature to achieve the node level access control on XML data.

Employee table scenario

In this scenario we continue use the employee table. The employee information is still categorized into general information and confidential information. However, the security requirements are less strict. The general employee information can be accessed by all employees. The personnel in the Human Resource Department can access the confidential personal information.

Figure 5-13 is a conceptual diagram of this scenario. USERA is a member of the Human Resource Department who can access everything in the employee database. USERB is a regular employee and is only allowed to access general information such as employee number, e-mail address, and work telephone.

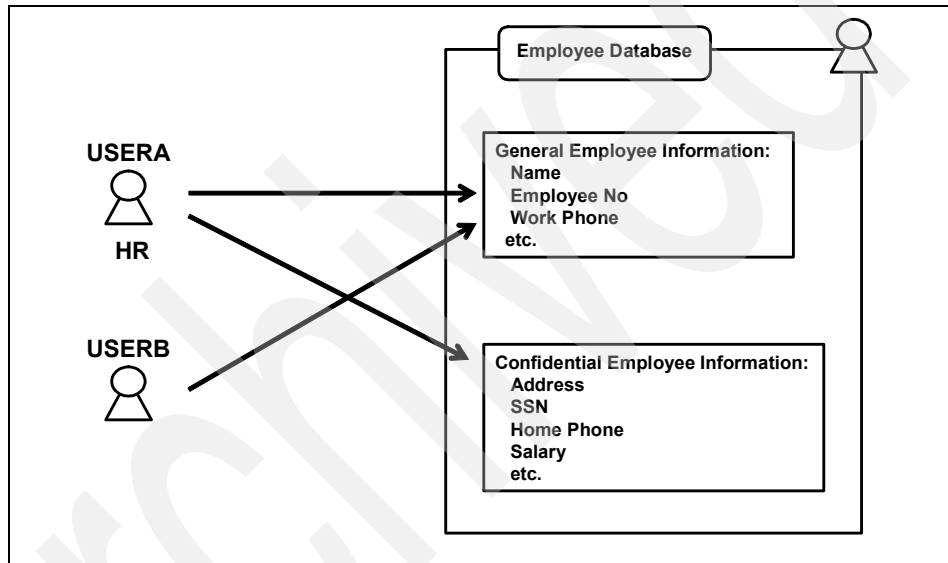


Figure 5-13 Logical diagram of security model

In this scenario, a table, EMPLOYEE, is created to store employees' information in XML format. There are five employees, John Smith1~ JohnSmith5, in this table. We want to define the range that USERA and USERB can access.

Figure 5-14 illustrates the logical model and access control required for the EMPLOYEE table. There are five XML documents stored in the EMP column in the EMPLOYEE table. USERA can access all information in XML documents, on the other hand, USERB can only access the following general information:

- ▶ /Employee/Name
- ▶ /Employee/EmpNo
- ▶ /Employee/Title
- ▶ /Employee/Email

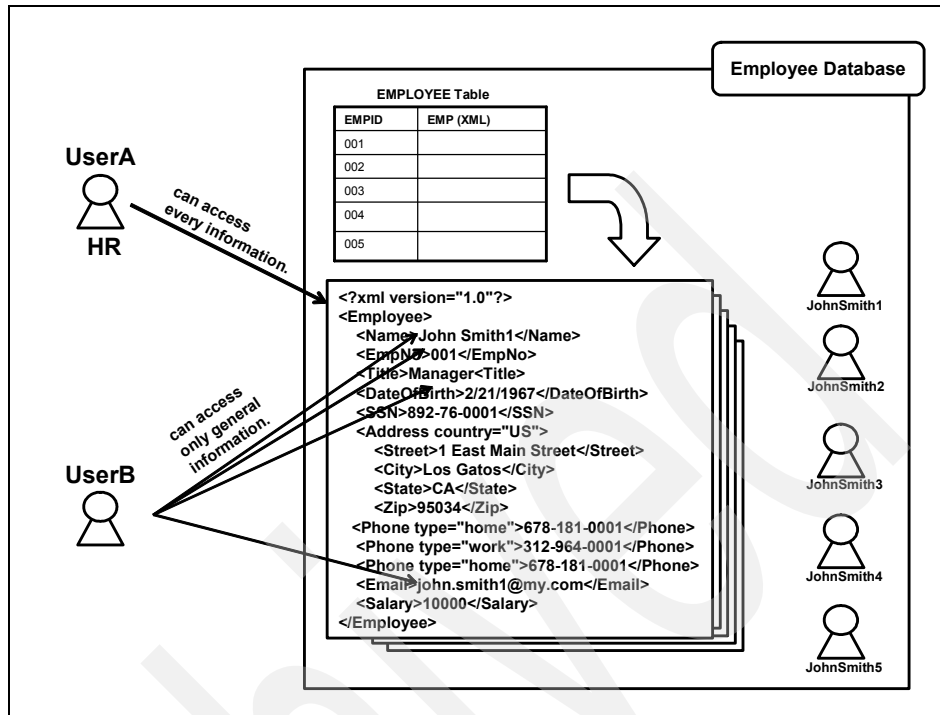


Figure 5-14 data model of EMPLOYEE table and access range for USERA and USERB

What do we do to construct a security model to meet these requirements? If you simply give USERA and USERB select authority for the EMPLOYEE table, USERB would be able to issue XQuery to the EMPLOYEE table and select the confidential data. The simplest way to achieve this data-access security requirement is using the view in DB2. You can map element and attribute values in XML documents into a relational column and then define views for USERA and USERB. There are several ways to map XML elements and attributes into relational columns. We use XMLTABLE functions in this scenario.

Figure 5-15 illustrates how data access can be restricted using views. Two views are defined for USERA and USERB. Both are based on the EMPLOYEE table. These two views reflect the result sets of the XMLTABLE function from the DB2ADMIN.EMPLOYEE table. We show the scenario setup and the creation of view commands in the next section.

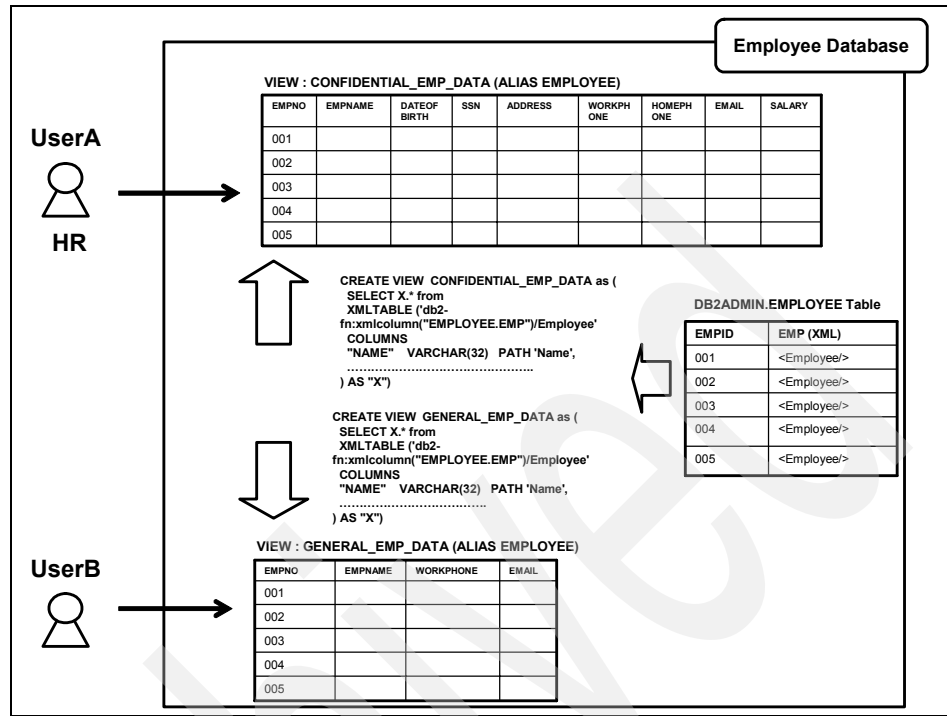


Figure 5-15 Restricting access via views

Establishing access control

In this section, we provide a step-by-step process for setting up node-level access control through a view. We provide database setup procedures and commands for creating views.

Creating user IDs

In this scenario, we use two user IDs, USERA and USERB. The IDs should be created in the operating system.

Creating the EMPLOYEE table

The EMPLOYEE table has two columns: ID and EMP columns. The EMP column stores employee information as XML documents. Use the following commands to create the EMPLOYEE table:

```
connect to XMLRB user db2admin using db2admin;
CREATE TABLE "EMPLOYEE" (
  ID char(3) NOT NULL PRIMARY KEY,
  EMP XML)
```

Importing XML document from files

Example 5-61 is one of the employee XML files that must be inserted. All the sample XML data are given in Appendix A, “Sample data” on page 361.

Example 5-61 employee001.xml

```
<?xml version="1.0"?>
<Employee>
  <Name>John Smith</Name>
  <EmpNo>001</EmpNo>
  <DateOfBirth>2/21/1967</DateOfBirth>
  <SSN>892-76-0001</SSN>
  <Address country="US">
    <Street>1 East Main Street</Street>
    <City>Los Gatos</City>
    <State>CA</State>
    <Zip>95034</Zip>
  </Address>
  <Phone type="work">312-964-0001</Phone>
  <Phone type="home">678-181-0001</Phone>
  <Email>john.smith1@my.com</Email>
  <Salary>10000</Salary>
</Employee>
```

The IMPORT command we used to move data into the table is:

```
IMPORT FROM impfile.txt OF DEL XML FROM . MODIFIED BY XMLCHAR REPLACE
INTO db2admin.employee
```

The XDS describing the XML data files is as shown in Example 5-62.

Example 5-62 impfile.txt

```
"001","<XDS FIL='employee001.xml' />"
"002","<XDS FIL='employee002.xml' />"
"003","<XDS FIL='employee003.xml' />"
"004","<XDS FIL='employee004.xml' />"
"005","<XDS FIL='employee005.xml' />"
```

Creating a view for USERA

USERA from the Human Resource Department has to access all elements and attributes in XML documents in EMPLOYEE.EMP column. You have to include all XML nodes to the XMLTABLE function. The CREATE VIEW command is displayed in Example 5-63.

Example 5-63 CREATE VIEW command

```
CREATE VIEW Db2admin.employee_a AS (  
  SELECT x.* FROM  
    XMLTABLE ('db2-fn:xmlcolumn("DB2ADMIN.EMPLOYEE.EMP")/Employee'  
    COLUMNS  
      "NAME"          VARCHAR(32) PATH 'Name',  
      "EMPNO"         VARCHAR(3)  PATH 'EmpNo',  
      "TITLE"         VARCHAR(12) PATH 'Title',  
      "DATEOFBIRTH"   VARCHAR(10) PATH 'DateOfBirth',  
      "SSN"           VARCHAR(11) PATH 'SSN',  
      "STREET"        VARCHAR(64) PATH 'Address/Street',  
      "CITY"          VARCHAR(12) PATH 'Address/City',  
      "STATE"         VARCHAR(2)  PATH 'Address/State',  
      "ZIP"           VARCHAR(5)  PATH 'Address/Zip',  
      "WORKPHONE"     VARCHAR(12) PATH 'Phone[@type="work"]',  
      "HOMEPHONE"     VARCHAR(12) PATH 'Phone[@type="home"]',  
      "EMAIL"         VARCHAR(32) PATH 'Email',  
      "SALARY"        INTEGER     PATH 'Salary'  
    ) AS "X"  
)
```

After creating the view, grant access privilege for EMPLOYEE_A view to USERA with the following command:

```
GRANT SELECT ON db2admin.employee_a TO USER usera
```

Create a view for USERB

Because USERB is allowed to access only five elements, the CREATE VIEW statement is shorter and displayed in Example 5-64.

Example 5-64 CREATE VIEW for USERB

```
CREATE VIEW db2admin.employee_b AS (  
  SELECT X.* from  
    XMLTABLE ('db2-fn:xmlcolumn("DB2ADMIN.EMPLOYEE.EMP")/Employee'  
    COLUMNS  
      "NAME"          VARCHAR(32) PATH 'Name',  
      "EMPNO"         VARCHAR(3)  PATH 'EmpNo',  
      "TITLE"         VARCHAR(12) PATH 'Title',  
      "WORKPHONE"     VARCHAR(12) PATH 'Phone[@type="work"]',  
      "EMAIL"         VARCHAR(32) PATH 'Email'  
    ) AS "X"  
)
```

Grant the access privilege for EMPLOYEE_B to USERB using the following command:

```
GRANT SELECT ON DB2ADMIN.EMPLOYEE_b TO USER userb
```

To have a unified name for all the views based on the EMPLOYEE table, USERA and USERB can create an alias for the views using the following commands:

```
CONNECT TO xmlrb USER usera USING usera;  
CREATE ALIAS employee FOR db2admin.employee_a;
```

```
CONNECT TO XMLRB USER USERB USING USERB;  
CREATE ALIAS employee FOR db2admin.employee_b;
```

Verifying security setup

After the views and aliases are created successfully, you are ready to test the access control by querying the data.

Connect to the database as USERA and issue a SELECT statement:

```
connecting xmlrb user usera using usera;  
SELECT * FROM employee;
```

Example 5-65 is the result set from the SELECT statement submitted by USERA. You can tell that all node values in the XML column are shredded into the relational columns.

Example 5-65 Result set of selecting all the data from EMPLOYEE view by USERA

NAME	EMPNO	TITLE	DATEOFBIRTH	SSN
John Smith1	001	Manager	2/21/1967	892-76-0001
John Smith2	002	Engineer	2/22/1967	892-76-0002
John Smith3	003	Architect	2/23/1967	892-76-0003
John Smith4	004	Director	2/24/1967	892-76-0004
John Smith5	005	CEO	2/25/1967	892-76-0005

STREET	CITY	STATE	ZIP
1 East Main Street	Los Gatos	CA	95034
2 East Main Street	Los Gatos	CA	95034
3 East Main Street	Los Gatos	CA	95034
4 East Main Street	Los Gatos	CA	95034
5 East Main Street	Los Gatos	CA	95034

WORKPHONE	HOMEPHONE	EMAIL	SALARY
312-964-0001	678-181-0001	john.smith1@my.com	10000
312-964-0002	678-181-0002	john.smith2@my.com	20000
312-964-0003	678-181-0003	john.smith3@my.com	30000
312-964-0004	678-181-0004	john.smith4@my.com	40000
312-964-0005	678-181-0005	john.smith5@my.com	50000

Connect to the database as USERB and issue the same SELECT query.

```
connect to xmlrb user userb using userb
SELECT * FROM EMPLOYEE
```

Example 5-66 is the result set. You can see that the identical query issued by USERA and USERB have returned different result sets.

Example 5-66 Result set of selecting all the data from EMPLOYEE view by USERB

NAME	TITLE	EMPNO	WORKPHONE	EMAIL
John Smith1	Manager	001	312-964-0001	john.smith1@my.com
John Smith2	Engineer	002	312-964-0002	john.smith2@my.com
John Smith3	Architect	003	312-964-0003	john.smith3@my.com
John Smith4	Director	004	312-964-0004	john.smith4@my.com
John Smith5	CEO	005	312-964-0005	john.smith5@my.com

From the test result, we verify that we have successfully set up the XML node level access control by view and XMLTABLE function. USERA can see all elements in the XML documents, and USERB can only select what they are allowed to see. By using both features provided in DB2 9, you can easily control the XML data access scope for each user.

Application development

This chapter covers various aspects of application development using DB2. The information contained here features topics and examples that are specific to application development involving XML. The subjects covered are:

- ▶ The database application development environment
- ▶ Application development tools
- ▶ Accessing pureXML from application overview
- ▶ XML and stored procedures
- ▶ Web services

The manuals listed here are suggested references for the DB2 application development topics that are covered in this chapter:

- ▶ *Call Level Interface Guide and Reference, Volume 1*, SC10-4224
- ▶ *Call Level Interface Guide and Reference, Volume 2*, SC10-4225
- ▶ *Command Reference*, SC10-4226
- ▶ *Developing ADO.NET and OLE DB Applications*, SC10-4230
- ▶ *Developing Embedded SQL Applications*, SC10-4232
- ▶ *Developing Java Applications*, SC10-4233
- ▶ *Developing Perl and PHP Applications*, SC10-4234
- ▶ *Developing SQL and External Routines*, SC10-4373
- ▶ *Getting Started with Database Application Development*, SC 10-4252
- ▶ *XML Guide*, SC10-4254

6.1 The database application development environment

The DB2 database application development environment is composed of several software elements:

- ▶ Operating system
- ▶ DB2 Client
- ▶ Database application programming interface (API)
- ▶ Programming language
- ▶ Transaction manager
- ▶ Development tools

Each of these elements requires some configuration for DB2 database application development.

To enable the DB2 database application development, the following prerequisites must be *installed* and *configured*:

- ▶ A supported operating system
- ▶ The DB2 Client
- ▶ The API driver(s) and, if required, driver manager(s)
- ▶ Compilers or interpreters required for the programming languages you will be using
- ▶ A transaction manager
- ▶ Application development tools

Note: For complete information regarding prerequisites, installation, and configuration for your application development environment, refer to the manual *Getting Started with Database Application Development*, SC10-4252.

6.2 Application development tools

In this section we describe and demonstrate the following tools for developing applications with XML:

- ▶ Developer Workbench
- ▶ DB2 Control Center
- ▶ DB2 Command Line processor
- ▶ IBM DB2 Visual Studio Add-In

Developer Workbench

The following support is provided for XML in the Developer Workbench (DWB):

- ▶ **Stored procedure support:**
Create and run stored procedures that contain XML data types as input or output parameters.
- ▶ **Data output support:**
View documents contained in XML columns as a tree or text.
- ▶ **SQL builder support:**
Build SQL expressions with XML functions and run SQL statements that contain XML host variables.
- ▶ **XML schema support:**
Manage schema documents in the XML schema repository (XSR), including registering and dropping schemas, as well as editing schema documents.
- ▶ **XML document validation support:**
Perform validation of XML documents against schemas registered in the XSR.
- ▶ **XQuery builder features:**
 - Build the XQuery statements visually by dragging and dropping nodes that represent elements in an XML schema or document.
 - Specify predicates, expressions, clauses, and sorting preferences for each node. XQuery builder then generates the query for you.
(Alternatively, you can write your own statements or modify the generated statements directly in the provided editing environment).
 - After the query is created, it can be tested by running it directly from the Developer Workbench.

DB2 Control Center

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

- ▶ **Creating tables with XML columns:**
Figure 6-1 shows the Control Center Add Column panel to add an XML column to a table.

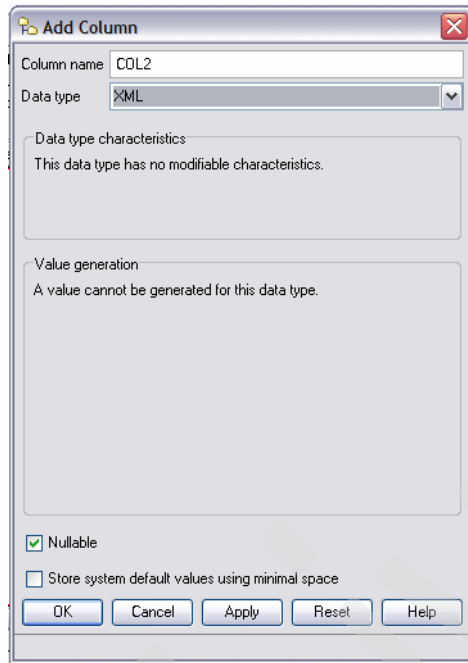


Figure 6-1 Using the Control Center to create a table with an XML column

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

Command line processor

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

- ▶ Issuing XQuery statements by prefixing them with the XQUERY keyword
- ▶ Importing and export XML data
- ▶ Collecting statistics on XML columns
- ▶ Calling stored procedures with IN, OUT, or INOUT parameters of XML data type

- ▶ Working with the XML schemas, DTDs, and external entities required to validate and process XML documents
- ▶ Reorganizing indexes over XML data and tables containing XML columns
- ▶ Decomposing XML documents

Figure 6-2 is an example of an XQuery issued from the Command Line processor.

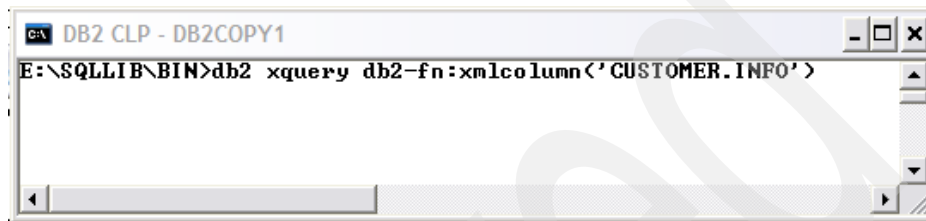


Figure 6-2 An XQuery issued from the Command Line Processor

DB2 Development Add-In for Microsoft Visual Studio .NET

Some general features of the DB2 development Add-In for Visual Studio include:

- ▶ IBM designers for working with database objects (such as tables, views, scripts, procedures, and result sets)
- ▶ Ability to generate and deploy DB2-based Web services without writing a single line of code
- ▶ Integration into Microsoft Server Explorer to perform database activity (such as managing connections and exploring database objects)
- ▶ Ability to create and debug SQL procedures (including common language runtime [CLR] stored procedures)
- ▶ Ability to clone DB2 database objects
- ▶ Schema cache for rapid access to schema information
- ▶ Advanced object filtering capabilities
- ▶ Importing and exporting capabilities from the data grid
- ▶ Integration with Microsoft Query Builder
- ▶ Parameter persistence support for rerun of routines
- ▶ Support for IBM family of data servers, including Informix® Dynamic Server (IDS), DB2 for z/OS®, and DB2 for iSeries™ in addition to DB2 for Linux, UNIX, and Windows
- ▶ Comprehensive support for new DB2 pureXML data type and features

Details of the DB2 Development Add-in for Visual Studio can be found in 6.8, “The DB2 .NET environment” on page 321.

6.2.1 Developer Workbench

The Developer Workbench (DWB) is available as a free download in DB2 9. It is a visual tool that aids in the rapid development of DB2 business objects. This newly designed tool is based on the Eclipse framework and replaces the Development Center from previous versions of DB2.

The following tasks can be executed with DWB:

- ▶ Create, view, and edit database objects (such as tables and schemas).
- ▶ Explore and edit data in tables and rows.
- ▶ Visually build SQL and XQuery statements.
- ▶ Develop and deploy stored procedures, user defined functions (UDFs), routines, and scripts.
- ▶ Debug SQL and Java stored procedures.
- ▶ Develop SQLJ applications.
- ▶ Develop queries and routines for XML data.
- ▶ Perform data movement (such as load and extract).
- ▶ Collaborate and share projects with team members.
- ▶ Migrate projects from the DB2 version 8 DB2 Development Center.

Thorough examination of the capabilities of DWB is beyond the scope of this IBM Redbook. Detailed information about this tool can be found in the tutorial *DB2 Developer Workbench, Part 1: DW Concepts and Basic Tasks*. This tutorial is available from developerWorks at the following URL:

http://www.ibm.com/developerworks/edu/dm-dw-dm-0608eaton-i.html?S_TACT=105AGX54&ca=dnw-729

XML support in Developer Workbench

The Developer Workbench contains support for XML, including:

- ▶ Support for the XML data type:
DWB allows the XML data type to be used in queries and routines. The XQuery contents of XML documents in the database can also be viewed, edited, and updated.

► Stored procedure support:

Stored procedures that contain XML data type (input or output) parameters or return XML data can be created and run.

► Data output view:

XML data type columns can be viewed on the results page, and the content of XML columns can be visualized as a tree or document text.

► XML Editor:

With the XML Editor, you can perform the following tasks:

- Create and edit XML documents
- Generate XML documents from an XML schema
- Annotated schema mapping tool

► Support for XML schema:

Existing XML schemas and XML schema documents can be loaded from the XML schema repository in the database and properties, such as target namespace or schema location, can be viewed. New XML schemas (and corresponding XML schema documents) can be registered or dropped.

► XML document validation:

XML value validation for XML documents against a registered XML schema can be performed.

► XQuery builder:

With the XQuery builder, you can complete queries without understanding XQuery semantics. An XML query can be built *visually* by selecting sample resultant nodes from a tree representation of a schema or XML document and dragging the nodes onto a return grid.

- After a node is listed on the return grid, you can drill down into the query to add predicates and sorting preferences.
- You can drill down multiple levels in a query to specify nested predicates, clauses, and expressions.
- After building the query, it can be run and tested directly from Developer Workbench.

6.2.2 Developer Workbench: Visual Query Builder overview

The XQuery Builder is a component of Developer Workbench. Its task is to create queries through a graphical user interface (GUI). As with other GUIs, one of the advantages of this tool is that it is not necessary for the user to be acquainted with the semantics of the XQuery language. The builder accomplishes its goal by creating a tree view of an instance of the XML documents that have to be queried, it then allows the user to drag and drop nodes from this tree to the design grids.

The design grids are context-sensitive and each grid represents different functions of the XQuery language. Users build the query by adding resources to these grids and then drilling into these resources. Different resources represent different functionalities in the XQuery language and thus present different sets of grids. This process of using context-sensitive grid sets (acting in unison) allows the tool to employ nuances of the language without the necessity for the user to understand the semantics of the language. The query is internally represented as an XML model and each resource that is added into the grid is added to the XML model.

Starting the XQuery Builder

The XQuery builder uses an XQuery builder file that has an .XQM extension. After this file is created, the query can be edited with the builder, saved, and edited again at a later time. The query can also be saved as a plain XQuery file, a flat text file. This allows the query to be used outside of the Developer Workbench.

To use XQuery Builder, the xquery must reside in a new or an existing project. The example here outlines the steps to follow when creating a *new* project that will contain XQuery files.

1. Open the Developer Workbench and ensure that the data perspective is open.

Tip: If the data perspective is not opened, you can open the data perspective, from the Menu by selecting:

Window → Open Perspective → Other ... → Data (default)

2. Choose the XML query wizard:

Select **File** → **New** → **Other**. The Select a Wizard window opens. See Figure 6-3. Choose **XML Query** and then select **NEXT**.

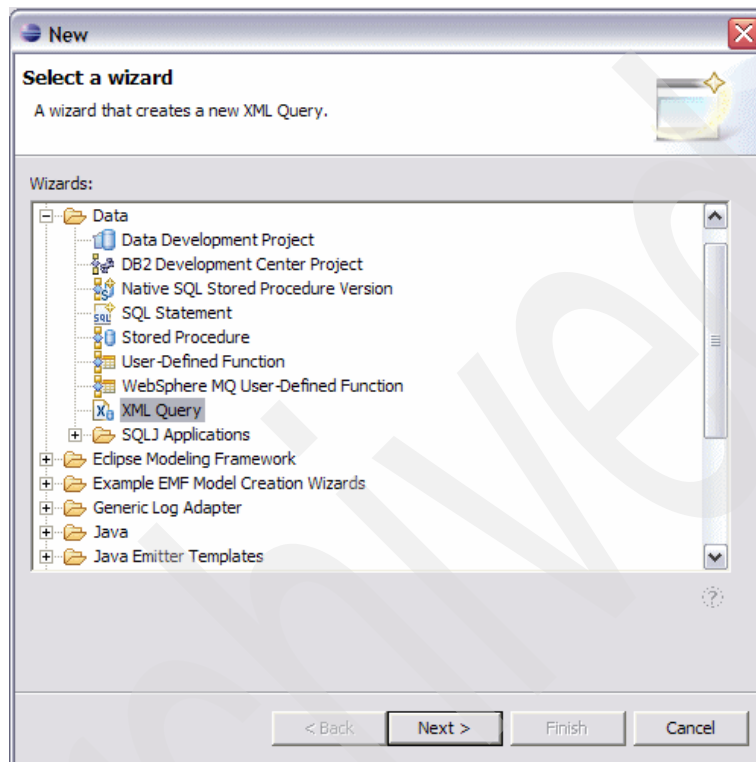


Figure 6-3 The Select a wizard dialog box showing XML Query selected

3. Create a new project, or specify an already existing one:

In the Specify a Project window, you can new project, or specify an already existing one. Choose **New** to create a new project *or* choose an existing project from the Project drop-down box. Click **NEXT**.

The *New Data Development Project* window opens. Specify a name for the new project. In our example a new project named *xmlLUW* is created. See Figure 6-4.

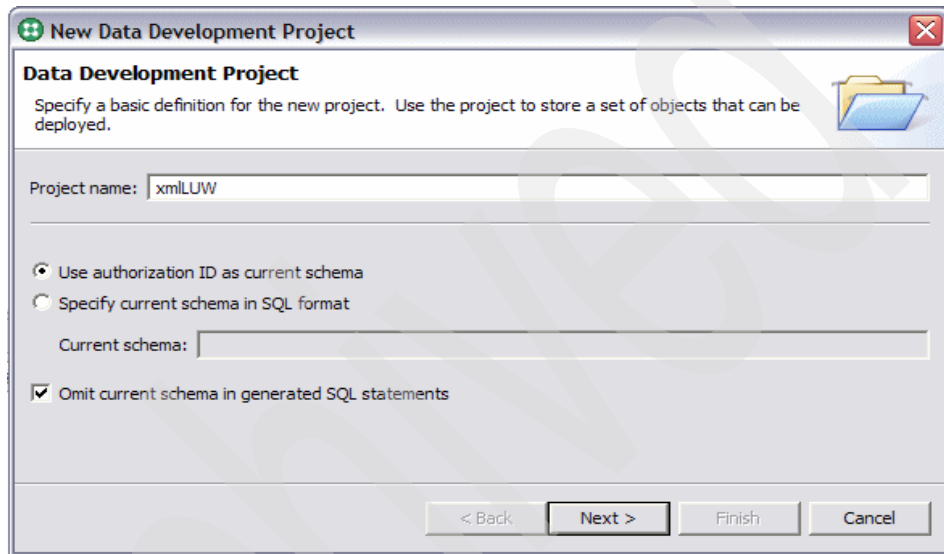


Figure 6-4 Creating a new data development project

4. Select a database connection:

After creating a new project (or choosing an existing project), the wizard proceeds to the Select Connection window. At this point, a new database connection can be created or an existing database connection can be chosen. See Figure 6-5. Click **Next**.

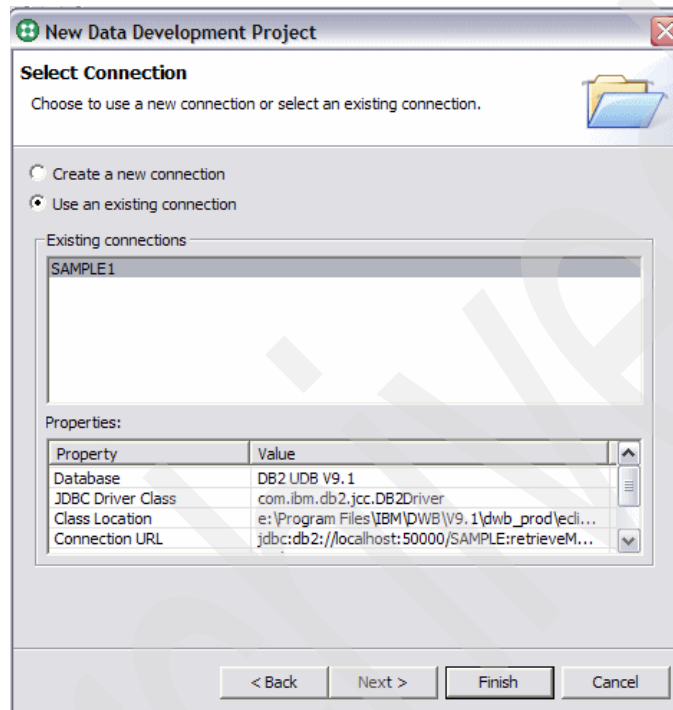


Figure 6-5 Select database connection

5. Specify the JDK™ home directory:

The Specify Routine Parameters window requests that the JDK home directory be specified (see Figure 6-6). Either accept the location displayed or browse to another location. Click **Finish**.

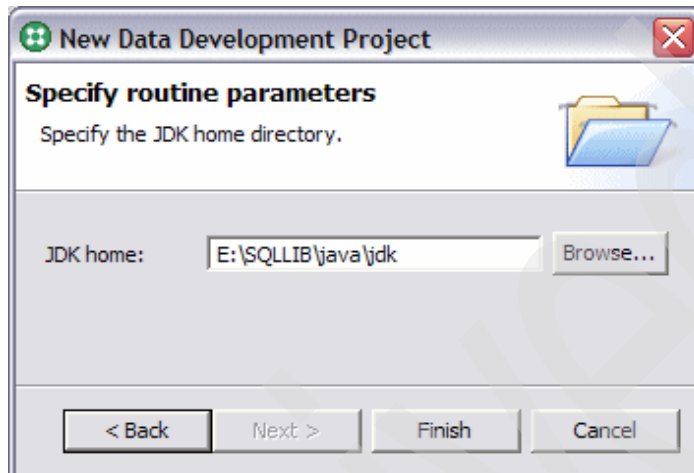


Figure 6-6 Specify the JDK home directory.

6. Specify a query name:

The New XML Query window opens, as shown in Figure 6-7. Specify a name for the query. In this example, the query is named xmlQuery1. Click **Next**.

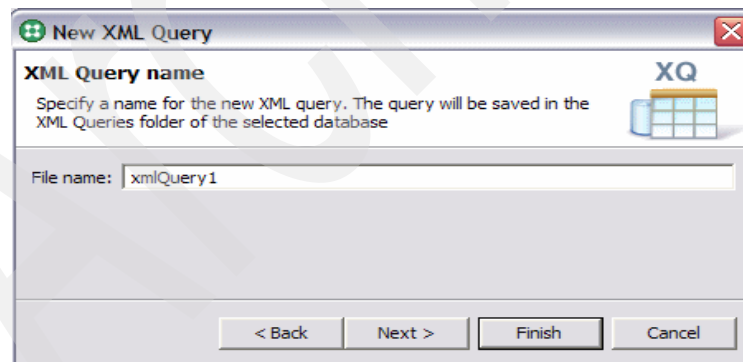


Figure 6-7 The New XML Query window; specifying a name for the query.

7. Add representative XML documents:

As shown in Figure 6-8, the New XML Query window will reopen to the Add representative XML documents window.

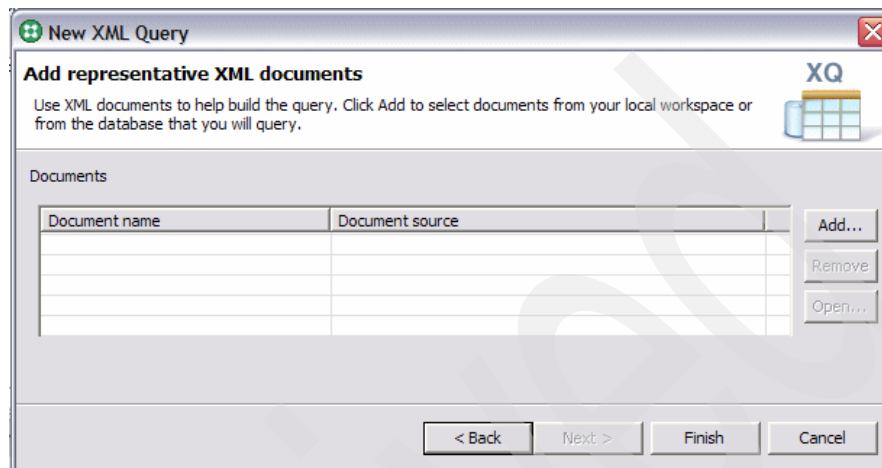


Figure 6-8 The Add representative XML documents window

To add a representative document from your local workspace, or from the database to be queried, select **ADD**. The following example, Figure 6-9, shows the Specify document location window after **ADD** has been selected. In our example, **Database** has been chosen as the location of the representative XML document.

Click **Next**.

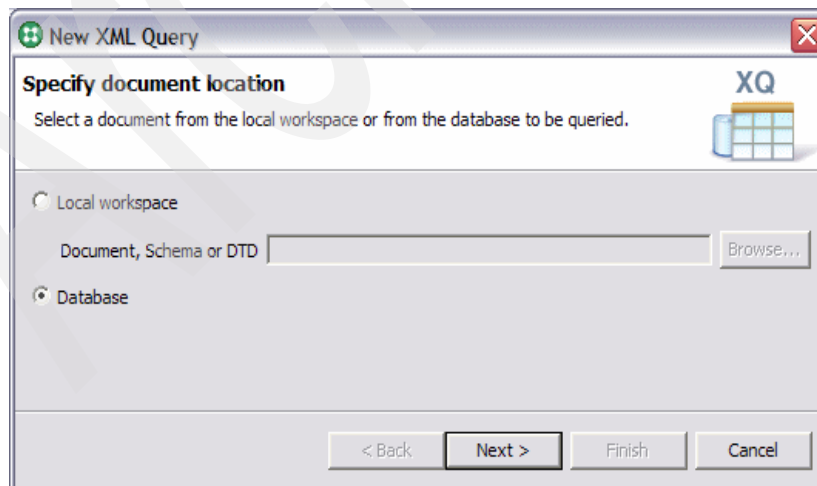


Figure 6-9 Selecting Database as the location of the representative XML document

8. Choose the XML document source:

The XML column or schema window opens (Figure 6-10) and the **INFO** column of the CUSTOMER table is chosen as the source containing the XML document to be queried. Click **Next**.

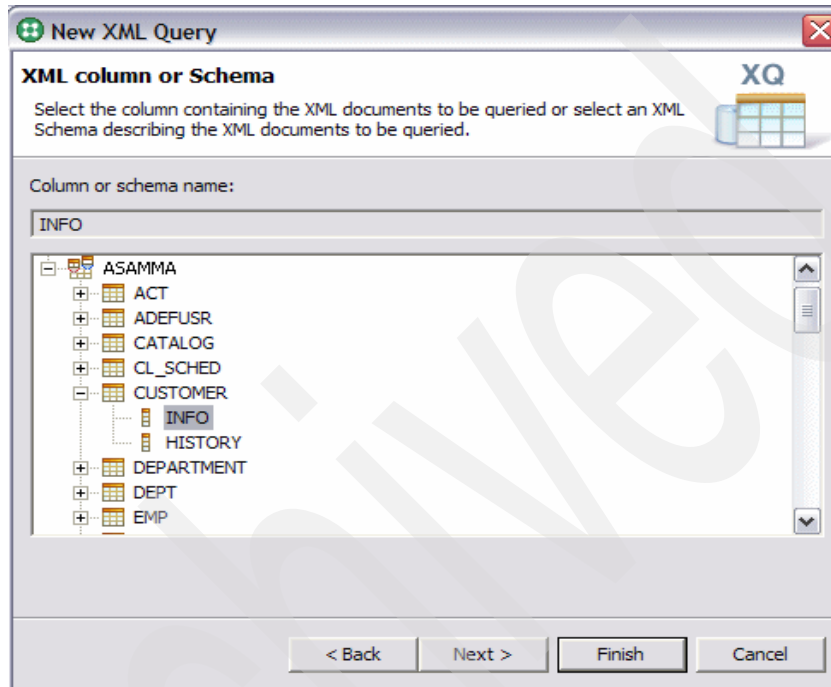


Figure 6-10 Choosing XML document source

In Figure 6-11, the New XML Query window reopens and displays the representative document that has been chosen in the previous step. Select the document, then click **Next**.

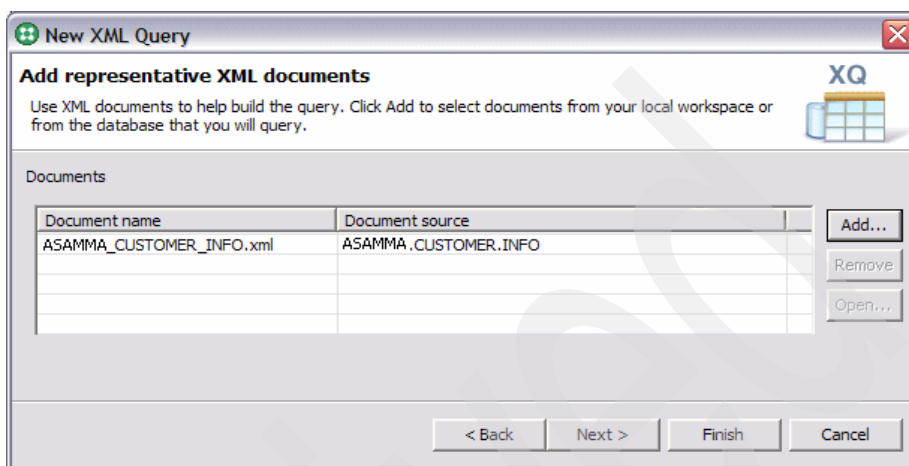


Figure 6-11 *CUSTOMER.INFO* column selected as the XML source

9. Associate the document with XML columns:

The Associate documents with XML columns window opens (Figure 6-12). Click **Finish**.

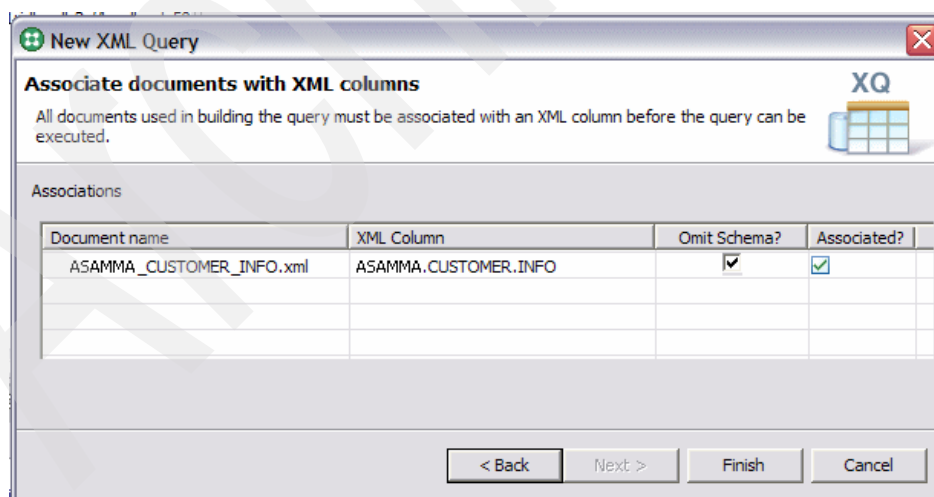


Figure 6-12 *The document is associated with an XML column*

A new query, xmlQuery1.xqm, is added to the Queries node and DWB opens to the XQuery Builder in Design View. This is the main view from which the query will be built. See Figure 6-13.

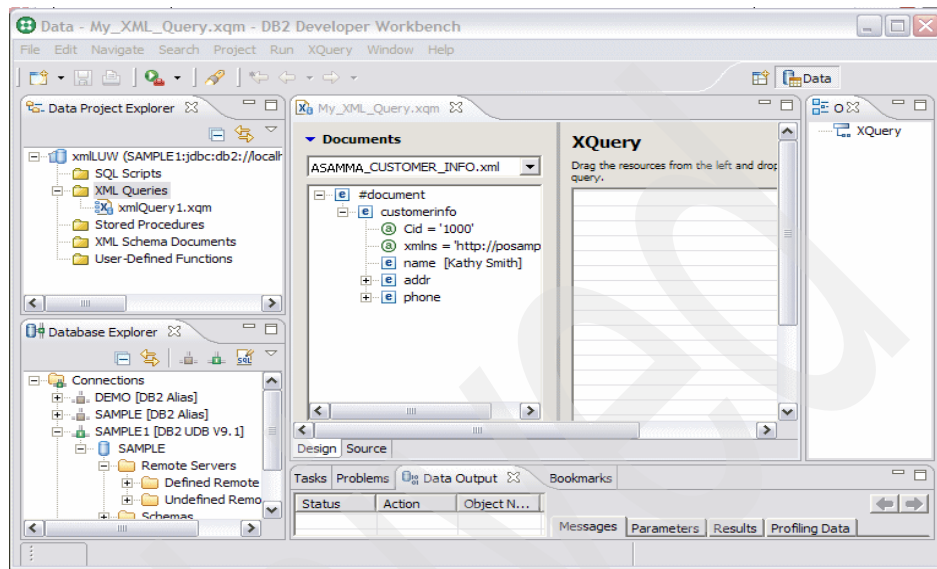


Figure 6-13 Developer Workbench with XQuery Builder open to Design view

Building and running a query in XQuery Builder

When the steps outlined in the previous section have been completed, the task of building a query through the XQuery builder can begin.

In the examples in this section, in the interest of clarity, only the portions of the DWB screen that are referenced by the examples are displayed.

Building an XQuery

Drag and drop the **customerinfo** node from the sample XML document tree to a row in the design grid.

The node name will appear in the grid and a drill in button (*arrow* highlighted by the red circle here) will be displayed at the end of the row (Figure 6-14).

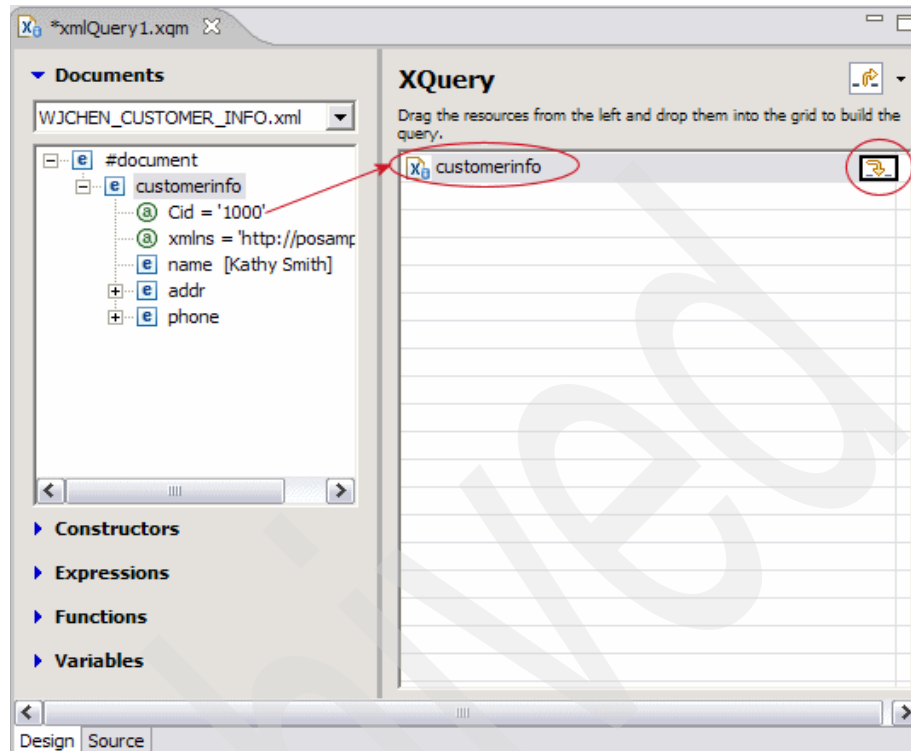


Figure 6-14 Drag and drop the customerinfo node to the design grid

The source code generated by the GUI can be viewed by selecting the **Source** tab in the Design view, see Figure 6-15.

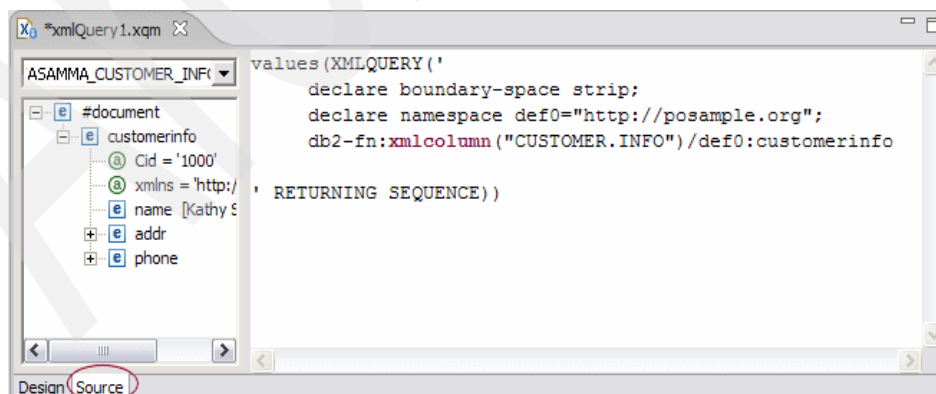



Figure 6-15 View generated source code from Source tab

FLWOR expression

You can create a more elaborate query by adding *predicates* to the search and ordering the returned elements.

To accomplish this, while in Design view, click the **Step into** button  at the end of the first row in the design grid. When you step into a query, the existing grid is replaced with five new grids representing the FOR, LET, WHERE, ORDERBY and RETURN parts of the FLWOR expression.

Add predicates to the search

To refine our query, we included some predicates. To achieve this, in our example, Figure 6-16, we have added or executed the following operations:

- ▶ Dragged and dropped the **Cid** attribute from the XML document tree onto the Operand1 column in the Where grid.
- ▶ Selected the = operator in the Operator column.
- ▶ Typed 1000 into the Operand2 column.
- ▶ Dragged and dropped the **name**, **addr**, and **phone** elements to the Return grid.

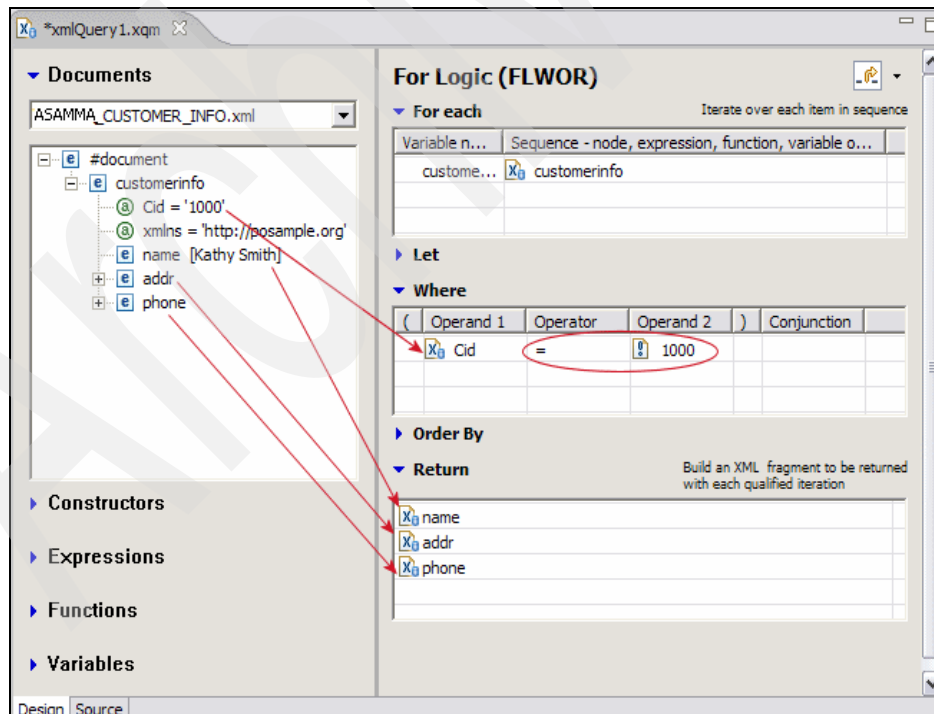


Figure 6-16 Adding predicates to the XQuery

When these changes are made in the GUI, the results can be viewed by selecting the **Source** tab. Example 6-1 shows the code from the Source tab that was generated by the GUI in the previous steps.

Example 6-1 Source code for the query created by XQuery Builder

```
values(XMLQUERY('
    declare boundary-space strip;
    declare namespace def0="http://posample.org";
    for $customerinfo0 in
db2-fn:xmlcolumn("CUSTOMER.INFO0")/def0:customerinfo
    where $customerinfo0/@Cid = 1000
    return
    (
        $customerinfo0/def0:name,
        $customerinfo0/def0:addr,
        $customerinfo0/def0:phone
    )
' RETURNING SEQUENCE))
```

In order to execute the query, select the **Run...** option in the **XQuery** menu, as shown in Figure 6-17.

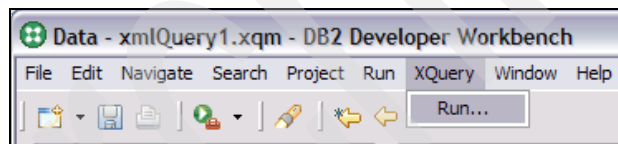


Figure 6-17 Executing the query by selecting Run

Figure 6-18 shows the result of the query execution.

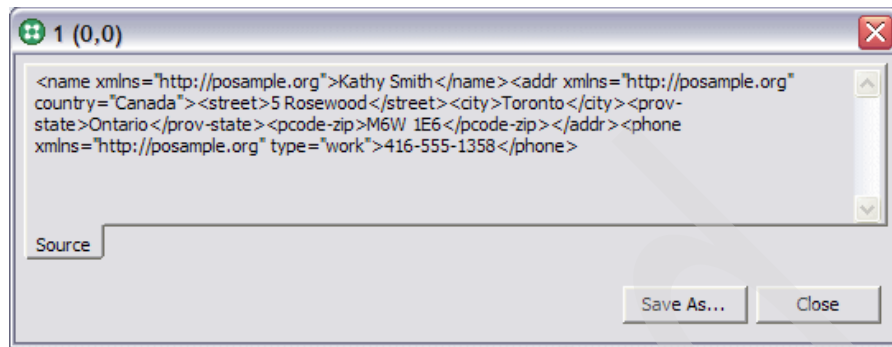


Figure 6-18 Results of the query execution

6.3 Accessing pureXML from application overview

This section provides an overview of accessing pureXML from CLI, C/C++, PHP, Java, and .NET. We discuss issues and concerns that are relevant to inserting, updating, and retrieving XML data using various application programming interfaces.

6.3.1 Application programming language support for XML

Any of the following languages can be used to write applications that involve XML data:

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

Applications written in any of these languages can store or retrieve XML data from DB2 database tables, or call stored procedures or user-defined functions with XML parameters. XML data can only be stored in a *Unicode* database. To create a Unicode database in DB2, the database must be created with the parameter CODESET set to UTF-8.

There are many options available when creating a DB2 database. Example 6-2 shows a simple DB2 command (showing only a minimum of options) to create a DB2 database with CODESET UTF-8.

```
CREATE DATABASE MYDB USING CODESET UTF-8 TERRITORY US
```

Note: Refer to the manual *Command Reference*, SC10-4226 for the complete syntax and options available for creating a DB2 database.

6.3.2 Considerations when updating and inserting XML data

When updating or inserting data into an XML column, the input to the XML column must be a well-formed XML document, as defined in the XML 1.0 specification¹. The application data types can be:

- ▶ XML
- ▶ Character
- ▶ Binary

When *inserting* data, we recommend that XML data be inserted from host variables, rather than literals. This is so that the DB2 database server can use the host variable data type to determine some of the encoding information.

XML data in an application is in its serialized string format. When you insert or update data into an XML column, it must be converted to its XML hierarchical format. This process is known as *XML parsing*.

XML parsing

XML parsing is the process of converting XML data from its serialized string format to its hierarchical format. Simply stated, XML parsing converts character or binary data and produces an XML value.

You can let the DB2 database manager perform parsing *implicitly*, or you can perform XML parsing *explicitly*.

Implicit XML parsing occurs:

- ▶ When you pass data to the database server using a host variable of type XML, or use a parameter marker of type XML. The database server does the parsing when it binds the value for the host variable or parameter marker for use in statement processing. You must use implicit parsing in this case.
- ▶ When you assign a host variable, parameter marker, or SQL expression with a string data type (character, graphic or binary) to an XML column in an INSERT, UPDATE, DELETE, or MERGE statement. The parsing occurs when the SQL compiler implicitly adds an XMLPARSE function to the statement.

¹ See www.w3.org/TR for information about the XML 1.0 specification. The *Extensible Markup Language (XML) fourth Edition (1.0)* is the latest recommendation as of the date of this book.

Example 6-3 demonstrates a case of implicit parsing. In this example, the source is an XML document from a column of type VARCHAR.

Example 6-3 An example of implicit parsing

```
/* 1) Assume table TABLE1 has been created with the following
definition: */
/* CREATE TABLE table1 (id INT, description VARCHAR(200)) */

/* 2) Assume TABLE1 has been populated as follows:*/
/* INSERT INTO table1 VALUES (22222, '<product xmlns =
"http://posample.org\" pid=\"80\"> <description><name> Plastic Casing
</name> <details> Green Color </details> <price> 7.89 </price> <weight>
6.23 </weight> </description></product>', 'Last Product')"
```

```
/* 3) Assume table po has been created with the following definition:
*/
/* CREATE TABLE po (poid BIGINT, porder XML) */

char stmt[500];
SQLRETURN cliRC = SQL_SUCCESS;

strcpy(stmt, "INSERT INTO po (poid, porder) "
"(SELECT id, description FROM table1 WHERE id = 22222)");

/* execute the statement */
cliRC = SQLExecDirect(hstmt, (SQLCHAR *)stmt, SQL_NTS);
STMT_HANDLE_CHECK(hstmt, hdbc,
cliRC);
```

Explicit parsing occurs:

- When the XMLPARSE function is invoked when inputting XML data. The XMLPARSE function takes a non-XML, character or binary data type as input. The result of the XMLPARSE function can be utilized in any context that accepts an XML data type, for example, it can be assigned to an XML column or used as a stored procedure parameter of type XML.

For embedded *dynamic* SQL applications, you must cast the parameter marker that represents the input document for XMLPARSE to the appropriate data type.

Example 6-4 illustrates casting the parameter marker to BLOB(1K) for the input *Document* parameter of the XMLPARSE function in a dynamic CLI application.

Example 6-4 Casting the parameter marker to BLOB using XMLPARSE function

```
char blobdata[500];
SQLRETURN cliRC = SQL_SUCCESS;
length = strlen(blobdata);

/* Assume table po has been created with the following definition: */
/* CREATE TABLE po (poid BIGINT, porder XML) */

strcpy(blobdata, "<product xmlns = \"http://posample.org\"
pid=\"10\"><description><name> Plastic Casing </name>\"
<details> Blue Color </details><price> 2.89 </price>\"
<weight> 0.23 </weight></description></product>");

strcpy(stmt, "INSERT INTO po (poid, porder) "
"VALUES (323, XMLPARSE(DOCUMENT CAST(? as BLOB(1K))))");

/* prepare the statement */
cliRC = SQLPrepare(hstmt, (SQLCHAR *)stmt, SQL_NTS);

/* bind Parameter to the Insert statement */
cliRC = SQLBindParameter(hstmt,
                        1,
                        SQL_PARAM_INPUT,
                        SQL_C_BINARY,
                        SQL_BLOB,
                        length,
                        0,
                        &blobdata,
                        length,
                        NULL);

/* execute the statement */
cliRC=SQLExecute(hstmt);
```

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

Example 6-5 illustrates a static embedded SQL application; in this example the host variable argument of the XMLPARSE function is declared as BLOB.

Example 6-5 A static embedded SQL statement calling XMLPARSE

```
EXEC SQL DECLARE SECTION;
    char xmldata[2000];
    char parse_option[30];
    short nullind = 0;
    static SQL TYPE IS BLOB(1k) hv_blob2 = SQL_BLOB_INIT("<init> a
</init>");
EXEC SQL END DECLARE SECTION;

/* Assume table PO has been created with the following definition: */
/* CREATE table PO (poid BIGINT, porder XML) */

strcpy(xmldata, "<product xmlns = \"http://posample.org\"
pid=\"10\"><description><name> Plastic Casing </name>\"
\"<details> Blue Color </details><price> 2.89 </price>\"
\"<weight> 0.23 </weight></description></product>");

strcpy(hv_blob2.data, xmldata);

EXEC SQL UPDATE PO SET porder = XMLPARSE(
    DOCUMENT :hv_blob2 STRIP WHITESPACE) WHERE POID = 1612;
```

XML parsing and whitespace handling

During implicit or explicit XML parsing, you can control the preservation or stripping of boundary whitespace characters when you store the data in the database.

According to the XML standard, whitespace is space characters (U+0020), carriage returns (U+000D), line feeds (U+000A), or tabs (U+0009) that are in the document to improve readability. When any of these characters appear as part of a text string, they are not considered to be whitespace.

Boundary whitespace is whitespace characters that appear between elements. For example, in the following document the spaces between <customerinfo> and <name> and between </customerinfo> and </name> are considered boundary whitespace.

```
<customerinfo> <name> </name> </customerinfo>
```

With *explicit* invocation of XMLPARSE, you use the STRIP WHITESPACE or PRESERVE WHITESPACE option to control preservation of boundary whitespace. The default is stripping of boundary whitespace.

With *implicit* XML parsing:

- ▶ If the input data type is not an XML type or is not cast to an XML data type, the DB2 database manager *always* strips whitespace.
- ▶ If the input data type is an XML data type, you can use the CURRENT IMPLICIT XMLPARSE OPTION special register to control preservation of boundary whitespace. You can set this special register to STRIP WHITESPACE or PRESERVE WHITESPACE. The default is stripping of boundary whitespace. Note that this special register only applies for nonvalidating XML parsing.
- ▶ If the input data type is non-XML, but is CAST as XML (either explicitly or as an ambiguous parameter marker) then implicit XML parse applies and the CURRENT IMPLICIT XMLPARSE OPTION special register will do as well.

Example 6-6 illustrate setting the CURRENT IMPLICIT XMLPARSE OPTION special register in various application settings.

Example 6-6 Setting the Current IMPLICIT XMLPARSE OPTION special register

CLI:

```
strcpy((char *)stmt, "SET CURRENT IMPLICIT XMLPARSE OPTION =  
'PRESERVE WHITESPACE'");  
  
rc = SQLExecDirect(hstmt, stmt, SQL_NTS);
```

Embedded SQL:

```
EXEC SQL BEGIN DECLARE SECTION;  
char parse_option[30];  
EXEC SQL END DECLARE SECTION;  
strcpy(parse_option, "preserve whitespace");  
  
/* SET the register with the option PRESERVE WHITESPACE */  
EXEC SQL SET CURRENT IMPLICIT XMLPARSE OPTION = :parse_option;
```

JAVA (SQLJ):

```
String parse_option = "preserve whitespace";  
#sql {  
    SET CURRENT IMPLICIT XMLPARSE OPTION = :parse_option};
```

Note: The CurrentImplicitXMLParseOption can also be set in the db2cli.ini initialization file. Refer to Current Implicit XML Parse Option in the *Call Level Interface Guide and Reference, Volume 1*, SC10-4224, for details.

XML validation

XML validation is the process of determining whether the structure, content, and data types of an XML document are valid. XML validation also adds type annotations to element nodes, attribute nodes, and atomic values, and strips off ignorable whitespace in the XML document. Validation is optional, but highly recommended.

The XMLVALIDATE function is used to validate an XML document. It is commonly used when *inserting* or *updating* an XML document in a DB2 database. XMLVALIDATE can also be invoked on an XML document that is *not* in a database. Before you can invoke XMLVALIDATE, all schema documents that make up an XML schema must be registered in the built-in XML schema repository (XSR). An XML schema provides the rules for a valid XML document.

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

- ▶ xmlvalidate(? ACCORDING TO XMLSCHEMA ID schema name)
- ▶ xmlvalidate(?)
- ▶ xmlvalidate(:hvxml ACCORDING TO XMLSCHEMA ID schema name)
- ▶ xmlvalidate(:hvxml)
- ▶ xmlvalidate(cast(? as xml) ACCORDING TO XMLSCHEMA ID schema name)
- ▶ xmlvalidate(cast(? as xml))

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

Important: The insert or update operation on which the XMLVALIDATE was specified will only occur *if the validation succeeds*.

Encoding considerations for input of XML data to a database

XML data can be *internally* or *externally* encoded. When the encoding of XML data is derived from the data itself, it is known as *internally* encoded data. If the data is derived from external sources, it is known as *externally* encoded data.

- ▶ XML data that is sent to the database server as binary data is treated as internally encoded data.
- ▶ XML data that is sent to the database server as character data is treated as externally encoded data.

External encoding on input for Java applications is *always* Unicode encoding.

Externally encoded data can have internal encoding. That is, the data might be sent to the database server as character data, but the data contains encoding information. The database server handles incompatibilities between internal and external encoding as follows:

- ▶ If the database server is DB2 Database for Linux, UNIX, and Windows, the database server generates an error if the external and internal encoding are incompatible, unless the external and internal encoding are Unicode. If the external and internal encoding are Unicode, the database server ignores the internal encoding. If internal encoding is Unicode, but external is non-Unicode, the mismatch will be flagged.
- ▶ If the database server is DB2 for z/OS, the database server ignores the internal encoding.

Data in XML columns is stored in UTF-8 encoding. The database server handles conversion of the data from its internal or external encoding to UTF-8.

When you store XML data in a DB2 table, observe the following rules:

- ▶ If the internal and external encoding are not Unicode encoding, for externally encoded XML data (data that is sent to the database server using character data types), any internally encoded declaration must match the external encoding. Otherwise, an error occurs, and the database manager rejects the document.
- ▶ If the external encoding and the internal encoding are Unicode encoding, and the encoding schemes do not match, the DB2 database server ignores the internal encoding.
- ▶ For internally encoded XML data (data that is sent to the database server using binary data types), the application must ensure that the data contains accurate encoding information.

6.3.3 Considerations when retrieving XML data

When an application retrieves data from XML columns, the DB2 database server converts the data from the XML hierarchical format to the XML serialized string format. In addition, the database server might have to convert the output data from UTF-8 to the application encoding.

For implicit and explicit XML serialization, implicit is much safer as the encoding is done automatically by the application interface. Explicit XMLSERIALIZE is subject to extra codepage conversion.

When you retrieve XML data, you have to be aware of the effect of code page conversion on data loss. Data loss can occur when characters in the source code page *cannot be represented* in the target code page.

An application program can retrieve an entire document or a fragment of a document from an XML column. However, you can store only an *entire* document in an XML column.

When you fetch an entire XML *document*, you retrieve the document into an application variable.

When you retrieve an XML *sequence*, you have several choices:

- ▶ Execute an XQuery expression directly.

To execute an XQuery expression in an application, you add the string “XQUERY” to the XQuery expression, and dynamically execute the resulting string.

When you execute an XQuery expression directly, the DB2 database server returns the sequence that is the result of the XQuery statement as a result table. Each row in the result table is an item in the sequence.

- ▶ Execute an XQuery expression within an SQL FETCH or single-row SELECT INTO operation by calling the XMLQUERY or XMLTABLE built-in functions and passing an XQuery expression as an argument. *XMLQUERY* is a scalar function that returns the entire sequence in an application variable. *XMLTABLE* is a table function that returns each item in the sequence as a row of the result table. The columns in the result table are values from the retrieved sequence item. An illustration of this is shown in Example 6-7.

This technique can be used with static or dynamic SQL and *any* application programming language.

Example 6-7 Executing an XQuery expression within an SQL FETCH

```
select deptID,xmlquery('for $d in $doc/dept
                        where $d/@bldg = 101
                        return $d/name' passing doc as "doc")
from dept
where deptID <> "PR27";
```

Parameter markers and host variables

The following rules/restrictions apply to employing parameter markers or host variables in an XQuery expression:

- ▶ Parameter markers or host variables *cannot* be specified anywhere in an XQuery expression, including within the SQL specified in an XQuery expression.
- ▶ You *cannot* specify a parameter marker or host variable in the XQuery expression, even within the fullselect.

- In order to pass application values to XQuery expressions, use the SQL/XML functions XMLQUERY and XMLTABLE. The PASSING clause of these functions allows you to use application values during the evaluation of the XQuery expression.

Example 6-8 illustrates passing an application value to an XQuery expression, in a Java application, using the SQL/XML function XMLQUERY.

Example 6-8 Passing an application value to an XQuery expression in a Java application

```
// The table CUSTOMER exists with the following definition:
// CREATE TABLE customer (cid BIGINT, info XML, history XML)

private static int cid=1002;
...
Statement stmt = con.createStatement();
String query="select xmlquery('declare default element namespace
'http://posample.org\";"+
" for $customer in $cust/customerinfo"+
" where ($customer/@Cid gt $id)"+
" return <customer id=\"{"$customer/@Cid}\">"+
" {"$customer/name} {"$customer/addr} </customer>'+
" passing by ref customer.info as \"cust\", cast(? as integer) as
'id\");"+
" from customer";

// Prepare the statement
PreparedStatement pstmt = con.prepareStatement(query);

// Set the value for the parameter marker
pstmt.setInt(1,cid);
ResultSet rs = pstmt.executeQuery();
```

Encoding considerations for retrieval of XML data from a database

When you retrieve XML data from a DB2 table, you have to avoid data loss and truncation.

- Data loss can occur when characters in the source data cannot be represented in the encoding of the target data.
Data loss is less of a problem for Java and .NET applications than for other types of applications because Java and .NET string data types use Unicode UTF-16 or UCS2 encoding.

- Truncation can occur when conversion to the target data type results in expansion of the data.

Truncation is possible because expansion can occur when UTF-8 characters are converted to UTF-16 or UCS-2 encoding.

Note: Refer to the *XML Guide*, SC10-4254, chapter 8, *XML CODING* for complete details regarding XML coding considerations.

6.4 DB2 application development with CLI and ODBC

DB2 Call Level Interface (DB2 CLI) is the IBM callable SQL interface to the DB2 family of database servers. It is a 'C' and 'C++' application programming interface for relational database access that uses function calls to pass dynamic SQL statements as function arguments. It is an alternative to embedded dynamic SQL, but unlike embedded SQL, DB2 CLI does not require host variables or a precompiler.

DB2 CLI is based on the Microsoft** Open Database Connectivity** (ODBC) specification, and the International Standard for SQL/CLI. These specifications were chosen as the basis for the DB2 Call Level Interface in an effort to follow industry standards and to provide a shorter learning curve for those application programmers already familiar with either of these database interfaces. In addition, some DB2 specific extensions have been added to help the application programmer specifically exploit DB2 features.

The DB2 CLI driver also acts as an ODBC driver when loaded by an ODBC driver manager. It conforms to ODBC 3.51.

Note: For complete information regarding CLI requirements, configuration, programming, and other relevant topics, refer to these manuals:

- *Call Level Interface Guide and Reference, Volume 1*, SC10-4224
- *Call Level Interface Guide and Reference, Volume 2*, SC10-4225

6.4.1 Setting up the CLI environment

Runtime support for DB2 CLI applications is contained in all DB2 clients. Support for building and running DB2 CLI applications is contained in the DB2 Client. This section describes the general setup required for DB2 CLI runtime support.

Note: Before you set up your CLI environment, ensure that you have set up the application development environment. Refer to the *Call Level Interface Guide and Reference, Volume 1*, SC10-4224 for an overview of the CLI application development environment setup.

In order for a DB2 CLI application to successfully access a DB2 database:

1. Ensure that the DB2 CLI/ODBC driver was installed during the DB2 client install.
2. Catalog the DB2 database and node if the database is being accessed from a remote client. On the Windows platform, you can use the CLI/ODBC Settings GUI to catalog the DB2 database.
3. Optional: Explicitly bind the DB2 CLI/ODBC bind files to the database with the command:

```
db2 bind ~/sqllib/bnd/@db2cli.lst blocking all sqlerror continue \
messages cli.msg grant public
```

On the Windows platform, you can use the CLI/ODBC Settings GUI to bind the DB2 CLI/ODBC bind files to the database.

4. Optional: Change the DB2 CLI/ODBC configuration keywords by editing the db2cli.ini file, located in the sqllib directory on Windows, and in the sqllib/cfg directory on UNIX platforms. On the Windows platform, you can use the CLI/ODBC Settings GUI to set the DB2 CLI/ODBC configuration keywords.

Once you have completed the foregoing steps, proceed to setting up your Windows CLI environment, or setting up your UNIX ODBC environment if you are running ODBC applications on UNIX.

6.4.2 Building CLI applications

DB2 provides build scripts for compiling and linking CLI programs. For UNIX or Windows, these scripts, along with sample programs that can be built with these files, are located in the following directories:

- ▶ UNIX: sqllib/samples/cli directory
- ▶ Windows: sqllib\samples\cli directory

In addition to the CLI samples that can be found in the directories above, there are specific XML examples that can be found in the following locations:

- ▶ On UNIX:
 - sqllib/samples/xml/cli
 - sqllib/samples/xml/xquery/cli

- ▶ On Windows:
 - sqllib\samples\xml\cli
 - sqllib\samples\xml\xquery\cli

The build scripts, bldapp (on UNIX) or bldapp.bat (Windows), contain the commands to build a DB2 CLI application. It takes up to four parameters, represented inside the UNIX script file by the variables: \$1, \$2, \$3, and \$4, or inside the Windows file by the variables: %1, %2, %3, and %4.

- ▶ Parameter \$1 (%1): This parameter specifies the name of your source file. This is the only *required* parameter, and the only one required for CLI applications that do not contain embedded SQL.

Building embedded SQL programs requires a connection to the database, so three optional parameters are also provided.

- ▶ Parameter \$2 (%2): This parameter specifies the name of the database to which you want to connect.
- ▶ Parameter \$3 (%3): This parameter specifies the user ID for the database.
- ▶ Parameter \$4 (%4): This parameter specifies the password.

If the program contains embedded SQL, with a .sql or the .sql extension, then the embprep (UNIX) or the embprep.bat (Windows) script is called to precompile the program, producing a program file with a .c or a .cxx extension.

To build the sample program tbinfo from the source file tbinfo.c, enter:

```
bldapp tbinfo
```

The result is an executable file, tbinfo. You can run the executable file by entering the executable name:

```
tbinfo
```

In addition to the sample build scripts supplied by DB2, it is possible to build all of the applications by executing the makefile that is found in the corresponding directories.

On UNIX, the makefile is found in these directories:

- ▶ sqllib/samples/cli
- ▶ sqllib/samples/xml/cli
- ▶ sqllib/samples/xml/xquery/cli

On Windows, the makefile is found in these directories:

- ▶ sqllib\samples\cli
- ▶ sqllib\samples\xml\cli
- ▶ sqllib\samples\xml\xquery\cli

Before running the makefile, modify the makefile to reflect your environment:

- ▶ set UID (user ID to access the sample database)
- ▶ set PWD (password to access the sample database)

To build the file or files, execute the appropriate command for your environment in your working directory, for example:

- ▶ UNIX
`make some_parameter`
- ▶ Windows
`nmake some_parameter`

Where *some_parameter* corresponds to one of the parameters specified below:

- ▶ **make** (or **nmake**) *<app_name>*
*/*Builds the program designated by <app_name>*/*
- ▶ **make** (or **nmake**) **all**
/ Builds all supplied sample programs */*
- ▶ **make** (or **nmake**) **srv**
*/*Builds sample that can only be run on the server, (stored procedure)*/*
- ▶ **make** (or **nmake**) **all_client**
/ Builds all client samples (all programs in the 'call_rtn' and 'client_run' categories). */*
- ▶ **make** (or **nmake**) **call_rtn**
/ Builds client programs that call stored procedure */*
- ▶ **make** (or **nmake**) **client_run**
/ Builds all programs that run completely on the client (not ones that call stored procedure)*/*
- ▶ **make** (or **nmake**) **clean**
/ Erases all intermediate files produced in the build process */*
- ▶ **make** (or **nmake**) **cleanall**
/ Erases all files produced in the build process (all files except the original source files)*/*

Note: To ensure a successful build of the sample applications, we suggest that you:

- ▶ Read the Prerequisites section of the header in the sample file and follow the directions and suggestions before building or running the sample.
- ▶ Make sure that a compatible make. or nmake, executable program is resident on your system in a directory included in your PATH variable.

6.4.3 XML data handling in CLI applications

DB2 CLI applications can retrieve and store XML data using the SQL_XML data type. This data type corresponds to the native XML data type of the DB2 database, which is used to define columns that store well-formed XML documents. The SQL_XML type can be bound to the following C types:

- ▶ SQL_C_BINARY
- ▶ SQL_C_CHAR
- ▶ SQL_C_WCHAR
- ▶ SQL_C_DBCHAR

Note: Using the default SQL_C_BINARY instead of character types is recommended, but not required, to avoid *possible* data loss or corruption resulting from code page conversion when character types are used.

Inserts and updates to XML columns in CLI applications

When you update or insert data into XML columns, the input data must be in serialized string format.

For XML data, use the function SQLBindParameter() to bind parameter markers to input data buffers. When you bind a data buffer that contains XML data as SQL_C_BINARY, DB2 CLI processes the XML data as *internally* encoded data. This is the preferred method because it avoids the overhead and potential data loss of character conversion when character types are used.

If you want the database server to implicitly parse the data before storing it in an XML column, the argument *ParameterType* in SQLBindParameter() should be specified as SQL_XML.

Implicit parsing is recommended, because explicit parsing of a character type with XMLPARSE can introduce encoding issues. Note that internally encoded data might require an XML declaration if encoded in anything other than UTF-8.

Example 6-9 shows an INSERT of XML data into an XML column. In this example, the data buffer is bound with the recommended SQL_C_BINARY type, and the *ParameterType* for SQLBindParameter() is SQL_XML. Because SQL_C_BINARY is used, the data must be internally encoded in order to be interpreted correctly. In this example the internal encoding is declared as ISO-8859-1.

Example 6-9 Inserting XML data with recommended SQL_C_BINARY type binding

```
char xmldata[500];
int length;
SQLRETURN cliRC = SQL_SUCCESS;

/* Assume the table PO has been created with the following definition:
*/
/* CREATE table PO (poid BIGINT, porder XML) */

    strcpy(xmldata, "<?xml='\"1.0\"' encoding='\"ISO-8859-1\"'?><product
xmlns = \"http://posample.org\" pid='\"10\"'><description>\"
<name> Plastic Casing </name>\"
<details> Blue Color </details>\"
<price> 2.89 </price>\"
<weight> 0.23 </weight>\"
</description></product>");

length = strlen(xmldata);

/* inserting when source is from host variable of type XML */
    strcpy(stmt, "INSERT INTO PO (poid, porder) "
                "VALUES (8956, ?)");

/* prepare the statement */
    cliRC = SQLPrepare(hstmt, (SQLCHAR *)stmt, SQL_NTS);

/* bind Parameter to the Insert statement */
    cliRC = SQLBindParameter(hstmt,
                            1,
                            SQL_PARAM_INPUT,
                            SQL_C_BINARY,
                            SQL_XML,
                            length,
                            0,
                            &xmldata,
                            length,
```

```
NULL);
```

```
cliRC = SQLExecute(hstmt);
```

Example 6-10 demonstrates an INSERT of XML data into an XML column. In this example, the data buffer is bound with the SQL_C_CHAR type. The function XMLCAST is used to typecast the character data to an XML data type.

Example 6-10 Using XMLCAST to typecast data into an XML column

```
char xmldata[500];
SQLRETURN cliRC = SQL_SUCCESS;

/* Assume the table PO exists with the following definition: */
/* CREATE table po (poid BIGINT, porder XML) */

strcpy(xmldata, "<product xmlns = \"http://posample.org\"
pid=\"10\"><description>
    <name> Plastic Casing </name>\"
    <details> Blue Color </details>\"
    <price> 2.89 </price>\"
    <weight> 0.23 </weight>\";
    </description></product>");

strcpy(stmt, "INSERT INTO PO (poid, porder) \"
VALUES(125, XMLCAST(? as XML))");
cliRC = SQLPrepare(hstmt, (SQLCHAR *)stmt, SQL_NTS);

/* bind Parameter to the Insert statement */
cliRC = SQLBindParameter(hstmt,
    1,
    SQL_PARAM_INPUT,
    SQL_C_CHAR,
    SQL_CHAR,
    500,
    0,
    &xmldata,
    500,
    NULL);

cliRC = SQLExecute(hstmt);
```

The code segment in Example 6-11 illustrates binding a parameter marker for an INSERT operation when the source is a variable of Type XML. This example also demonstrates *implicit parsing*.

Example 6-11 An INSERT with implicit parsing

```
int length;
int rc = 0;
char xmldata[500];
SQLRETURN cliRC = SQL_SUCCESS;

strcpy(xmldata, "<?xml version='1.0' encoding='ISO-8859-1'>
?><product xmlns = \"http://posample.org\"
pid=\"10\"><description>
<name> Plastic Casing </name>
<details> Blue Color </details>
<price> 2.89 </price>
<weight> 0.23 </weight>
</description></product>");

length = strlen(xmldata);

/* Assume the table P0 exists with the following definition: */
/* CREATE table po (poid BIGINT, porder XML) */

strcpy(stmt, "INSERT INTO P0 (poid, porder) "
"VALUES (8956, ?)");
cliRC = SQLBindParameter(hstmt,
1,
SQL_PARAM_INPUT,
SQL_C_CHAR,
SQL_XML,
length,
0,
&xmldata,
length,
NULL);

cliRC = SQLExecute(hstmt);
```

Example 6-12 illustrates performing an INSERT when the source is an XML document from a column of type VARCHAR. In this case, the description column is *explicitly* parsed because an SQL expression with a string data type is assigned to an XML column.

Example 6-12 INSERT an XML document from VARCHAR column with explicit parsing

```
/* 1) Assume table TABLE1 has been created with the following
definition: */
/* CREATE TABLE table1 (id INT, description VARCHAR(500)) */

/* 2) Assume TABLE1 has been populated as follows:*/
/* INSERT INTO table1 VALUES (22222, '<product xmlns =
"http://posample.org\" pid=\"80\"> <description><name> Plastic Casing
</name> <details> Green Color </details> <price> 7.89 </price> <weight>
6.23 </weight> </description></product>', 'Last Product')"

/* 3) Assume table po has been created with the following definition:
*/
/* CREATE TABLE po (poid BIGINT, porder XML) */

char stmt[500];
SQLRETURN cliRC = SQL_SUCCESS;

strcpy(stmt, "INSERT INTO po (poid, porder) "
"(SELECT id, XMLPARSE(DOCUMENT description) FROM table1 WHERE id =
22222)");

/* execute the statement */
cliRC = SQLExecDirect(hstmt, (SQLCHAR *)stmt, SQL_NTS);
STMT_HANDLE_CHECK(hstmt, hdbc,
cliRC);
```

Retrieving data from XML columns in a CLI application

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

For XML data, as with any data returned from a CLI application, the function SQLBindCol() is used to bind the columns of a query result set to application variables. The data types of the application variables can be specified as:

- ▶ SQL_C_BINARY
- ▶ SQL_C_CHAR
- ▶ SQL_C_DBCHAR or
- ▶ SQL_C_WCHAR

When retrieving a result set from an XML column, we recommend that you bind your application variable to the SQL_C_BINARY type. Binding to character types can result in possible data loss resulting from code page conversion. Data loss can occur when characters in the source code page cannot be represented in the target code page. Binding your variable to the SQL_C_BINARY C type avoids these issues.

XML data is returned to the application as internally encoded data. DB2 CLI determines the encoding of the data as follows:

- ▶ If the C type is SQL_C_BINARY, the data is returned in the UTF-8 encoding scheme.
- ▶ If the C type is SQL_C_CHAR or SQL_C_DBCHAR:
 - If the C type is SQL_C_CHAR, the data is returned in the application character code page encoding scheme.
 - If the C type is SQL_C_DBCHAR, the data is returned in the application graphic code page encoding scheme.
- ▶ If the C type is SQL_C_WCHAR, the data is returned in the UCS-2 encoding scheme.

When an XML value is retrieved into an application data buffer, the DB2 server performs an *implicit* serialization on the XML value to convert it from its stored hierarchical form to the serialized string form. For character typed buffers, the XML value is implicitly serialized to the application code page associated with the character type.

By default, an XML declaration is included in the *output* serialized string. This default behavior can be changed by setting the Attribute and ValuePtr arguments of SQLSetStmtAttr(), respectively, to:

- ▶ SQL_ATTR_XML_DECLARATION
- ▶ SQL_XML_DECLARATION_NONE

For further information about CLI connection attributes, refer to the manual: *CLI Guide and Reference, volume 2*.

The default behavior for including an XML declaration in the output serialized string can also be altered by changing XMLDeclaration in the CLI/ODBC configuration keyword in the db2cli.ini file. Refer to the manual *CLI Guide and Reference, volume 1*, for more information.

Example 6-13 on page 288 illustrates setting the SQL_ATTR_XML_DECLARATION attribute in the SQLSetStmtAttr() function.

Example 6-13 Setting the SQL_ATTR_XML_DECLARATION attribute

```
int rc = 0;
rc=SQLSetStmtAttr(hdbc, SQL_ATTR_XML_DECLARATION,
(SQLPOINTER)SQL_XML_DECLARATION_NONE, SQL_NTS);
```

The code segment in Example 6-14 illustrates binding the column of a result set to an application variable declared as a character data type SQL_C_CHAR. This example also shows an XQuery that is *not* preceded by the keyword XQUERY. As required, the SQL_ATTR_XQUERY_STATEMENT attribute of the SQLSetStmtAttr() function has been set to SQL_TRUE, indicating that the statement is an XQUERY.

Example 6-14 Binding the column of a result set to a character data type

```
SQLRETURN cliRC = SQL_SUCCESS;
int rc = 0;
SQLHANDLE hstmt; /* statement handle */
SQLVARCHAR xmldata[3000];

/* The table Customer exists with the following definition: */
/* CREATE table CUSTOMER ( cid BIGINT, info XML, history XML) */

/* query to be executed */
SQLCHAR *stmt = (SQLCHAR *)"declare default element namespace
'http://posample.org';"
"for $custinfo in db2-fn:xmlcolumn('CUSTOMER.INFO')"
"/customerinfo[addr/@country=\"Canada\"]"
" order by $custinfo/name"
" return $custinfo";

cliRC = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

/* Set the attribute SQL_ATTR_XQUERY_STATEMENT to indicate that the
query is an XQuery */
rc = SQLSetStmtAttr(hstmt, SQL_ATTR_XQUERY_STATEMENT,
(SQLPOINTER)SQL_TRUE, SQL_NTS);

if (rc != 0)
{
    return rc;
}

cliRC = SQLExecDirect(hstmt, stmt, SQL_NTS);

/* bind column 1 to variable */
```

```

    cliRC = SQLBindCol(hstmt, 1, SQL_C_CHAR, &xmldata, 1000, NULL);
    /* fetch each row and display */
    cliRC = SQLFetch(hstmt);
    ...

```

The code segment in Example 6-15 illustrates a query that binds the result of an SQL/XML query to an application variable bound with an **SQL_C_BINARY** data type.

*Example 6-15 An SQL/XML query with the result bound to a column of **SQL_C_BINARY** data type*

```

...
char xmlBuffer[10240];
/* xmlBuffer is used to hold the retrieved XML document */
integer length;

/* Assume a table named dept has been created with the definition */
/* CREATE TABLE dept (id CHAR(8), deptdoc XML) */

length = sizeof (xmlBuffer);
SQLExecute (hStmt, "SELECT deptdoc FROM dept WHERE id='001'", SQL_NTS);
SQLBindCol (hStmt, 1, SQL_C_BINARY, xmlBuffer, &length, NULL);
SQLFetch (hStmt);
SQLCloseCursor (hStmt);
// xmlBuffer now contains a valid XML document encoded in UTF-8
...

```

6.4.4 Embedded SQL applications: overview

Despite differences between host languages, embedded SQL applications (C/C++, COBOL, FORTRAN and REXX) are all made up of three main elements that are required to set up and execute an SQL statement:

- ▶ A **DECLARE SECTION** for declaring host variables:
 - The declaration of the **SQLCA** structure, which does not have to be in the **DECLARE** section
- ▶ The main body of the application, the setup, and execution of SQL statements
- ▶ Placements of logic that either commits, or rolls back, the changes made by the SQL statements

For each host language, there are differences between the general guidelines that apply to all languages, and rules that are specific to individual languages.

This section focuses on various aspects of embedded SQL application programming, specifically in relation to XML. All of the specifics, and nuances, of developing embedding SQL applications are beyond the scope of this document.

Note: For a complete understanding of application development using embedded SQL, refer to the manual *Developing Embedded SQL Applications*, SC10-4232.

6.5 Building applications in C or C++

DB2 provides build scripts for compiling and linking embedded SQL and DB2 administrative API programs in C or C++, along with sample programs that can be built with these files.

On UNIX platforms, the build file is `bldapp`, and it is found in these directories:

- ▶ `sqllib/samples/c` for C applications
- ▶ `sqllib/samples/cpp` for in C++ applications

On Windows, the build file is `bldapp.bat`, and it is found in these directories:

- ▶ `sqllib\samples\c` for C applications
- ▶ `sqllib\samples\cpp` for C++ applications

In addition to the embedded SQL samples that can be found in the foregoing directories, there are specific XML examples that can be found in the following locations:

- ▶ On UNIX:
 - `sqllib/samples/xml/c`
 - `sqllib/samples/xml/xquery/c` for C applications and
 - `sqllib/samples/xml/cpp`
 - `sqllib/samples/xml/xquery/cpp` for C++ applications
- ▶ On Windows:
 - `sqllib\samples\xml\c`
 - `sqllib\samples\xml\xquery\c` for C applications and
 - `sqllib\samples\xml\cpp`
 - `sqllib\samples\xml\xquery\cpp` for C++ applications

The build scripts, `bldapp` (on UNIX) or `bldapp.bat` (Windows,) contain the commands necessary to build a DB2 CLI application. It takes up to four parameters, represented inside the UNIX script file by the variables: `$1`, `$2`, `$3`, and `$4`, or inside the Windows file by the variables: `%1`, `%2`, `%3`, and `%4`.

- ▶ Parameter \$1 (%1): Specifies the name of your source file. This is the only required parameter, and the only one required for CLI applications that do not contain embedded SQL.

Building embedded SQL programs requires a connection to the database so three optional parameters are also provided.

- ▶ Parameter \$2 (%2): Specifies the name of the database to which you want to connect.
- ▶ Parameter \$3 (%3): Specifies the user ID for the database.
- ▶ Parameter \$4 (%4): Specifies the password.

For embedded SQL programs, the build files, `bldapp` or `bldapp.bat`, pass the parameters to the precompile and bind script, `embprep` (UNIX) or `embprep.bat` (Windows). If no database name is supplied, the default `SAMPLE` database is used. The user ID and password parameters are only required if the instance where the program is built is different from the instance where the database is located.

6.5.1 Building C/C++ applications with the sample build script

Using the supplied build files, there are three ways to build an embedded SQL application. Using, as an example, the source file `tbmod.sqc` for C or `tbmod.sqC`, the process is as follows:

- ▶ If connecting to the sample database on the same instance, enter:
`bldapp tbmod`
- ▶ If connecting to another database on the same instance, also enter the database name:
`bldapp tbmod database`
- ▶ If connecting to a database on another instance, also enter the user ID and password of the database instance:
`bldapp tbmod database userid password`

The result is an executable file, `tbmod`.

Running C/C++ applications with the sample build script

After the application has been built, there are three ways to run this embedded SQL application:

- ▶ If accessing the sample database on the same instance, simply enter the executable name:
`tbmod`

- ▶ If accessing another database on the same instance, enter the executable name and the database name:

```
tbmod database
```

- ▶ If accessing a database on another instance, enter the executable name, database name, and user ID and password of the database instance:

```
tbmod database userid password
```

Building C/C++ applications using the makefile

In addition to the sample build scripts supplied by DB2, it is possible to build all of the applications by executing the makefile that is found in the corresponding directories.

On UNIX the makefile is found in these directories:

- ▶ sqllib/samples/c
- ▶ sqllib/samples/xml/c
- ▶ sqllib/samples/xml/xquery/c for C applications and,
- ▶ sqllib/samples/cpp
- ▶ sqllib/samples/xml/cpp
- ▶ sqllib/samples/xml/xquery/cpp for in C++ applications.

On Windows the makefile is found in these directories:

- ▶ sqllib\samples\c
- ▶ sqllib\samples\xml\c
- ▶ sqllib\samples\xml\xquery\c for C applications, and
- ▶ sqllib\samples\cpp
- ▶ sqllib\samples\xml\cpp
- ▶ sqllib\samples\xml\xquery\cpp for C++ applications

Before running the makefile, modify the makefile to reflect your environment:

- ▶ set UID (user ID to access the sample database)
- ▶ set PWD (password to access the sample database)

To build the file or files, execute the appropriate command for your environment in your working directory, for example,

- ▶ UNIX:

```
make some_parameter
```

- ▶ Windows

```
nmake some_parameter
```

Where *some_parameter* corresponds to one of the parameters specified here:

- ▶ **make (or nmake) <app_name>**
/*Builds the program designated by <app_name>*/
- ▶ **make (or nmake) all**
/* Builds all supplied sample programs */
- ▶ **make (or nmake) srv**
/*Builds sample that can only be run on the server, (stored procedure)*/
- ▶ **make (or nmake) all_client**
/* Builds all client samples (all programs in the 'call_rtn' and 'client_run' categories). */
- ▶ **make (or nmake) call_rtn**
/* Builds client programs that call stored procedure */
- ▶ **make (or nmake) client_run**
/* Builds all programs that run completely on the client (not ones that call stored procedure)*/
- ▶ **make (or nmake) clean**
/* Erases all intermediate files produced in the build process */
- ▶ **make (or nmake) cleanall**
/* Erases all files produced in the build process (all files except the original source files)*/

Note: To ensure a successful build of the sample applications, we suggest that you:

- ▶ Read the Prerequisites section of the header in the sample file and follow the directions/suggestions before building or running the sample.
- ▶ Make sure that a compatible make. or nmake, executable program is resident on your system in a directory included in your PATH variable.

6.5.2 Declaring XML host variables

To transmit XML data between the database server and an embedded SQL application, you must declare host variables in your application source code.

DB2 9 introduces an XML data type that stores XML data in a structured set of nodes in a tree format. Columns with this XML data type are described as a *SQL_TYP_XML* column SQLTYPE, and applications can bind various language-specific data types for input to and output from these columns or parameters. Note that SQL_TYP_XML is a describe-only data type. It cannot be used in an SQLDA to insert or retrieve XML values; a string or binary type is required.

To access XML data, use XML host variables within your embedded SQL applications instead of casting the data to character or binary data types. If you do not make use of XML host variables, the best alternative for accessing XML data is with FOR BIT DATA or BLOB data types to avoid codepage conversion.

If a CHAR, VARCHAR, CLOB, or BLOB host variable is used for input where an XML value is expected, the value will be subject to an XMLPARSE function operation with default whitespace (STRIP) handling. Otherwise, an XML host variable is required.

To declare an XML host variable, the XML host variable must be declared as a LOB data types in the declaration section as following:

- ▶ SQL TYPE IS XML AS CLOB(n) <hostvar_name>

Where <hostvar_name> is a CLOB host variable that contains XML data encoded in the mixed codepage of the application. See Example 6-16.

Example 6-16 CLOB SQL type

```
EXEC SQL BEGIN DECLARE SECTION;
...
SQL TYPE IS XML AS CLOB(10K) xmlclob;
...
EXEC SQL END DECLARE SECTION;
```

- ▶ SQL TYPE IS XML AS DBCLOB(n) <hostvar_name>

Where <hostvar_name> is a DBCLOB host variable that contains XML data encoded in the application graphic codepage. See Example 6-17 on page 295.

Example 6-17 DBCLOB SQL type

```
EXEC SQL BEGIN DECLARE SECTION;
...
SQL TYPE IS XML AS DBCLOB(10K) xmldbclob;
...
EXEC SQL END DECLARE SECTION;
```

- ▶ SQL TYPE IS XML AS BLOB(n) <hostvar_name>

Where <hostvar_name> is a BLOB host variable that contains XML data internally encoded. See Example 6-18.

Example 6-18 BLOB SQL type

```
EXEC SQL BEGIN DECLARE SECTION;
...
SQL TYPE IS XML AS BLOB(10K) xmlblob;
...
EXEC SQL END DECLARE SECTION;
```

- ▶ SQL TYPE IS XML AS CLOB_FILE <hostvar_name>

Where <hostvar_name> is a CLOB file that contains XML data encoded in the application mixed codepage. See Example 6-19.

Example 6-19 CLOB_FILE SQL type

```
EXEC SQL BEGIN DECLARE SECTION;
...
SQL TYPE IS XML AS CLOB_FILE clob_file;
...
EXEC SQL END DECLARE SECTION;
```

- ▶ SQL TYPE IS XML AS DBCLOB_FILE <hostvar_name>

Where <hostvar_name> is a DBCLOB file that contains XML data encoded in the application graphic codepage. See Example 6-20.

Example 6-20 DBCLOB_FILE SQL type

```
EXEC SQL BEGIN DECLARE SECTION;
...
SQL TYPE IS XML AS DBCLOB_FILE dbclob_file;
...
EXEC SQL END DECLARE SECTION;
```

- SQL TYPE IS XML AS BLOB_FILE <hostvar_name>

Where <hostvar_name> is a BLOB file that contains XML data internally encoded. Example 6-21 shows an example.

Example 6-21 BLOB_FILE SQL type

```
EXEC SQL BEGIN DECLARE SECTION;
...
SQL TYPE IS XML AS BLOB_FILE blob_file;
...
EXEC SQL END DECLARE SECTION;
```

6.5.3 Referencing XML host variables

Example 6-22 shows how to reference XML host variables in a C/C++ application.

Example 6-22 Referencing XML host variables in a C/C++ application

```
// The table definition for the table myTable is: //
// CREATE TABLE myTable (id varchar(5), xmlCol XML) //
```

```
EXEC SQL BEGIN DECLARE;
  SQL TYPE IS XML AS CLOB( 10K ) xmlBuf;
  SQL TYPE IS XML AS BLOB( 10K ) xmlblob;
  SQL TYPE IS CLOB( 10K ) clobBuf;
EXEC SQL END DECLARE SECTION;
```

```
// as XML AS CLOB
EXEC SQL SELECT xmlCol INTO :xmlBuf
  FROM myTable
  WHERE id = '001';
EXEC SQL UPDATE myTable
  SET xmlCol = :xmlBuf
  WHERE id = '001';
```

```
// as XML AS BLOB
EXEC SQL SELECT xmlCol INTO :xmlblob
  FROM myTable
  WHERE id = '001';
EXEC SQL UPDATE myTable
  SET xmlCol = :xmlblob
  WHERE id = '001';
```

```

/* as CLOB using XMLSERIALIZE to return a serialized version of CLOB
data type */
// The output will be encoded in the application character codepage,
// but will not contain an XML declaration
EXEC SQL SELECT XMLSERIALIZE (xmlCol AS CLOB(10K)) INTO :clobBuf
      FROM myTable
      WHERE id = '001';
EXEC SQL UPDATE myTable
      SET xmlCol = XMLPARSE (:clobBuf PRESERVE WHITESPACE)
      WHERE id = '001';

```

6.5.4 Declaring large object type host variables

Observe the following considerations when declaring large object type (LOB type) host variables:

- ▶ The SQL TYPE IS clause is required to distinguish the three LOB-types (BLOB, CLOB, and DBCLOB) from each other. This is so that type checking and function resolution can be carried out for LOB-type host variables that are passed to functions.
- ▶ The declaration SQL TYPE IS, BLOB, CLOB, DBCLOB, K, M, G can be in mixed case.
- ▶ The maximum length allowed for the initialization string “init-data” is 32,702 bytes, including string delimiters. This is the same as the existing limit on C and C++ strings within the precompiler.
- ▶ The initialization length, *init-len*, must be a numeric constant (for example, it cannot include K, M, or G).
- ▶ A length for the LOB *must be specified*. Length can be any valid constant expression, in which the constant K, M, or G can be used. The value of length after evaluation for BLOB and CLOB must be $1 \leq \text{length} \leq 2,147,483,647$. The value of length after evaluation for DBCLOB must be $1 \leq \text{length} \leq 1,073,741,823$.
- ▶ If the LOB is not initialized within the declaration, no initialization will be done within the precompiler-generated code.
- ▶ If a DBCLOB is initialized, it is the user's responsibility to prefix the string with an “L” (indicating a wide-character string).

Note: Wide-character literals, for example L“Hello”, should only be used in a precompiled program if the WCHARTYPE CONVERT precompile option is selected. For detailed information regarding precompiler options, refer to *Developing Embedded SQL Applications*, SC10-4232.

The precompiler generates a structure tag which can be used to cast to the host variable's type. Following are generated structure tags for various data type declarations:

► BLOB example:

Declaration:

```
static Sql Type is Blob(2M) my_blob=SQL_BLOB_INIT("mydata");
```

This declaration results in the generation of the following structure:

```
static struct my_blob_t {
    sqluint32      length;
    char           data[2097152];
} my_blob=SQL_BLOB_INIT("mydata");
```

► CLOB example:

Declaration:

```
volatile sql type is clob(125m) *var1, var2 = {10, "data5data5"};
```

This declaration results in the generation of the following structure:

```
volatile struct var1_t {
    sqluint32      length;
    char           data[131072000];
} * var1, var2 = {10, "data5data5"};
```

► DBCLOB examples:

Declaration:

```
SQL TYPE IS DBCLOB(30000) my_dbclob1;
```

When precompiled with the *WCHARTYPE NOCONVERT* option, this declaration results in the generation of the following structure:

```
struct my_dbclob1_t {
    sqluint32      length;
    sqldbchar      data[30000];
} my_dbclob1;
```

Declaration:

```
SQL TYPE IS DBCLOB(30000) my_dbclob2 =
SQL_DBCLOB_INIT(L"mydbdata");
```

When precompiled with the *WCHARTYPE CONVERT* option, this declaration results in the generation of the following structure:

```
struct my_dbclob2_t {
    sqluint32      length;
    wchar_t        data[30000];
} my_dbclob2 = SQL_DBCLOB_INIT(L"mydbdata");
```

6.5.5 Referencing LOB type host variables

Example 6-23 shows examples of referencing XML LOB data type host variables in a C/C++ application:

Example 6-23 Declaring XML Lob type host variables

```
EXEC SQL BEGIN DECLARE SECTION;
    short nullind;
    static SQL TYPE IS XML AS CLOB(1k) xmlclob1=SQL_CLOB_INIT("<a> a
    </a>") ;
    static SQL TYPE IS BLOB(1k) hv_blob2 = SQL_BLOB_INIT("<init> a
    </init>");
    static SQL TYPE IS XML AS BLOB(1k) xmlblob3 = SQL_BLOB_INIT("<init>
    a</init>");
EXEC SQL END DECLARE SECTION;

EXEC SQL INSERT INTO purchaseorder (poid, porder)
    VALUES (1612, :xmlclob1:nullind);

EXEC SQL INSERT INTO purchaseorder (poid, porder)
    VALUES (712, XMLPARSE(DOCUMENT :hv_blob2:nullind STRIP WHITESPACE));

EXEC SQL INSERT INTO purchaseorder (poid, porder)
    VALUES (999, :xmlclob3:nullind);
```

6.5.6 Executing XQuery expressions in embedded SQL applications

You can store XML data in your tables and use embedded SQL applications to access the XML columns using XQuery expressions. To access XML data, use XML host variables instead of casting the data to character or binary data types.

To directly execute an XQuery expression in an embedded SQL application, prepend the expression with the XQUERY keyword. For static SQL use the XMLQUERY function. When the XMLQUERY function is called, the XQuery expression is not prefixed by XQUERY.

It is significant to note that an XQUERY statement cannot be executed statically. In order to embed an XQuery statement, an application must make use of dynamic sql statements, such as:

- ▶ PREPARE
- ▶ DECLARE CURSOR
- ▶ OPEN
- ▶ FETCH

Example 6-24 shows an embedded XQuery statement. Observe that the statement is dynamically *prepared, declared, opened, and fetched*.

Example 6-24 An embedded XQuery statement

```
EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;
    char stmt[16384];
    SQL TYPE IS XML AS BLOB( 10K ) xmlblob;
EXEC SQL END DECLARE SECTION;

sprintf( stmt, "XQUERY declare default element namespace
\"http://posample.org\";\"
        \"db2-fn:xmlcolumn('CUSTOMER.INFO')\"");

EXEC SQL PREPARE s1 FROM :stmt;
EXEC SQL DECLARE c1 CURSOR FOR s1;
EXEC SQL OPEN c1;
EXEC SQL FETCH c1 INTO :xmlblob;

while( sqlca.sqlcode == SQL_RC_OK )
{
    /* Display results */
    xmlblob.data[xmlblob.length]='\0';
    printf("\n\n\n%s",xmlblob.data);
    EXEC SQL FETCH c1 INTO :xmlblob;
    EMB_SQL_CHECK("cursor -- fetch");
}

EXEC SQL CLOSE c1;
```

The alternative to using dynamic XQuery statements, is to use the XMLQUERY function. In this way, XQuery constructs can be embedded statically in an SQL statement.

The code segment in Example 6-25 shows an embedded static SQL statement in which an XQuery is called from the XMLQUERY function.

Example 6-25 XQuery called from within an XMLQUERY function.

```
EXEC SQL BEGIN DECLARE SECTION;
    char stmt[16384];
    SQL TYPE IS XML AS BLOB( 10K ) xmlblob;
EXEC SQL END DECLARE SECTION;
```

```

EXEC SQL DECLARE C2 CURSOR FOR SELECT XMLQUERY(
'declare default element namespace "http://posample.org";
$cust/customerinfo[addr/city="Toronto"]'
PASSING CUSTOMER.INFO as "cust" RETURNING SEQUENCE BY REF) from
customer;

EXEC SQL OPEN c2;

EXEC SQL FETCH c2 INTO :xmlblob;

while( sqlca.sqlcode == SQL_RC_OK )
{
    /* Display results */
    xmlblob.data[xmlblob.length]='\0';
    printf("\n\n\n%s",xmlblob.data);
    EXEC SQL FETCH c2 INTO :xmlblob;
    EMB_SQL_CHECK("cursor -- fetch");
}

EXEC SQL CLOSE c2;

```

Identifying XML values in an SQLDA

To indicate that a base type holds XML data, the sqlname field of the SQLVAR must be updated as follows:

- ▶ The sqlname.length must be eight (8).
- ▶ The first two bytes of sqlname.data must be X'0000'
- ▶ The third and fourth bytes of sqlname.data should be X'0000' .
- ▶ The fifth byte of sqlname.data must be X'01' (referred to as the XML subtype indicator only when the first two conditions are met).
- ▶ The remaining bytes should be X'000000.'

If the XML subtype indicator is set in an SQLVAR whose SQLTYPE is non-LOB, an SQL0804 error (rc=115) will be returned at runtime.

Note: SQL_TYP_XML can only be returned from the DESCRIBE statement. This type cannot be used for any other requests. The application must modify the SQLDA to contain a valid character or binary type, and set the sqlname field appropriately to indicate that the data is XML.

For complete details concerning the SQLDA structure, refer to Chapter 3 of the manual: *Developing Embedded SQL Applications*, SC10-4232.

6.6 Java application programming

In JDBC and SQLJ applications, you can:

- ▶ Store an entire XML document in an XML column using setXXX methods.
- ▶ Retrieve an entire XML document from an XML column using getXXX methods.
- ▶ Retrieve a sequence from a document in an XML column by using the SQL XMLQUERY function to retrieve the sequence into a serialized XML string in the database, and then using getXXX methods to retrieve the data into an application variable.
- ▶ Retrieve a sequence from a document in an XML column by using an XQuery expression, prepended with the string 'XQUERY', to retrieve the elements of the sequence into a result table in the database, in which each row of the result table represents an item in the sequence. Then use getXXX methods to retrieve the data into application variables.
- ▶ Retrieve a sequence from a document in an XML column as a user-defined table by using the SQL XMLTABLE function to define the result table and retrieve it. Then use getXXX methods to retrieve the data from the result table into application variables.

Note: Java has *no* XML data type, and invocations of metadata methods, such as `ResultSetMetaData.getColumnTypeName`, will return a type of `java.sql.Types.OTHER` for an XML column type.

6.6.1 Setting up the DB2 JDBC and SQLJ development environment

In this section, we describe the procedure for setting up the DB2 JDBC and SQLJ development environment.

The following is required before setting up the DB2 JDBC and SQLJ environment:

- ▶ An SDK for Java, 1.4.2 or later. For all DB2 products except the DB2 Runtime Client, the installation process automatically or optionally installs an SDK for Java.
- ▶ JVM™ native threads support.

During the DB2 Database for Linux, UNIX, and Windows installation process, select Java support on UNIX or Linux, or JDBC support on Windows. These selections are the defaults.

Selection of Java support or JDBC support causes the DB2 installation process to automatically perform the following actions:

1. Install the IBM DB2 Driver for JDBC and SQLJ class files, and to modify the CLASSPATH to include them.
2. Install IBM DB2 Driver for JDBC and SQLJ license files, and modify the CLASSPATH to include them.
3. Configure TCP/IP.

In addition to these steps, the following steps must be completed:

- ▶ On DB2 servers on which you plan to run Java stored procedures or user-defined functions, update the database manager configuration to include the path where the SDK for Java is located.
- ▶ If you plan to run Java stored procedures that work with XML data on DB2 Database for Linux, UNIX, and Windows servers, you must set the IBM DB2 Driver for JDBC and SQLJ as the default JDBC driver for running stored procedures.

Note: For complete information regarding the installation of the DB2 driver for JDBC and SQLJ, refer to: *Developing Java Applications, SC10-4233*.

6.6.2 Building JDBC applications

DB2 provides source code samples of JDBC applications. The samples are available in the following directories:

- ▶ UNIX
JDBC: `sqllib/samples/java/jdbc`
- ▶ Windows
JDBC: `sqllib\samples\java\jdbc`

In addition to the JDBC samples that might be in the directories listed above, there are examples, *specific to XML*, that might be found in the following locations:

- ▶ On UNIX:
 - `sqllib/samples/xml/java/jdbc`
 - `sqllib/samples/xml/xquery/java/jdbc`

- On Windows:
 - sqllib\samples\xml\java\jdbc
 - sqllib\samples\xml\xquery\java\jdbc

To build and run the sample JDBC applications from the command line:

1. Compile the source_filename.java (where source_filename is the name of a source file in the samples directory) to produce the file source_filename.class with this command:

```
javac source_filename.java
```

For example, if the file is DbInfo.java the command would be:

```
javac DbInfo.java
```

2. Execute the application with this command:

```
java source_filename
```

For example, to execute the DbInfo.class the command would be:

```
java DbInfo
```

Note: You can also use the Java **makefile** command to build the sample programs provided. The **makefile** command can be found in the same directories as the source code for JDBC and SQLJ sample applications.

Inserting and updating XML data in JDBC applications

When you update or insert data into XML columns of a DB2 table, the input data must be in the serialized string format.

Table 6-1 lists the *methods* and corresponding *input data types* that can be used to input data into XML columns.

Table 6-1 *PreparedStatement methods and input data types for updating XML columns*

METHOD	Input Data Type
PreparedStatement.setAsciiStream	InputStream
PreparedStatement.setBinaryStream	InputStream
PreparedStatement.setBlob	BLOB
PreparedStatement.setBytes	byte[]
PreparedStatement.setCharacterStream	Reader
PreparedStatement.setClob	CLOB

METHOD	Input Data Type
PreparedStatement.setObject	byte[], BLOB, CLOB, DB2Xml, InputStream, Reader, String
PreparedStatement.setString	String

Encoding considerations

XML data can be internally or externally encoded. When the encoding of XML data is derived from the data itself, it is known as *internally* encoded data. If the data is derived from external sources, it is known as *externally* encoded data.

- ▶ XML data that is sent to the database server as binary data is treated as internally encoded data.
- ▶ XML data that is sent to the database server as character data is treated as externally encoded data.

External encoding for Java applications is *always* Unicode encoding.

Externally encoded data can have internal encoding. That is, the data might be sent to the database server as character data, but the data contains encoding information. The database server handles incompatibilities between internal and external encoding as follows:

- ▶ If the database server is DB2 Database for Linux, UNIX, and Windows, the database server generates an error if the external and internal encoding are incompatible, unless the external and internal encoding are Unicode. If the external and internal encoding are Unicode, the database server ignores the internal encoding.
- ▶ If the database server is DB2 for z/OS, the database server ignores the internal encoding.

Data in XML columns is stored in UTF-8 encoding. The database server handles conversion of the data from its internal or external encoding to UTF-8.

Example 6-26 illustrates a technique of inserting XML data from a file into a DB2 database using the PreparedStatement.setBinaryStream method. The data is inserted as binary data, so the database accepts the encoding.

Example 6-26 Inserting XML data from a file input as binary data

```
// Assume the table P0 exists with the following definition://
// CREATE table P0 (poid BIGINT, porder XML)
```

```
String sql = "INSERT INTO P0 VALUES(?, ?)";
PreparedStatement stmt = connection.prepareStatement(sql);
stmt.setInt(1, 5000);
```

```
File binFile = new File("myXmlFile.xml");
InputStream inBin = new FileInputStream(binFile);
stmt.setBinaryStream(2, inBin, (int) binFile.length());
stmt.execute();
```

Example 6-27 shows a technique of inserting XML data from a file into a DB2 database using the `PreparedStatement.setClob()` method. The data is inserted as character data (CLOB), so it is treated as externally encoded data.

Example 6-27 Inserting XML data from a file using the `setClob()` method

```
int customerid = 0;
String customerInfo = "";
String Data = new String();

Data=returnFileValues("myXmlFile.xml");

// Create a CLOB object
java.sql.Clob clobData =
    com.ibm.db2.jcc.t2zos.DB2LobFactory.createClob(Data);

PreparedStatement pstmt = con.prepareStatement(
    "UPDATE customer " +
    "SET INFO=XMLPARSE(document cast(? as Clob) strip whitespace)" +
    " WHERE cid=1008");

System.out.println(" Set parameter value: parameter 1 = " + "clobData"
);

pstmt.setClob(1, clobData);

pstmt.execute();
```

General recommendations for input of XML data

Here are some basic recommendations:

- ▶ If the input data is in a file, read the data in as a binary stream (`setBinaryStream`) so that the database manager processes it as internally encoded data.
- ▶ If the input data is in a Java application variable, your choice of application variable type determines whether the DB2 database manager uses any internal encoding. If you input the data as a character type (for example, `setString`), the database manager converts the data from UTF-16 (the application code page) to UTF-8 before storing it.

Retrieving XML data in JDBC applications

When you retrieve data from XML columns of a DB2 table, the output data is in the serialized string format. This is true whether you retrieve the entire contents of an XML column or a sequence from the column.

One of the following techniques can be employed to retrieve XML data:

- Use a `ResultSet.getXXX` method (other than `ResultSet.getObject()`) to retrieve the data into a compatible data type, as shown in Example 6-28.

Example 6-28 Retrieving data using `resultSet.getXXX` methods

```
String sql = "SELECT POID, DESCRIPTION from P0 where POID = ?";
PreparedStatement stmt = connection.prepareStatement(sql);

stmt.setInt(1, 5000);

ResultSet resultSet = stmt.executeQuery();
String xml = resultSet.getString("PORDER");

// also possible
InputStream inputStream = resultSet.getBinaryStream("PORDER");

// also possible
Reader reader = resultSet.getCharacterStream("PORDER");
```

- Use the `ResultSet.getObject` method to retrieve the data, and then cast it to the `DB2Xml` type and assign it to a `DB2Xml` object. Then use a `DB2Xml.getDB2XXX` or `DB2Xml.getDB2XmlXXX` method to retrieve the data into a compatible output data type. Example 6-29 illustrates this point.

Example 6-29 Retrieving data using `getObject`

```
ResultSet rs = stmt.executeQuery("XQUERY for $i in db2-fn:" +
                                "xmlcolumn('COMPANY.DOC') /company/" +
                                "emp[@id = '42366'] return $i/name ");

while (rs.next())
{
    com.ibm.db2.jcc.DB2Xml data = (com.ibm.db2.jcc.DB2Xml)
    rs.getObject(1);
    // Print the result as an DB2 XML String
    System.out.println();
    System.out.println(data.getDB2XmlString());
    System.out.println();
}
```

Table 6-2 lists the `ResultSet` methods and corresponding output data types for retrieving XML data.

Table 6-2 ResultSet methods and output data types for retrieving XML data

Method	Output data type
<code>ResultSet.getAsciiStream</code>	<code>InputStream</code>
<code>ResultSet.getBinaryStream</code>	<code>InputStream</code>
<code>ResultSet.getBytes</code>	<code>byte[]</code>
<code>ResultSet.getcharacterStream</code>	<code>Reader</code>
<code>ResultSet.getObject</code>	<code>DB2Xml</code>
<code>ResultSet.getString</code>	<code>String</code>

Table 6-3 lists the methods and corresponding output data types for retrieving data from a `DB2Xml` object, as well as the type of encoding in the XML declaration that the driver adds to the output data. To summarize Table 6-3:

- ▶ `DB2Xml.getDB2XmlXXX` methods add XML declarations with encoding specifications to the output data.
- ▶ `DB2Xml.getDB2XXX` methods do *not* add XML declarations with encoding specifications to the output data.

Table 6-3 Methods, output data types, and encoding specifications

Method	Output data type	Type of XML internal encoding declaration added
<code>DB2Xml.getDB2AsciiStream</code>	<code>InputStream</code>	None
<code>DB2Xml.getDB2BinaryStream</code>	<code>InputStream</code>	None
<code>DB2Xml.getDB2Bytes</code>	<code>byte[]</code>	None
<code>DB2Xml.getDB2CharacterStream</code>	<code>Reader</code>	None
<code>DB2Xml.getDB2String</code>	<code>String</code>	None
<code>DB2Xml.getDB2XmlAsciiStream</code>	<code>InputStream</code>	US-ASCII
<code>DB2Xml.getDB2XmlBinaryStream</code>	<code>InputStream</code>	Specified by <code>getDB2XmlBinaryStream</code> <code>targetEncoding</code> parameter
<code>DB2Xml.getDB2XmlBytes</code>	<code>byte[]</code>	Specified by <code>DB2Xml.getDB2XmlBytes</code> <code>targetEncoding</code> parameter
<code>DB2Xml.getDB2XmlCharacterString</code>	<code>Reader</code>	ISO-10646-UCS-2
<code>DB2Xml.getDB2XmlString</code>	<code>String</code>	ISO-10646-UCS-2

General recommendations for output of XML data

When XML data is output to a file as nonbinary data, XML internal encoding should be added to the output data, that is, DB2Xml.getDB2XmlXXX methods.

6.6.3 Building SQLJ applications

DB2 provides source code samples of SQLJ applications. The samples are available in the following directories:

- ▶ UNIX
SQLJ:
sqllib/samples/java/sqlj
- ▶ Windows
SQLJ:
sqllib/samples/java/sqlj

In addition to the SQLJ samples in the foregoing directories, there are examples, *specific to XML*, that can be found in the following locations:

- ▶ On UNIX:
 - sqllib/samples/xml/java/sqlj
 - sqllib/samples/xml/xquery/java/sqlj
- ▶ On Windows:
 - sqllib\samples\xml\java\sqlj
 - sqllib\samples\xml\xquery\java\sqlj

DB2 provides build files, found in the same directories as the source code for the sample applications, which contain commands to build either an SQLJ applet or application. The build files are:

- ▶ bldsqlj (UNIX), or
- ▶ bldsqlj.bat (Windows)

To build and run the sample SQLJ applications:

1. Compile the source code by entering this command on the command line:

```
bldsqlj source_filename <userid> <password> <server_name>  
<port_number> <db_name>
```

source_filename is the name of a .sqlj source file in the samples directory.

The parameters <userid> <password> <server_name> <port_number> <db_name> can have default values, as explained in the build file.

For example, if the file is DbAuth.sqlj the command would be:

```
bldsqlj DbAuth
```

OR

```
bldsqlj,bat DbAuth
```

2. Execute the application with this command:

```
java DbAuth
```

Note: If you are running a Java application on UNIX in a 64-bit DB2 instance but the software development kit for Java is 32-bit, you have to change the DB2 library path before running the application. For example, on AIX:

- ▶ If using bash or Korn shell:

```
export LIBPATH=$HOME/sql1lib/lib32
```
- ▶ If using C shell:

```
setenv LIBPATH $HOME/sql1lib/lib32
```

Inserting and updating XML data in SQLJ applications

When you update or insert data into XML columns of a DB2 table, the input data must be in the serialized string format. The host expression data types that you can use to update XML columns are:

- ▶ `com.ibm.db2.jcc.DB2Xml`
- ▶ `String`
- ▶ `byte`
- ▶ `Blob`
- ▶ `Clob`
- ▶ `sqlj.runtime.AsciiStream`
- ▶ `sqlj.runtime.BinaryStream`
- ▶ `sqlj.runtime.CharacterStream`

Encoding considerations

As with JDBC applications, XML data in SQLJ applications can be internally or externally encoded. When the encoding of XML data is derived from the data itself, it is known as *internally* encoded data. If the data is derived from external sources, it is known as *externally* encoded data.

- ▶ XML data that is sent to the database server as binary data is treated as internally encoded data.
- ▶ XML data that is sent to the database server as character data is treated as externally encoded data.

External encoding for Java applications is *always* Unicode encoding.

Externally encoded data can have internal encoding. That is, the data might be sent to the database server as character data, but the data contains encoding information. The database server handles incompatibilities between internal and external encoding as follows:

- ▶ If the database server is DB2 Database for Linux, UNIX, and Windows, the database server generates an error if the external and internal encoding are incompatible, unless the external and internal encoding are Unicode. If the external and internal encoding are Unicode, the database server ignores the internal encoding.
- ▶ If the database server is DB2 for z/OS, the database server ignores the internal encoding.

Data in XML columns is stored in UTF-8 encoding. The database server handles conversion of the data from its internal or external encoding to UTF-8.

Examples

Example 6-30 demonstrates inserting data from a String host expression, `xmlData`, into an XML column. The String `xmlData` is a character type, so external encoding is used, whether or not internal encoding is specified.

Example 6-30 Inserting data from a String host expression

```
String xmlData = "XMLPARSE(document '<customerinfo ' +
    "cid=\"999\"><address country= " +
    "\"US\"><street>225 Brown St.\" +
    "</street><city>White Plains</city><state>"+
    "NEW YORK</state></address>" +
    "</customerinfo>' preserve whitespace)";

#sql [ctx] {INSERT INTO CUSTOMER VALUES (1, :xmlData)};
```

Example 6-31 demonstrates copying data from a String, `xmlString`, into a byte array with CP500 encoding; the data then contains an XML declaration with an encoding declaration for CP500. In this example, the data is then inserted from the `byte[]` host expression into an XML column. A byte string is considered to be *internally* encoded data.

Example 6-31 Copying data from a String into a byte array with CP500 encoding

```
String xmlData = "XMLPARSE(document '<customerinfo ' +
    "cid=\"999\"><address country= " +
    "\"US\"><street>225 Brown St.\" +
    "</street><city>White Plains</city><state>"+
    "NEW YORK</state></address>" +
    "</customerinfo>' preserve whitespace)";
```

```
byte[] xmlBytes = xmlData.getBytes("CP500");
#sql[ctx] {INSERT INTO CUSTOMER VALUES (4, :xmlBytes)};
```

Example 6-32 shows an example of copying data from a `String`, `xmlData`, into a byte array with US-ASCII encoding. Following this, an `sqlj.runtime.AsciiStream` host expression is constructed, and data is inserted from the `sqlj.runtime.AsciiStream` host expression into an XML column. `sqljXmlAsciiStream` is a stream type, so its *internal* encoding is used. The data is converted from its internal encoding to UTF-8 encoding and stored in its hierarchical form on the database server.

Example 6-32 Inserting data from an `sqlj.runtime.AsciiStream`

```
String xmlData = "XMLPARSE(document '<customerinfo " +
    "cid=\"999\"><address country= " +
    "\"US\"><street>225 Brown St.\" +
    "</street><city>White Plains</city><state>" +
    "NEW YORK</state></address>" +
    "</customerinfo>' preserve whitespace)";

byte[] b = xmlData.getBytes("US-ASCII");
java.io.ByteArrayInputStream xmlAsciiInputStream = new
java.io.ByteArrayInputStream(b);
sqlj.runtime.AsciiStream sqljXmlAsciiStream = new
sqlj.runtime.AsciiStream(xmlAsciiInputStream, b.length);
#sql[ctx] {INSERT INTO CUSTOMER VALUES (4, :sqljXmlAsciiStream)};
```

Example 6-33 illustrates constructing an `sqlj.runtime.CharacterStream` host expression, and inserting data from the `sqlj.runtime.CharacterStream` host expression into an XML column. `sqljXmlCharacterStream` is a character type, so its external encoding is used, whether or not it has an internal encoding specification.

Example 6-33 Inserting data from a `sqljXmlCharacterStream` host expression

```
String xmlData = "XMLPARSE(document '<customerinfo " +
    "cid=\"999\"><address country= " +
    "\"US\"><street>225 Brown St.\" +
    "</street><city>White Plains</city><state>" +
    "NEW YORK</state></address>" +
    "</customerinfo>' preserve whitespace)";
```

```

java.io.StringReader xmlReader = new java.io.StringReader(xmlData);
sqlj.runtime.CharacterStream sqljXmlCharacterStream = new
sqlj.runtime.CharacterStream(xmlReader, xmlData.length());
#sql [ctx] {INSERT INTO CUSTOMER VALUES (4, :sqljXmlCharacterStream)};

```

Example 6-34 demonstrates retrieving a document from an XML column into a `com.ibm.db2.jcc.DB2Xml` host expression. The data is then inserted into an XML column, in the same table. No conversion occurs because after you retrieve the data it is still in UTF-8 encoding.

Example 6-34 Retrieving a document into a `com.ibm.db2.jcc.DB2Xml` host expression

```

java.sql.ResultSet rs = s.executeQuery ("SELECT * FROM CUSTOMER");
rs.next();
com.ibm.db2.jcc.DB2Xml xmlObject =
(com.ibm.db2.jcc.DB2Xml)rs.getObject(2);
#sql [ctx] {INSERT INTO CUSTOMER VALUES (6, :xmlObject)};

```

Retrieving XML data in SQLJ applications

When you update or insert data into XML columns of a DB2 table, the input data must be in the serialized string format. The host expression data types that you can use to update XML columns are:

- ▶ `com.ibm.db2.jcc.DB2Xml`
- ▶ `String`
- ▶ `byte`
- ▶ `sqlj.runtime.AsciiStream`
- ▶ `sqlj.runtime.BinaryStream`
- ▶ `sqlj.runtime.CharacterStream`

Table 6-4 lists the methods that can be called to retrieve data from an `com.ibm.db2.jcc.DB2Xml` object, as well as the corresponding output data types and type of encoding in the XML declarations.

Table 6-4 Methods for retrieving XML data

Method	Output data type	Type of XML internal encoding declaration added
<code>DB2Xml.getDB2AsciiStream</code>	<code>InputStream</code>	None
<code>DB2Xml.getDB2BinaryStream</code>	<code>InputStream</code>	None
<code>DB2Xml.getDB2Bytes</code>	<code>byte[]</code>	None
<code>DB2Xml.getDB2CharacterStream</code>	<code>Reader</code>	None

Method	Output data type	Type of XML internal encoding declaration added
DB2Xml.getDB2String	String	None
DB2Xml.getDB2XmlAsciiStream	InputStream	US-ASCII
DB2Xml.getDB2XmlBinaryStream	InputStream	Specified by getDB2XmlBinaryStream targetEncoding parameter
DB2Xml.getDB2XmlBytes	byte[]	Specified by DB2Xml.getDB2XmlBytes targetEncoding parameter
DB2Xml.getDB2XmlCharacterString	Reader	ISO-10646-UCS-2
DB2Xml.getDB2XmlString	String	ISO-10646-UCS-2

If the application does not call the XMLSERIALIZE function before data retrieval, the data is converted from UTF-8 to the external application encoding for the character data types, or the internal encoding for the binary data types. No XML declaration is added.

Examples

The code segment in Example 6-35 is an example of retrieving data from an XML column into a String host expression. Because the String type is a character type, the data is converted from UTF-8, to the external encoding and returned without any XML declaration.

Example 6-35 Retrieving data from an XML column into a string

```
#sql iterator XmlStringIter (int, String);
#sql [ctx] siter = {SELECT poid, porder FROM po};
#sql {FETCH :siter INTO :row, :outString};
```

Example 6-36 demonstrates retrieving data from an XML column into a byte [] host expression. Because the byte [] data type is a binary type, the data is converted from UTF-8 to the internal encoding, and returned without any XML declaration.

Example 6-36 Retrieving data from an XML column into a byte[] host expression

```
#sql iterator XmlByteArrayIter (int, byte[]);
XmlByteArrayIter biter = null;
#sql [ctx] biter = {SELECT poid, porder FROM po};
#sql {FETCH :biter INTO :row, :outBytes};
```

The code segment for Example 6-37 shows retrieving a document from an XML column into a `com.ibm.db2.jcc.DB2Xml` host expression. In this example, the data is in a byte string with an XML declaration that includes an internal encoding specification for UTF-8.

Example 6-37 Retrieving data from an XML column into a UTF-8 byte[] host expression

```
#sql iterator DB2XmlIter (int, com.ibm.db2.jcc.DB2Xml);  
DB2XmlIter db2xmliter = null;  
com.ibm.db2.jcc.DB2Xml outDB2Xml = null;  
#sql [ctx] db2xmliter = {SELECT poid, porder FROM po};  
#sql {FETCH :db2xmliter INTO :row, :outDB2Xml};  
byte[] byteArray = outDB2XML.getDB2XmlBytes("UTF-8");
```

The FETCH statement retrieves the data into the DB2Xml object in UTF-8 encoding. The `getDB2XmlBytes` method with the UTF-8 argument adds an XML declaration with a UTF-8 encoding specification and stores the data in a byte array.

6.7 Building DB2 applications with PHP

PHP Hypertext Preprocessor (PHP) is an interpreted programming language primarily intended for the development of Web applications. PHP has become a popular language for Web application development because of its focus on practical solutions and support for the most commonly required functionality in Web applications.

PHP is a modular language that enables you to customize the available functionality through the use of extensions. These extensions can simplify tasks such as reading, writing, and manipulating XML, creating SOAP clients and servers, and encrypting communications between server and browser. The most popular extensions for PHP, however, provide read and write access to databases so that you can easily create a dynamic database-driven Web site.

On Windows, precompiled binary versions of PHP are available for download from this Web site:

<http://php.net/>

Most Linux distributions include a precompiled version of PHP.

Your own version of PHP can be compiled on UNIX operating systems that do not include a precompiled version of PHP.

6.7.1 Setting up the PHP application development environment

This section covers prerequisites for, and installation of, the PHP application development environment on Linux, UNIX, and Windows.

Note: For complete information about setting up the PHP application development environment on Linux, UNIX, and Windows, refer to the manual *Developing Perl and PHP Applications*, SC10-4234.

Linux and UNIX

Prerequisites for a PHP installation on Linux and UNIX are as follows:

- ▶ The Apache HTTP Server must be installed on your system.
- ▶ The DB2 development header files and libraries must be installed on your system.
- ▶ The gcc compiler and other development packages, including the apache-devel, autoconf, automake, bison, flex, gcc, and libxml2-devel packages, must be installed on your system.

Here is a brief overview of the steps involved in installing PHP on Linux or UNIX:

1. Download the latest version of the PHP tar file from:
<http://www.php.net>
2. Configure the makefile.
3. Compile the files by issuing the `make` command.
4. Install the files by issuing the `make install` command.
5. Install the `ibm_db2` extension.
6. Edit the `php.ini` file.
7. Restart the Apache server.

Windows

There is one prerequisite for an installation of PHP on Windows:

- ▶ The Apache HTTP Server must be installed.

Here is a brief overview of the steps involved in installing PHP on Windows:

1. Download the latest version of the PHP zip package and the collection of PECL modules zip package from
<http://www.php.net>
2. Extract the PHP zip package into an install directory.
3. Extract the collection of PECL modules zip package into the \ext\ subdirectory of your PHP installation directory.
4. Edit the php.ini file.
5. Enable PHP support in Apache HTTP Server 2.x.
6. Restart the Apache HTTP Server.

PHPEclipse and the Developer Workbench

Developer Workbench is built on the Eclipse framework. This framework allows the installation of IDE plug-ins that are created to support various application development APIs. One such plug-in, PHPEclipse, is available for PHP. To acquire the PHPEclipse plug-in, complete the following steps:

1. Open the Eclipse IDE on your development desktop.
2. Click **Help** → **Software Updates** → **Find/Install** from the file menu in Eclipse.
3. Select the radio button labeled, **Search for new features to install**.
4. Click the **New Remote Site** button.
5. Type PHP SourceForge as the name, and enter the URL as:
<http://phpeclipse.sourceforge.net/update/releases>
6. Click **OK**, then click **Finish**.
7. A list of features will be presented; open the list and check the one labeled **phpeclipse**.
8. Click **Next**.
9. Follow the on-screen instructions to finish the automatic installation.

After restarting Eclipse, switch to the IDE perspective specific to PHP:

1. Under the Window menu, choose **Open Perspective**.
2. Select **Other**
3. Select **PHP** and click **OK**.

The PHPEclipse IDE in Developer Workbench is shown in Figure 6-19.

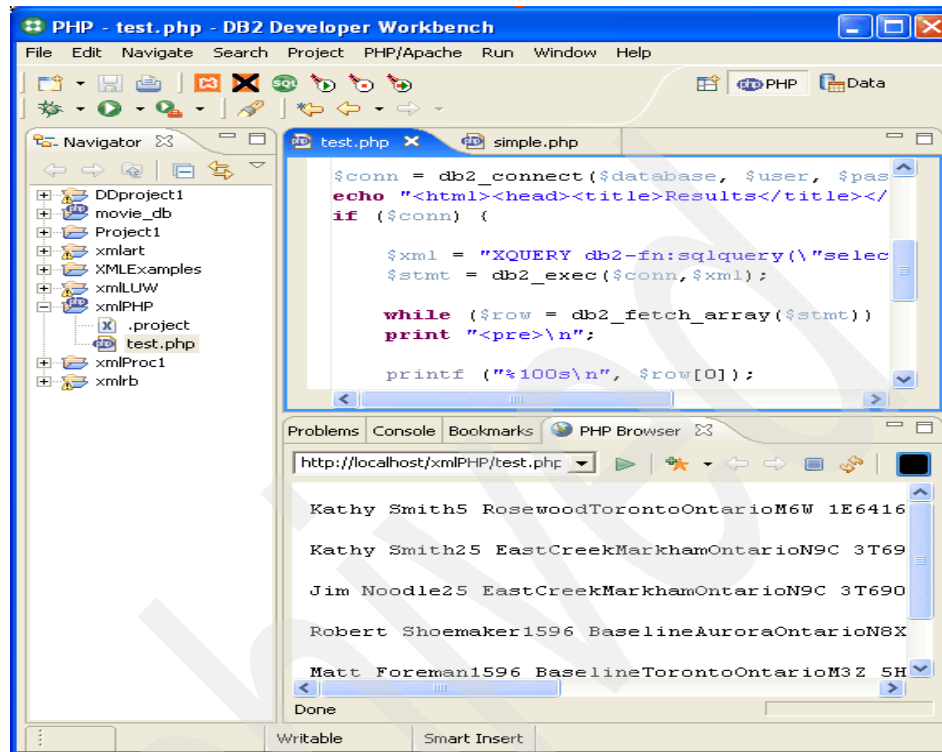


Figure 6-19 PHPEclipse IDE in Developer Workbench

6.7.2 Introduction to PHP application development for DB2

IBM supports access to DB2 databases from PHP applications through two extensions offering distinct sets of features:

- ▶ *ibm_db2* is an extension written, maintained, and supported by IBM for access to DB2 databases. The *ibm_db2* extension offers a procedural application programming interface (API) that, in addition to the normal create, read, update, and write database operations, also offers extensive access to the database metadata. You can compile the *ibm_db2* extension with either PHP 4 or PHP 5.
- ▶ *PDO_ODBC* is a driver for the PHP Data Objects (PDO) extension that offers access to DB2 databases through the standard object-oriented database interface introduced in PHP 5.1. Despite its name, you can compile the *PDO_ODBC* extension directly against the DB2 libraries to avoid the communications overhead and potential interference of an ODBC driver manager.

A third extension, Unified ODBC, has historically offered access to DB2 database systems. We do *not* recommend that you write new applications with this extension because `ibm_db2` and `PDO_ODBC` both offer significant performance and stability benefits over Unified ODBC. The `ibm_db2` extension API makes porting an application that was previously written for Unified ODBC almost as easy as globally changing the `odbc_` function name to `db2_` throughout the source code of your application.

Executing XQuery expressions in PHP (`ibm_db2`)

After connecting to a DB2 database, your PHP script is ready to issue XQuery expressions. The `db2_exec()` and `db2_execute()` functions execute SQL statements, through which you can pass your XQuery expressions. A typical use of `db2_exec()` is to set the default schema for your application in a common include file or base class.

Call `db2_exec()` with the following arguments:

- ▶ The connection resource
- ▶ A string containing the SQL statement, including the XQuery expression:
The XQuery expression must be wrapped in an `XMLQUERY` clause in the SQL statement.
- ▶ (Optional): An array containing one of the two following statement options:
 - `DB2_ATTR_CASE`
 - `DB2_ATTR_CURSOR`

DB2_ATTR_CASE

For compatibility with database systems that do not follow the SQL standard, this option sets the case in which column names will be returned to the application. By default, the case is set to `DB2_CASE_NATURAL`, which returns column names as they are returned by DB2. You can set this parameter to `DB2_CASE_LOWER` to force column names to lower case, or to `DB2_CASE_UPPER` to force column names to upper case.

DB2_ATTR_CURSOR

This option sets the type of cursor that `ibm_db2` returns for result sets. By default, `ibm_db2` returns a forward-only cursor (`DB2_FORWARD_ONLY`) which returns the next row in a result set for every call to `db2_fetch_array()`, `db2_fetch_assoc()`, `db2_fetch_both()`, `db2_fetch_object()`, or `db2_fetch_row()`. You can set this parameter to `DB2_SCROLLABLE` to request a scrollable cursor so that the `ibm_db2` fetch functions accept a second argument specifying the absolute position of the row that you want to access within the result set.

The value returned by `db2_exec()` will indicate if the SQL statement succeeded or failed. The significance of the values returned can be explained this way:

- ▶ If the value is `FALSE`, the SQL statement failed. You can retrieve diagnostic information through the `db2_stmt_error()` and `db2_stmt_errormsg()` functions.
- ▶ If the value is *not* `FALSE`, the SQL statement succeeded and returned a statement resource that can be used in subsequent function calls related to this query.

Example 6-38 illustrates an example of a PHP program that executes an XQuery statement and returns a result set.

Example 6-38 PHP program that executes an XQuery and returns a result set

```
<?php
$database = 'sample';
$user = 'db2inst1';
$password = 'db2pwd';

$conn = db2_connect($database, $user, $password);

if ($conn) {

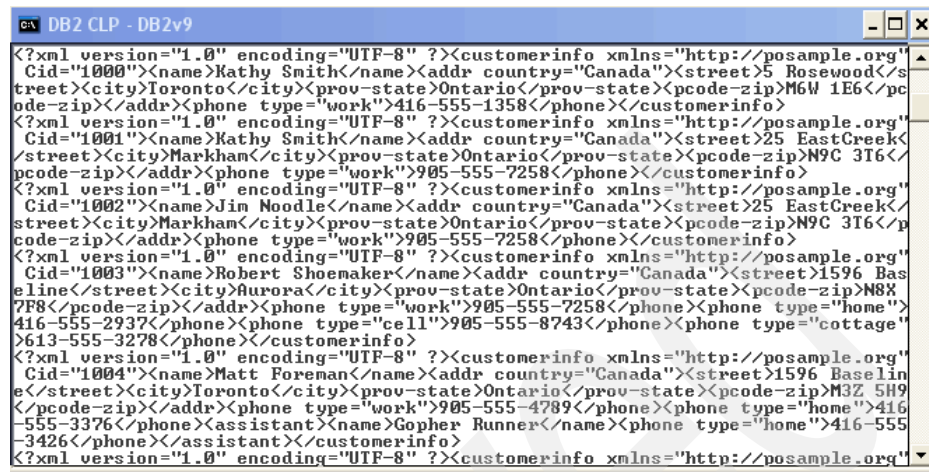
    $xml = "XQUERY db2-fn:sqlquery(\"select info from customer\")";
    $stmt = db2_exec($conn,$xml);

    while ($row = db2_fetch_array($stmt)) {
        printf ("%100s\n", $row[0]);
    }

    db2_close($conn);
}
else {
    echo "Connection failed.";
}

?>
```

Figure 6-20 shows the output of the execution of the preceding PHP source.



```
<?xml version="1.0" encoding="UTF-8" ?><customerinfo xmlns="http://posample.org"
Cid="1000"><name>Kathy Smith</name><addr country="Canada"><street>5 Rosewood</s
treet><city>Toronto</city><prov-state>Ontario</prov-state><pcode-zip>M6W 1E6</pc
ode-zip></addr><phone type="work">416-555-1358</phone></customerinfo>
<?xml version="1.0" encoding="UTF-8" ?><customerinfo xmlns="http://posample.org"
Cid="1001"><name>Kathy Smith</name><addr country="Canada"><street>25 EastCreek</
street><city>Markham</city><prov-state>Ontario</prov-state><pcode-zip>N9C 3T6</p
code-zip></addr><phone type="work">905-555-7258</phone></customerinfo>
<?xml version="1.0" encoding="UTF-8" ?><customerinfo xmlns="http://posample.org"
Cid="1002"><name>Jim Noodle</name><addr country="Canada"><street>25 EastCreek</
street><city>Markham</city><prov-state>Ontario</prov-state><pcode-zip>N9C 3T6</p
code-zip></addr><phone type="work">905-555-7258</phone></customerinfo>
<?xml version="1.0" encoding="UTF-8" ?><customerinfo xmlns="http://posample.org"
Cid="1003"><name>Robert Shoemaker</name><addr country="Canada"><street>1596 Baselin
e</street><city>Aurora</city><prov-state>Ontario</prov-state><pcode-zip>N8X
7F8</pcode-zip></addr><phone type="work">905-555-7258</phone><phone type="home">
416-555-2937</phone><phone type="cell">905-555-8743</phone><phone type="cottage"
>613-555-3278</phone></customerinfo>
<?xml version="1.0" encoding="UTF-8" ?><customerinfo xmlns="http://posample.org"
Cid="1004"><name>Matt Foreman</name><addr country="Canada"><street>1596 Baselin
e</street><city>Toronto</city><prov-state>Ontario</prov-state><pcode-zip>M3Z 5H9
</pcode-zip></addr><phone type="work">905-555-4789</phone><phone type="home">416
-555-3376</phone><assistant><name>Gopher Runner</name><phone type="home">416
-3426</phone></assistant></customerinfo>
<?xml version="1.0" encoding="UTF-8" ?><customerinfo xmlns="http://posample.org"
```

Figure 6-20 Results of the execution of an XQuery in a PHP program

6.8 The DB2 .NET environment

There are several prerequisites that you must check before developing applications in the .NET environment. In brief, verify the following items:

- ▶ What is the supported .NET development software?
- ▶ Is the Windows application development environment set up and configured correctly?
- ▶ Is the DB2 Visual Studio Add-In installed?
- ▶ Have the DB2 .Net provider system requirements been met?

Note: For in-depth information about environmental prerequisites, installation, and configuration concerns, refer to the manual: *Developing ADO.NET and OLE DB Applications*, SC10-4230.

6.8.1 Building sample applications for the DB2 .NET data provider

DB2 provides a batch file, *bldapp.bat*, for compiling and linking DB2 Visual Basic or DB2 C# .NET applications.

The Visual Basic .NET samples are located in:

sql1lib\samples\.NET\vb

The DB2 C# .Net samples are located in:

sql11b\samples\ .NET\cs directory

Along with these files are the sample programs that can be built with these files. The batch file (bldapp.bat), takes one parameter, %1, for the name of the source file to be compiled (without the .vb or .cs extension).

Refer to “XML and XQuery support in C# .NET CLR routines” on page 343 in this document for information regarding XML and C# applications.

6.8.2 XML support in Visual Studio.NET: overview

Support for DB2 9 XML features in IBM Visual Studio 2005 Add-In includes:

- ▶ XML data visualization:
 - Full integration with .NET XML designer
 - XML data validation
 - XML data import and export
- ▶ New XML index designer; XML pattern builder
- ▶ XQuery script designer
- ▶ Full integration with DB2 XML Schema Repository:
 - Full integration with .NET XML Schema designer
 - Can create/register/test annotated XSD using IBM DB2 mapping editor

DB2.NET Data Provider provides the following features:

- ▶ Enables access to DB2 Family of servers
- ▶ Fully implements the Microsoft ADO.NET data access APIs/interfaces
- ▶ Binding XML types to String, byte[], XMLReader, XPathDocument
- ▶ IBM.Data.DB2 namespace

6.8.3 XML data type support in Visual Studio .NET

Because DB2 now provides support for XML data processing, and XML values can be stored natively in an XML data type column, there are additional features that have been implemented in the Visual Studio add-in. The areas affected in .NET are:

- ▶ IBM Table Designer
- ▶ IBM View Designer
- ▶ IBM Procedure Designer
- ▶ IBM Data Designer
- ▶ Index Designer

IBM Table Designer and IBM View Designer

You can define a column of XML type in tables or views using the IBM Table or View Designer. Figure 6-21 illustrates an example of creating a table with an XML column using the IBM Table Designer.

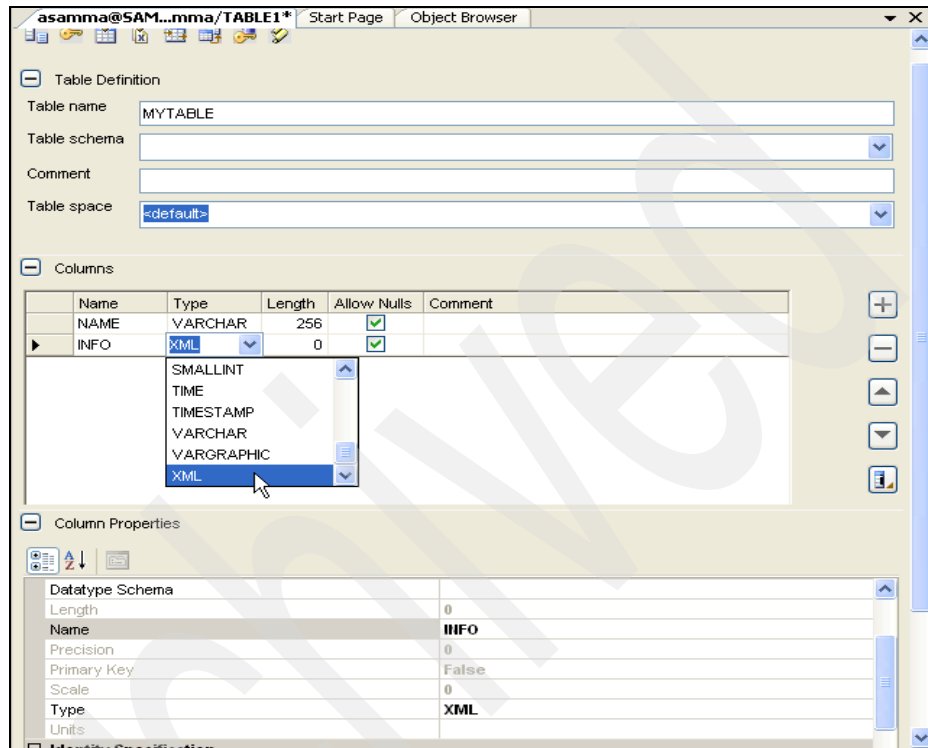


Figure 6-21 Creating an XML column using the IBM Table Designer

IBM Procedure Designer

An XML data type can be defined as an *IN*, *OUT*, or *INOUT* parameter in a stored procedure created by the IBM Procedure Designer. An example is shown in Figure 6-22.

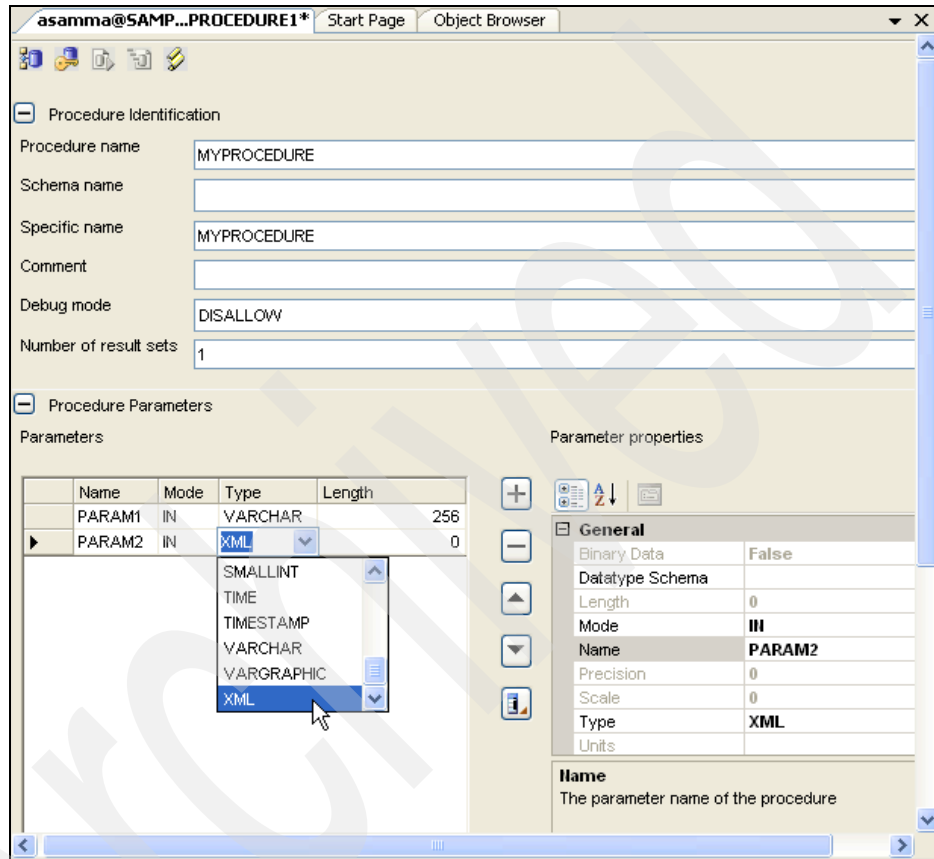


Figure 6-22 Using IBM Procedure Designer to create a procedure

IBM Data Designer

IBM Data Designer allows you to modify, export, or import data and visualize XML data. As shown in Figure 6-23, when an XML column is selected, a drop-down box offers three choices:

- ▶ XML Designer
- ▶ HTML Visualizer
- ▶ Clear Data

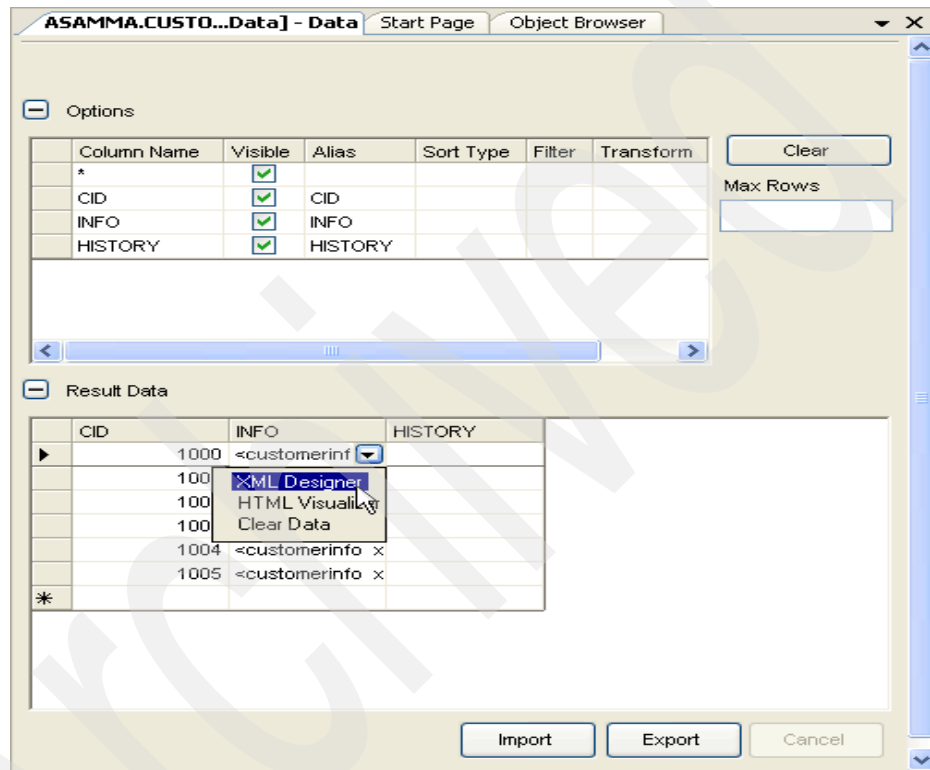


Figure 6-23 Selecting an XML column offers three choices

XML Designer

Choosing XML Designer opens the DB2 XML Designer window. The XML Designer window contains three tabs: Text View; Grid View; and Sample XML.

Text View

The editor section, the top portion, of the Text View window allow you to enter XML *manually*. The editor also provides intellisense, word completion, and syntactical colorization.

Alternately, you can choose an XML file from the file system by selecting **Open File** from the lower portion of the window. See Figure 6-24.

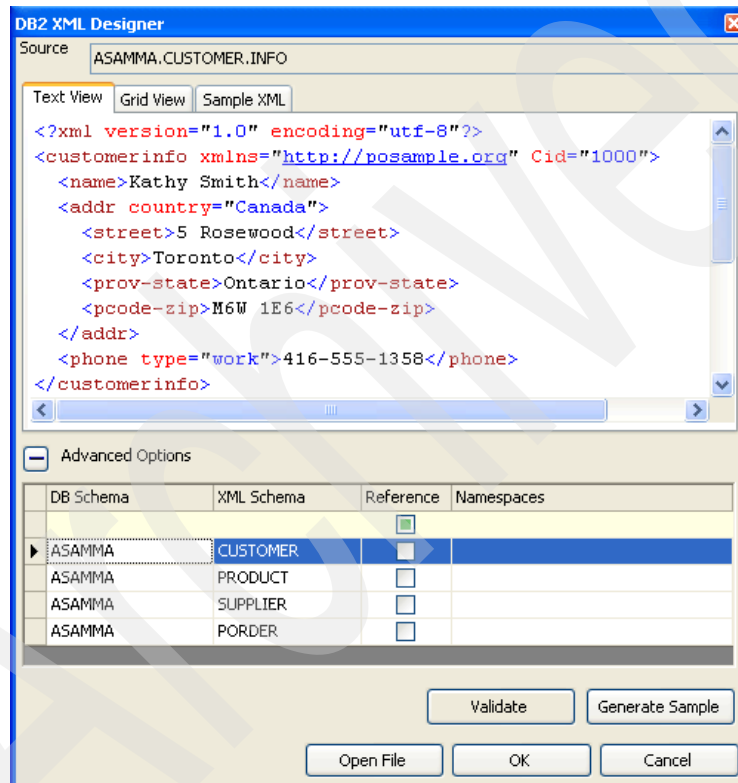


Figure 6-24 The Text View from the DB2 XML Designer

Grid view

When you select the Grid view tab, the XML document is shown in grid form. From this view, you can enter values inside the XML navigation grid cell. See Figure 6-25.

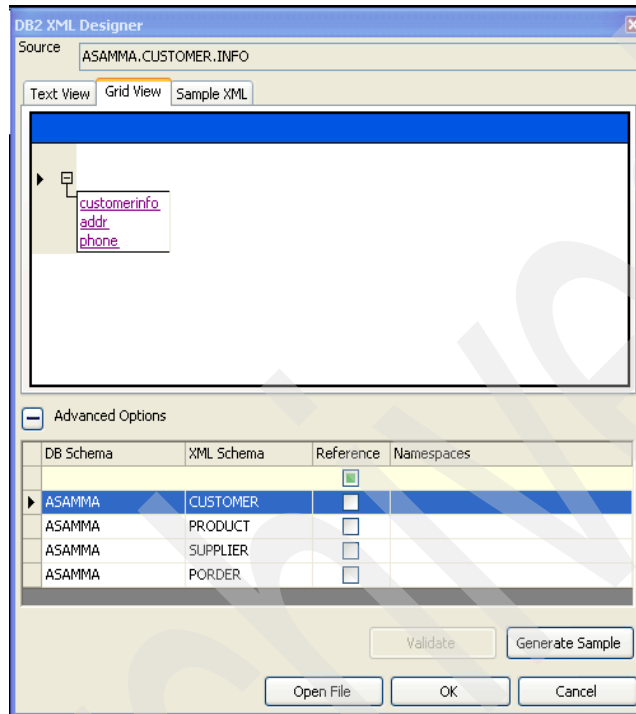


Figure 6-25 DB2 XML Designer Grid View

If you select an element from the Grid View, it is possible to drill-down into the child elements and attributes of that element. An example of this is seen in Figure 6-26. In this example the **customerinfo** element was chosen. From this view, it is also possible to modify the current cell or add a new row.

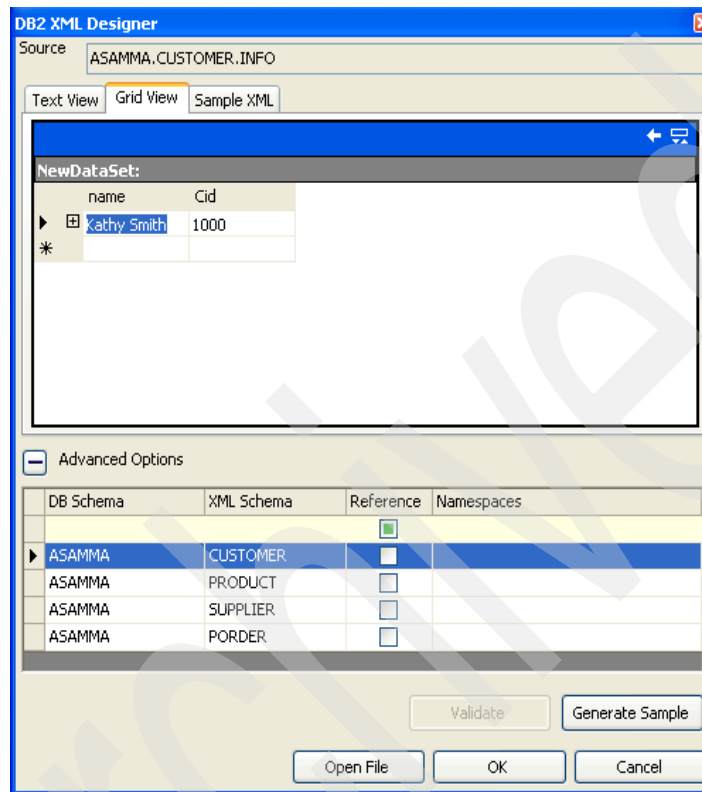


Figure 6-26 Drilling-down into the elements of an XML column

If the content is changed in the Text View, the changes will be synchronized and shown in the grid view, and vice versa.

HTML Visualizer

When you choose the HTML Visualizer from the IBM Data Designer, an embedded browser is launched, as in Figure 6-27. The XML content is shown in the browser.

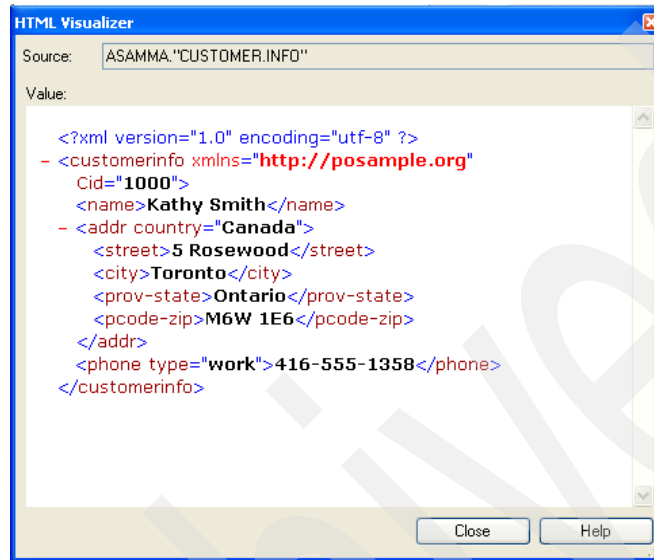


Figure 6-27 XML content in HTML Visualizer

Clear Data

When you choose **Clear Data** from the IBM Data Designer, data is deleted from the XML column.

Index Designer

After you have created a table, you can add an index to an XML column. To complete this, *right-click* on an existing table and select **Open Definition** to start the IBM Table Designer. See Figure 6-28.

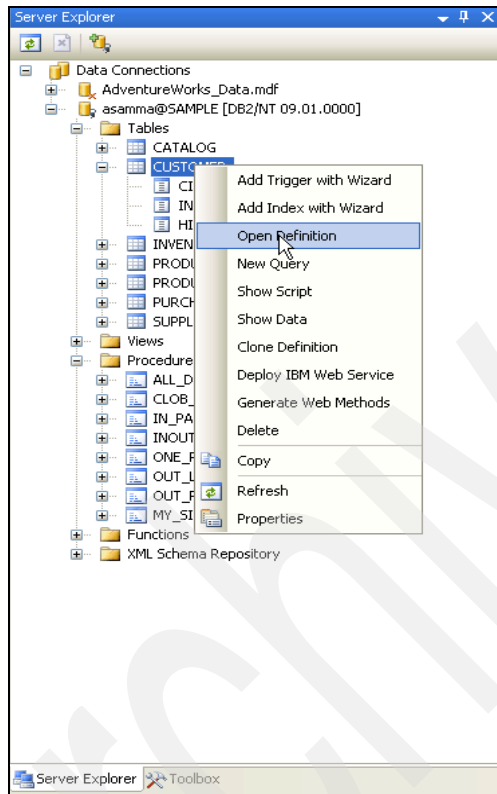


Figure 6-28 Open the IBM Table Designer

To launch the XML Index view, click the XML indexes toolbar button, highlighted in red in Figure 6-29.

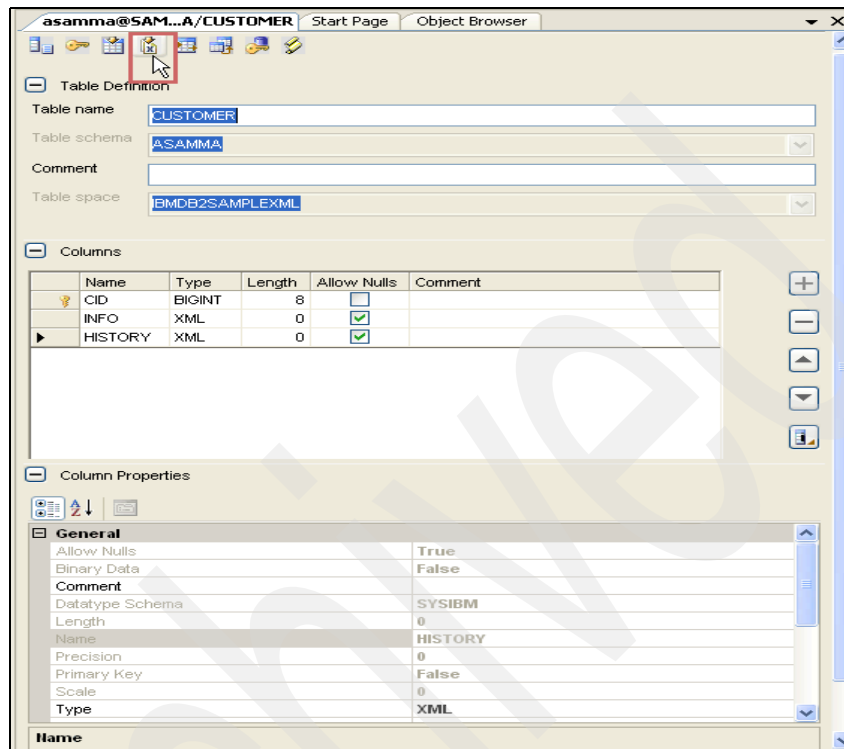


Figure 6-29 Select the XML indexes toolbar button to launch the XML Index view

The XML Index designer consists of two panes:

- ▶ Index Properties Grid
- ▶ XML Pattern Selection

From the Property Grid on the Index Properties Grid Pane it is possible to add or remove indices by selecting the (+) or (-) symbols. In the Index properties, you can set or unset index properties. See Figure 6-30.

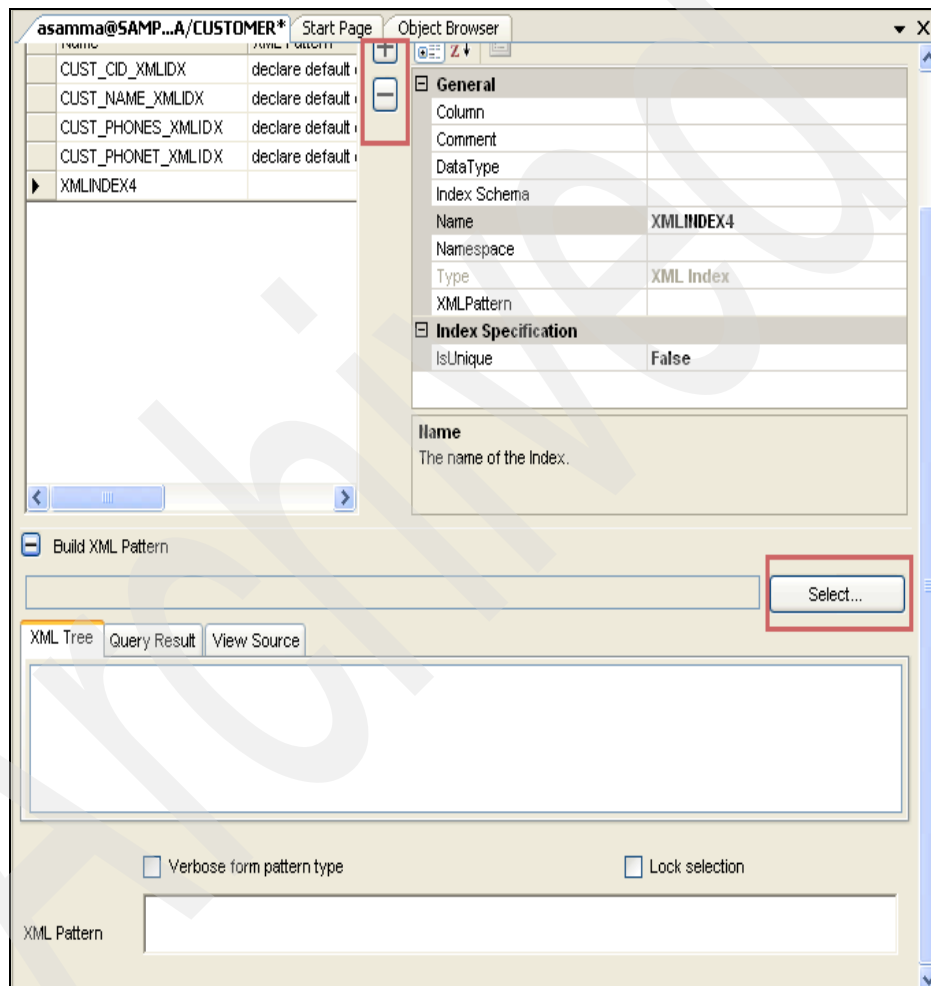


Figure 6-30 Index properties Grid pane

When you choose **Select...** on the Build XML Pattern source button, it launches the XML Pattern Source dialog box (Figure 6-31). The following options are available from this dialog box:

- ▶ Use XSR object as source
- ▶ Use column value as XML pattern source
- ▶ Use a document from file system

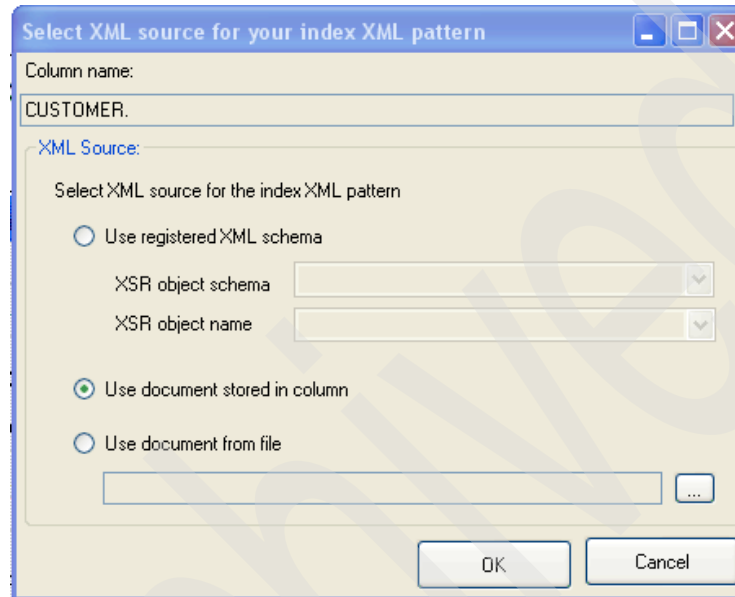


Figure 6-31 XML Pattern source dialog

Details for the options are as follows:

- ▶ Use registered XML schema:
Use this option if you want to use an XML Schema from XSR as the XML source.
- ▶ Use document from the column:
Use this option if the selected table contains at least one row and the XML column is already populated with an XML document.
- ▶ Use schema/XML document on disk:
Use this option if neither of the other options applies. Select file of type **xml** or **xsd** from your file system.

6.8.4 XQuery support in Visual Studio.NET

The IBM Script Designer allows you to execute relational SQL, SQLXML, or XQuery queries and view the results.

To access Script Designer, right-click the DB2 connection in Visual Studio's Server Explorer and then select **New Script**. This is shown in Figure 6-32.

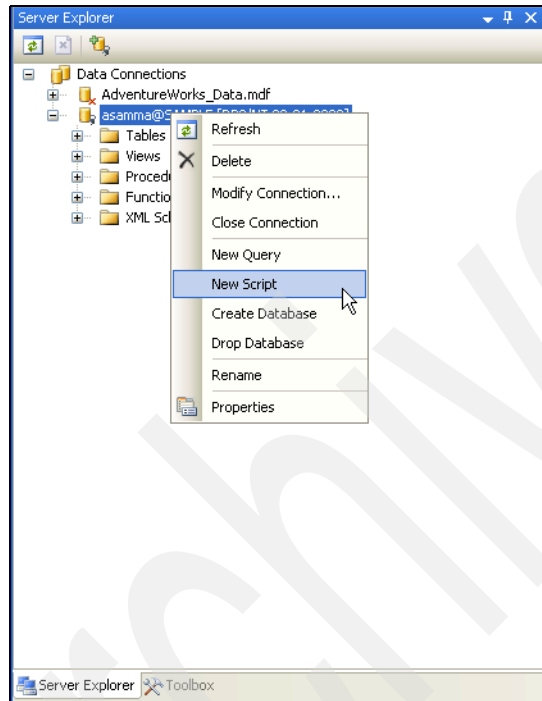


Figure 6-32 Accessing the Script Designer

Figure 6-33 shows Script Designer after it has been opened. In this figure, an example of an XQuery statement has been entered. From this tool it is possible to enter single or multiple SQL, SQLXML or Xqueries and return single or multiple result sets.

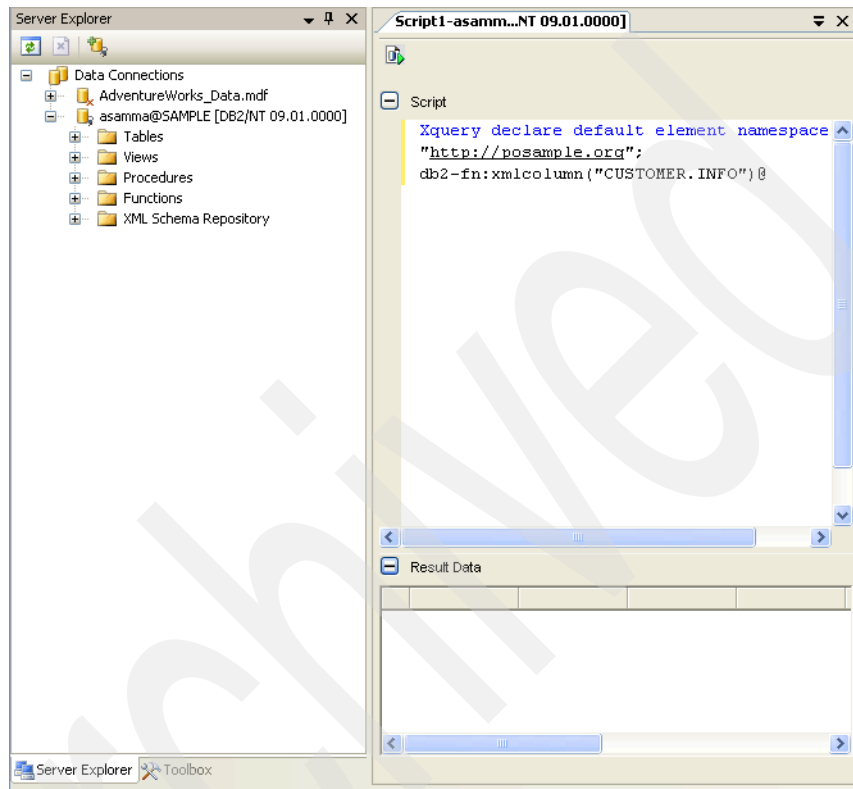


Figure 6-33 Script Designer opened with an XQuery entered

To execute the query, select **Execute Script**. The button for Execute Script is highlighted in red in Figure 6-34.

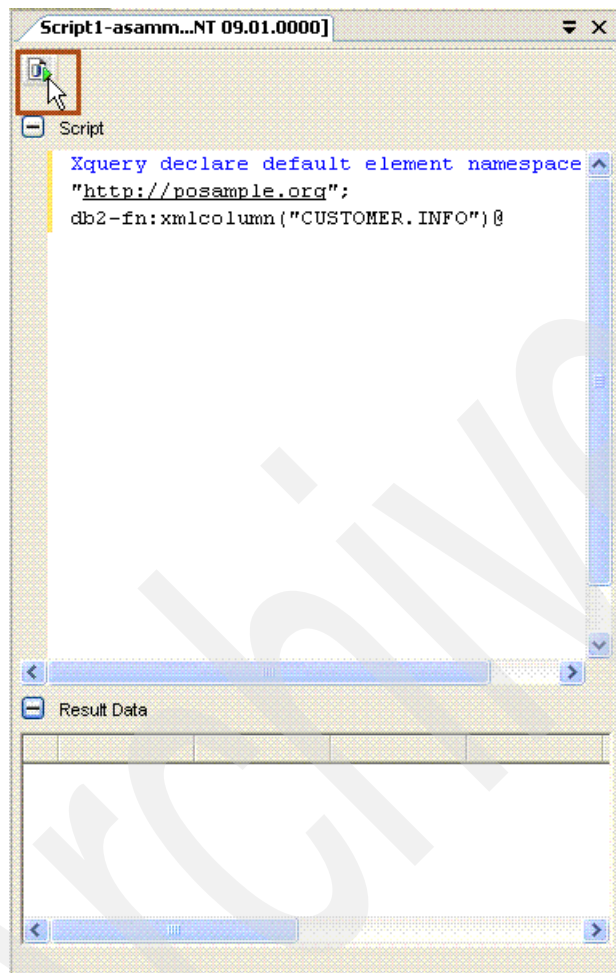


Figure 6-34 Execute the script

When the script has been executed, the results can be seen in the Result Data window. XML data will appear with an ellipsis (...). To view the data, either expand the column or click the ellipsis (...). See Figure 6-35.

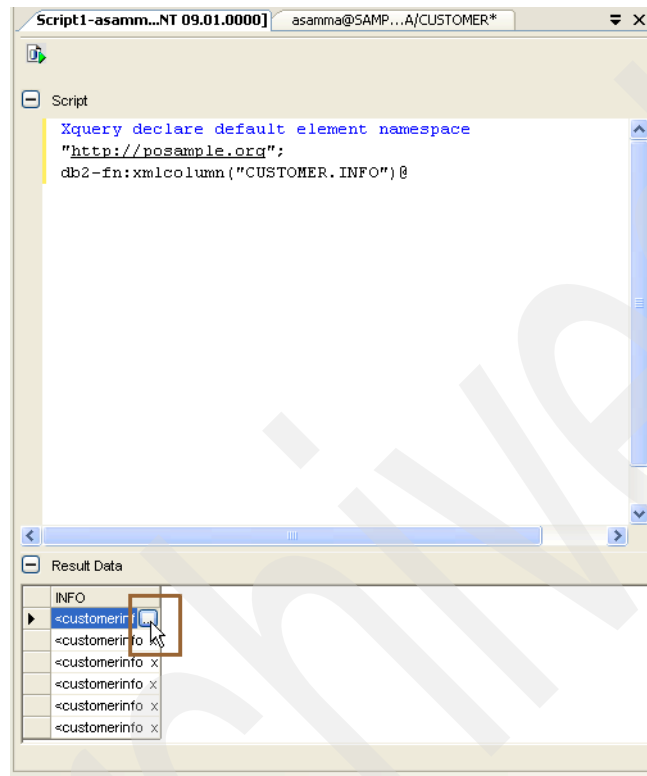


Figure 6-35 To view XML data expand the column or click the ellipsis

When you view the data by clicking the ellipsis, the HTML Visualizer window opens to the selected row, as shown in Figure 6-36.

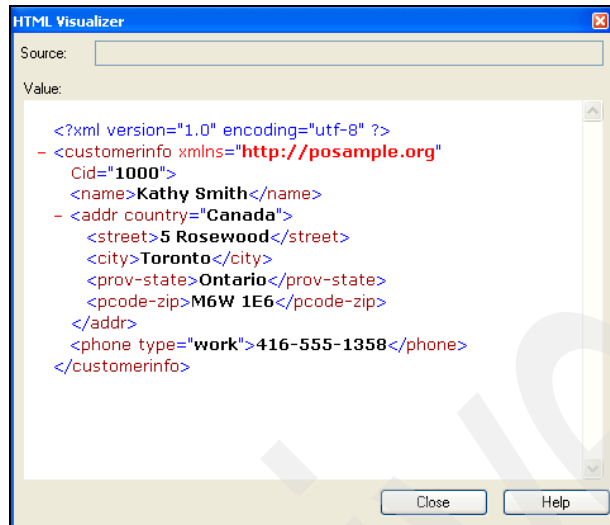


Figure 6-36 The HTML Visualizer opened to the selected row

6.9 XML and stored procedures

XML data can be passed to SQL procedures and external procedures by including parameters of data type XML in CREATE PROCEDURE parameter signatures.

Parameters of type XML are supported in:

- ▶ SQL procedures
- ▶ External procedures and external functions implemented in the following programming languages: C, C++, COBOL, Java, and .NET CLR

Variables of type XML are supported in:

- ▶ SQL procedures
- ▶ External procedures and external functions implemented in the following programming languages: C, C++, COBOL, Java, and .NET CLR

XML parameters and XML variables within procedures can be:

- ▶ Referenced in contexts including SQL statements where XML values are allowed

- ▶ Assigned to other variables using the following statements:
 - SELECT...INTO statement
 - VALUES...INTO statement
 - FETCH...INTO statement
 - CALL statement
 - EXECUTE ...INTO statement
 - SET statement

6.9.1 XML and XQuery support in SQL procedures

DB2 SQL procedures support parameters and variables of data type XML. They can be used in SQL statements in the same way as variables of any other data type. In addition, variables of data type XML can be passed as parameters to XQuery expressions in XMLEXISTS, XMLQUERY and XMLTABLE expressions.

Example 6-39 shows the declaration, use, and assignment of XML parameters and variables in a simple SQL procedure:

In this procedure, `simpleProc`, the following occurs:

- ▶ `var1` is inserted into column COL1 in table T1.
- ▶ An XML variable, `var1`, is declared.
- ▶ The value of the XML INPUT parameter, `parm1`, is checked to determine if it contains an item with a value less than 200. If so, the XML value is directly inserted into column COL1 in table T1.
- ▶ The value of parameter `parm2` is parsed using the **XMLPARSE** function and assigned to XML variable `var1`.
- ▶ The value of `var1` is then inserted into column COL1 in table T1.

Example 6-39 An SQL procedure using XML parameters

```

/* Assume table T1 exists with the following definition */
/* CREATE TABLE T1(col1 XML) */

CREATE PROCEDURE simpleProc (IN parm1 XML, IN parm2 VARCHAR(32000))
LANGUAGE SQL
BEGIN
DECLARE var1 XML;

/* check if the value of XML parameter parm1 contains an item with a
value less than 200 */
IF(XMLEXISTS('$x/ITEM[value < 200]' passing by ref parm1 as "x"))THEN
/* if it does, insert the value of parm1 into table T1 */
  INSERT INTO T1 VALUES(parm1);

```

```

END IF;

/* parse the parameter and assign it to the XML variable */
SET var1 = XMLPARSE(document parm preserve whitespace);

/* insert variable var1 into table T1 */
INSERT INTO T1 VALUES(var1);

END

```

Cursors for XQuery expressions in SQL procedures

SQL Procedures support the definition of cursors on XQuery expressions. Unlike cursors defined on SQL statements, which can be defined either statically or dynamically, *cursors on XQuery expressions can only be defined dynamically.*

To declare a cursor dynamically, it is necessary to:

- ▶ Declare a variable of type CHAR or VARCHAR to contain the XQuery expression using DECLARE statement.
- ▶ Prepare the XQuery expression before the cursor can be opened using PREPARE statement.

Example 6-40 is an example of an SQL procedure that dynamically declares a cursor for an XQuery expression, opens the cursor, and fetches XML data.

Example 6-40 Declaring a dynamic cursor for an XQuery expression

```

CREATE PROCEDURE my_Simple_XML_Proc_SQL()
RESULT SETS 1
LANGUAGE SQL
BEGIN
    DECLARE stmt_text VARCHAR (1024);
    DECLARE city VARCHAR(100);
    DECLARE stmt STATEMENT;
    DECLARE cur1 CURSOR WITH RETURN FOR stmt;

    SET city = 'Toronto';

    -- find out all the customers from Toronto
    SET stmt_text = 'XQUERY declare default element namespace
"http://posample.org"; for $cust in
db2-fn:xmlcolumn("CUSTOMER.INFO")/customerinfo/addr[city= '' || city
|| ''"] return <Customer>{$cust/../@Cid}{$cust/../name}</Customer>';

```

```

PREPARE stmt FROM stmt_text;

OPEN curl;

END

```

Note: When a commit or rollback is enacted during the execution of an SQL procedure, the values assigned to XML parameters and XML variables *will no longer be available*. After a commit or rollback, any attempt to reference these variables or parameters will cause an error (SQL1354N, 560CE) to be raised. To successfully reference XML parameters and variables after a commit or rollback, new values must be assigned to them.

The code segment in Example 6-41 shows a CLI stored procedure that utilizes an XQuery to return a result set to the caller.

Example 6-41 A CLI stored procedure utilizing XQuery to return a result set

```

SQL_API_RC SQL_API_FN my_simple_proc ( char sqlstate[6],
                                         char qualName[28],
                                         char specName[19],
                                         char diagMsg[71])
{
    SQLHANDLE henv;
    SQLHANDLE hdbc = 0;
    SQLHANDLE hstmt5;
    SQLRETURN cliRC;
    SQLCHAR stmt5[1024];
    SQLINTEGER custid,quantity,count;
    char city[100];

    cliRC = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
    SRV_HANDLE_CHECK(SQL_HANDLE_ENV, henv, cliRC, henv, hdbc);

    /* allocate the database handle */
    cliRC = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    SRV_HANDLE_CHECK(SQL_HANDLE_ENV, henv, cliRC, henv, hdbc);

    /* set AUTOCOMMIT off */
    cliRC = SQLSetConnectAttr(hdbc,
                              SQL_ATTR_AUTOCOMMIT,
                              SQL_AUTOCOMMIT_OFF,
                              SQL_NTS);
    SRV_HANDLE_CHECK(SQL_HANDLE_DBC, hdbc, cliRC, henv, hdbc);

```

```

/* issue NULL Connect, because in CLI a statement handle is
   required and thus a connection handle and environment handle.
   A connection is not established; rather the current
   connection from the calling application is used. */

/* connect to a data source */
cliRC = SQLConnect(hdbc, NULL, SQL_NTS, NULL, SQL_NTS, NULL,
SQL_NTS);
SRV_HANDLE_CHECK(SQL_HANDLE_DBC, hdbc, cliRC, henv, hdbc);

/* allocate the statement handle */
cliRC = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt5);
SRV_HANDLE_CHECK(SQL_HANDLE_DBC, hdbc, cliRC, henv, hdbc);

/* The query will find customers from Toronto... */
strcpy((char *)city, "Toronto");

/* XQuery to find all the customers from Toronto and return to
   caller */
strcpy((char *)stmt5, "XQUERY declare default element namespace "
                    "\"http://posample.org\"; for $cust in
db2-fn:xmlcolumn"
                    "\"(\"CUSTOMER.INFO\")/customerinfo/addr[city=\\"]";
                    strcat((char *)stmt5, city);
                    strcat((char *)stmt5, "\\"] return
<Customer>{$cust/../@Cid}{$cust/../name}</Customer>");

cliRC = SQLPrepare(hstmt5, stmt5, SQL_NTS);
SRV_HANDLE_CHECK_SETTING_SQLST_AND_MSG(SQL_HANDLE_STMT,
hstmt5,
cliRC,
henv,
hdbc,
sqlstate,
diagMsg,
"XQUERY statement
failed.");
cliRC = SQLExecute(hstmt5);
SRV_HANDLE_CHECK(SQL_HANDLE_STMT, hstmt5, cliRC, henv, hdbc);

...
return (0);
}

```

6.9.2 XML support in external routines

DB2 supports parameters and variables of data type XML in external procedures and functions written in the following programming languages:

- ▶ C
- ▶ C++
- ▶ CLI
- ▶ COBOL
- ▶ Java
- ▶ .NET CLR languages

Because XML data type values are represented in external routines in the same way as CLOB data types, the routines must specify that the XML data type is to be stored as a CLOB data type.

Note: The size of the CLOB value should be close to the size of the XML document represented by the XML parameter.

Example 6-42 shows a CREATE PROCEDURE statement for an external procedure implemented in the C. The statement shows the proper declaration of input and output XML parameters.

Example 6-42 A CREATE PROCEDURE statement for a C routine

```
CREATE PROCEDURE Simple_XML_Proc_C( IN inXML XML as CLOB(5000),  
                                   OUT outXML XML as CLOB(5000))  
  
LANGUAGE C  
PARAMETER STYLE SQL  
FENCED  
DYNAMIC RESULT SETS 1  
PARAMETER CCSID UNICODE  
EXTERNAL NAME 'simple_xmlproc!simple_proc'
```

XML and XQuery support in C# .NET CLR routines

When any C# routines in a file contain parameters or variables of type XML, it is required that the IBM.Data.DB2Types inclusion be specified. Example 6-43 illustrates this point.

Example 6-43 The IBM.Data.DB2Types inclusion

```
using System;  
using System.IO;  
using System.Data;  
using IBM.Data.DB2;
```

```

using IBM.Data.DB2Types;
    namespace bizLogic
    {
        class empOps
        { ...
        // C# procedures ...
        }
    }

```

XML data type values are represented in .NET routines in the same way as in other external routines, that is, the routines must specify that the XML data type is to be stored as a *CLOB* data type. Example 6-44 shows the correct parameter designation for input and output parameters of type XML in a CREATE PROCEDURE statement for a C# application.

Example 6-44 A CREATE PROCEDURE statement for a C# routine

```

CREATE PROCEDURE xmlProc1 ( IN inNUM INTEGER, IN inXML XML as CLOB
(1K), OUT inXML XML as CLOB (1K), OUT inXML XML as CLOB (1K) )
LANGUAGE CLR
PARAMETER STYLE GENERAL
DYNAMIC RESULT SETS 0
FENCED
THREADSAFE
DETERMINISTIC
NO DBINFO
MODIFIES SQL DATA
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenProc.dll:bizLogic.empOps!xmlProc1' ;

```

XML and XQUERY support in Java

When any Java routines in a source file contain parameters or variables of type XML, the `com.ibm.db2.jcc.DB2Xml` import is required. Example 6-45 illustrates this point.

Example 6-45 Importing the `com.ibm.db2.jcc.DB2Xml`

```

using System;
import java.lang.*;
import java.io.*;
import java.sql.*;
import java.util.*;
import com.ibm.db2.jcc.DB2Xml;
public class stpclass
{ ...

```

```
// Java procedure implementations ...  
}
```

XML data type values are represented in JAVA routines in the same way as in other external routines. That is, the routines must specify that the XML data type is to be stored as a CLOB data type. Example 6-46 shows the correct parameter designation for input and output parameters of type XML.

Example 6-46 Using XML input and output parameters

```
CREATE PROCEDURE xmlProc1 ( IN inNUM INTEGER, IN inXML XML as CLOB  
(1K), OUT out1XML XML as CLOB (1K), OUT out2XML XML as CLOB (1K) )  
DYNAMIC RESULT SETS 0  
DETERMINISTIC  
LANGUAGE JAVA  
PARAMETER STYLE JAVA  
MODIFIES SQL DATA  
FENCED  
THREADSAFE  
PROGRAM TYPE SUB  
NO DBINFO  
EXTERNAL NAME 'myJar:stpclass.xmlProc1'@
```

Invocation of routines with XML parameters in Java applications

When you call a stored procedure that has XML parameters, a compatible data type must be used in the invoking statement.

For JDBC applications, when calling a routine with XML input parameters, use parameters of the `com.ibm.db2.jcc.DB2Xml` type.

To register XML output parameters, use parameters as the `com.ibm.db2.jcc.DB2Types.XML` type.

The code segment in Example 6-47 calls a stored procedure, `SP_xml`, with one input and two output parameters:

Example 6-47 A Java Stored Procedure call with one input and two output parameters

```
// Declare nput, output, and inout parameters  
com.ibm.db2.jcc.DB2Xml in_xml = xmlvar;  
com.ibm.db2.jcc.DB2Xml out_xml = null;  
com.ibm.db2.jcc.DB2Xml inout_xml = xmlvar;  
...
```

```
Connection con; CallableStatement cstmt;  
ResultSet rs;
```

```

...
// Create a CallableStatement object
cstmt = con.prepareCall("CALL SP_xml(?,?,?)");

// Set input parameter as type com.ibm.db2.jcc.DB2Xml
cstmt.setObject (1, in_xml);

// Register output parms as type com.ibm.db2.jcc.DB2Types.XML
cstmt.registerOutParameter (2, com.ibm.db2.jcc.DB2Types.XML);
cstmt.registerOutParameter (3, com.ibm.db2.jcc.DB2Types.XML);

// Call the stored procedure
cstmt.executeUpdate();
System.out.println("Parameter values from SP_xml call: ");

System.out.println("Output parameter value ");

// Use the DB2-only method getBytes to
// convert the value to bytes for printing
printBytes(out_xml.getDB2String());

System.out.println("Input/output parameter value ");
printBytes(inout_xml.getDB2String());

...

```

When you call a stored procedure that has XML parameters, a compatible data type must be used in the invoking statement.

For JDBC applications, when calling a routine with XML *input* parameters, use parameters of the *com.ibm.db2.jcc.DB2Xml* type.

To register XML *output* parameters, use parameters as the *com.ibm.db2.jcc.DB2Types.XML* type.

For additional information regarding the retrieval of output parameters, refer to “Retrieving XML data in JDBC applications” on page 307.

For considerations of retrieving output parameters when invoking Java/SQLJ stored procedures, refer to “Retrieving XML data in SQLJ applications” on page 313.

Example 6-48 demonstrates the invocation of a Java stored procedure that has two *XML* type INPUT parameters and one *INTEGER* OUTPUT parameter.

```
public static void callSimple_Proc(Connection con)
{
    try
    {
        // prepare the CALL statement
        String procName = "Simple_XML_Proc_Java";
        String sql = "CALL " + procName + "( ?, ?, ?)";

        CallableStatement callStmt = con.prepareCall(sql);

        // input data
        String inXml = "<customerinfo xmlns=\"http://posample.org\"
Cid=\"5002\">"
+ "<name>Kathy Smith</name><addr country=\"Canada\"><street>25
EastCreek"
+ "</street><city>Markham</city><prov-state>Ontario</prov-state><pcode-z
ip>"
+ "N9C-3T6</pcode-zip></addr><phone type=\"work\">905-566-7258"
+ "</phone></customerinfo>";

        callStmt.setString (1, inXml ) ;

        // register the output parameters
        // the XML output parm is registered as com.ibm.db2.jcc.DB2Types.XML
        type
        callStmt.registerOutParameter(2, com.ibm.db2.jcc.DB2Types.XML);
        callStmt.registerOutParameter(3, Types.INTEGER);

        // call the stored procedure
        System.out.println();
        System.out.println("Calling stored procedure " + procName);
        callStmt.execute();
        System.out.println(procName + " called successfully");

        // retrieve output parameters using type com.ibm.db2.jcc.DB2Xml
        // The
        com.ibm.db2.jcc.DB2Xml outXML = (DB2Xml) callStmt.getObject(2);
        System.out.println("\n \n Location is :\n "
            + outXML.getDB2String());
        ResultSet rs = callStmt.getResultSet();

        Fetch...
```

To call a routine with XML parameters from an SQLJ program, use parameters of the `com.ibm.db2.jcc.DB2Xml` type. Example 6-49 shows an SQLJ program that calls a stored procedure that takes three XML parameters: an IN parameter, an OUT parameter, and an INOUT parameter.

Example 6-49 Call a routine from an SQLJ program

```
com.ibm.db2.jcc.DB2Xml in_xml = xmlvar;
com.ibm.db2.jcc.DB2Xml out_xml = null;
com.ibm.db2.jcc.DB2Xml inout_xml = xmlvar;
// Declare an input, output, and
// input/output XML parameter
...
#sql [myConnCtx] {
CALL SP_xml(:IN in_xml, :OUT out_xml, :INOUT inout_xml)
};
// Call the stored procedure
System.out.println("Parameter values from SP_xml call: ");
System.out.println("Output parameter value ");
printBytes(out_xml.getDB2String());
// Use the DB2-only method getBytes to
// convert the value to bytes for printing
System.out.println("Input/output parameter value ");
printBytes(inout_xml.getDB2String());
```

6.9.3 XML Schema Repository object registration

An XML schema, DTD, or external entity must be *registered* with the XML Schema repository (XSR), before it can be used for validation and annotation. XSR object registration involves the following steps:

1. Register the XML schema document in the XML schema repository.
2. Specify additional XML schema documents to be included with the XSR object (required only if your XML schema consists of more than one schema document).
3. Complete the registration process with the XML schema repository.

When a DB2 database is created, the stored procedures required to register an XML schema are also created. They are:

- ▶ XSR_REGISTER procedure
- ▶ XSR_ADDSCHEMADOC procedure
- ▶ XSR_COMPLETE procedure
- ▶ XSR_DTD procedure
- ▶ XSR_EXTENTITY procedure

The XSR object registration steps can be performed by any of the following methods:

- ▶ Stored procedures
- ▶ Command line processor
- ▶ Java applications

Java support for XML schema registration and removal

As indicated previously, DB2 provides the SYSPROC.XSR_REGISTER, SYSPROC.XSR_ADDSCHEMADOC, SYSPROC.XSR_COMPLETE, and SYSPROC.XSR_REMOVE stored procedures for registering and removing XML schemas and components.

The IBM DB2 Driver for JDBC and SQLJ provides *methods* that let you perform the same functions from a Java application program. Those methods are:

- ▶ `DB2Connection.registerDB2XMLSchema`
Registers an XML schema in DB2, using one or more XML schema documents. There are two forms of this method: one form for XML schema documents that are input from an `InputStream` objects, and one form for XML schema documents that are in a `Strings`.
- ▶ `DB2Connection.deregisterDB2XMLObject`
Removes an XML schema definition from DB2. Before you can invoke these methods, the underlying stored procedures must be installed on the DB2 database server.

6.10 Web services

Web services are self-describing and modular applications that expose business logic as services that can be published, discovered, and invoked over the Internet. It is a technology that is well-suited to implementing a *service-oriented architecture* (SOA).

Based on XML standards, Web services can be developed as *loosely-coupled* application components using any programming language, any protocol, or any platform. This mode of development facilitates the delivery of business applications as a service accessible to anyone, anytime, at any location, and using any platform. Of course, Web services are not the only technology that can be used to implement an SOA. Many examples of organizations that have successfully implemented SOAs using other technologies can be found. Web services have also been used by others to implement architectures that are not service-oriented.

Web services and SOAs are dedicated to reducing or eliminating impediments to interoperable integration of applications, regardless of their operating system platform or language of implementation. The following list summarizes and highlights the most compelling characteristics of Web services and SOA:

► **Componentization:**

SOA encourages an approach to systems development in which software is encapsulated into components called *services*. Services interact through the exchange of messages that conform to published interfaces. The interface supported by a service is all that concerns any prospective consumers; implementation details of the service itself are hidden from all consumers of the service.

► **Platform independence:**

In an SOA, the implementation details are hidden. Therefore, services can be combined and orchestrated regardless of programming language, platform, and other implementation details. Web services provide access to software components through a wide variety of transport protocols, increasing the number of channels through which software components can be accessed.

► **Investment preservation:**

As a benefit of componentization and encapsulation, existing software assets can be exposed as services within an SOA using Web services technologies. When existing software assets are exposed in this way, they can be extended, refactored, and adapted into appropriate services to participate within an SOA. This reuse reduces costs and preserves the investment. The evolutionary approach enabled by Web services eliminates the necessity to rip and replace existing solutions.

► **Loose coupling:**

As another benefit of componentization, the SOA approach encourages loose coupling between services, which is a reduction of the assumptions and requirements shared between services. Implementations of individual services can be replaced and evolved over time without disrupting the normal activities of the SOA system as a whole. Therefore, loosely coupled systems tend to reduce overall development and maintenance costs by isolating the impact of changes to the internal implementation of components and encouraging reuse of components.

► **Distributed computing standardization:**

Web services are the focal point of many, if not most, of the current standardization initiatives related to advancement of distributed computing technology. Additionally, much of the computer industry's research and development effort related to distributed computing is centered on Web services.

- Broad industry support:

Core Web services standards (SOAP, WSDL, XML, and XML Schema) are universally supported by all major software vendors. This universal support provides a broad choice of middleware and tooling products with which to build service-oriented applications.

- Composability:

Web services technologies are planned to enable designers to mix and match different capabilities through composition. For example, systems that require message-level security can leverage the Web services Security standard. Any system that does not require message-level security is not forced to deal with the complexity and overhead of signing and encrypting its messages.

This approach to composability applies to all the various qualities of service, such as reliable delivery of messages and transactions. Composability enables Web services technologies to be applied consistently in a broad range of usage scenarios, such that only the required functionality has to be implemented.

6.10.1 Components of Web Services

Several key technologies and standards exist within the Web services community. The most common and widely accepted Web services standards are Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL), and Universal Description, Discovery and Integration (UDID). SOAP is used as a communication protocol. WSDL is used for describing a construction of Web Services. UDDI is a registry of Web Services.

SOAP

SOAP is a simple, flexible, and extendable mechanism for exchanging structured data. Web Services uses SOAP as a communication protocol. Different from HTTP protocol which uses text strings for GET/POST methods and URL, SOAP is an XML-based messaging protocol. SOAP encodes messages as XML documents for sending requests and receiving responses.

SOAP consists of two parts:

- Protocol binding header:

SOAP can use HTTP, SMTP and FTP as the underneath protocol. The SOAP library generates the protocol binding header based on the protocol specified. When a Web server reads a protocol binding header, it understands that the following message is a SOAP message.

► SOAP envelope:

A SOAP envelope contains a header and a body. A SOAP header is optional, and contains information such as security information and routing information. A SOAP body contains call and response information. It includes method names and arguments if it is a Remote Procedure Call (RPC).

Example 6-50 shows a sample SOAP message:

Example 6-50 A SOAP message

(1) Protocol Binding Header

```
POST /services/weather/QueryWeather.dadx/SOAP HTTP/1.0
Host: localhost
Content-Type: text/xml; charset=utf-8
SOAPAction: http://tempuri.org/weather/QueryWeather.dadx
```

(2) SOAP Envelope

```
<?xml version="1.0" encoding="UTF-8" ?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
```

(a) SOAP Header

```
<soap:Header>
  <t:Transaction xmlns:t="http://tempuri.org/transaction"
    soap:mustUnderstand="1">
    5
  </t:Transaction>
</soap:Header>
```

(b) SOAP Body

```
<soap:Body>
  <m:getWeather xmlns:m="http://tempuri.org/weather/QueryWeather.dadx">
    <wDate xsi:type="xsd:date">2003-02-25</wDate>
    <prefName xsi:type="xsd:string">TOKYO</prefName>
  </m:getWeather>
</soap:Body>

</soap:Envelope>
```

WSDL

WSDL is a standardized XML interface description used to define a Web Service interface. The interface includes the information about how to structure content request messages, how to interpret response messages, and which transport protocol to use to invoke the Web service.

The Web Services provider provides an interface description. Based on the description, the Web Services requesters create applications to request Web services.

UDDI

UDDI is an open framework for describing, publishing, and finding Web services on the Internet. UDDI is similar to a phone book where companies can list the Web services they provide. Web Services requesters can search for UDDI to locate the Web Service they require.

6.10.2 Web services in DB2 9

DB2 provides optimized supports for Web services since Version 8. DB2 9 pureXML features, native XML store, SQL/XML, XQuery, and XPath supports enrich DB2 Web services even more. DB2 users can take advantages of Web services in two ways: as a provider and as a consumer or requestor.

Figure 6-37 illustrates Web services functionality in DB2 9.

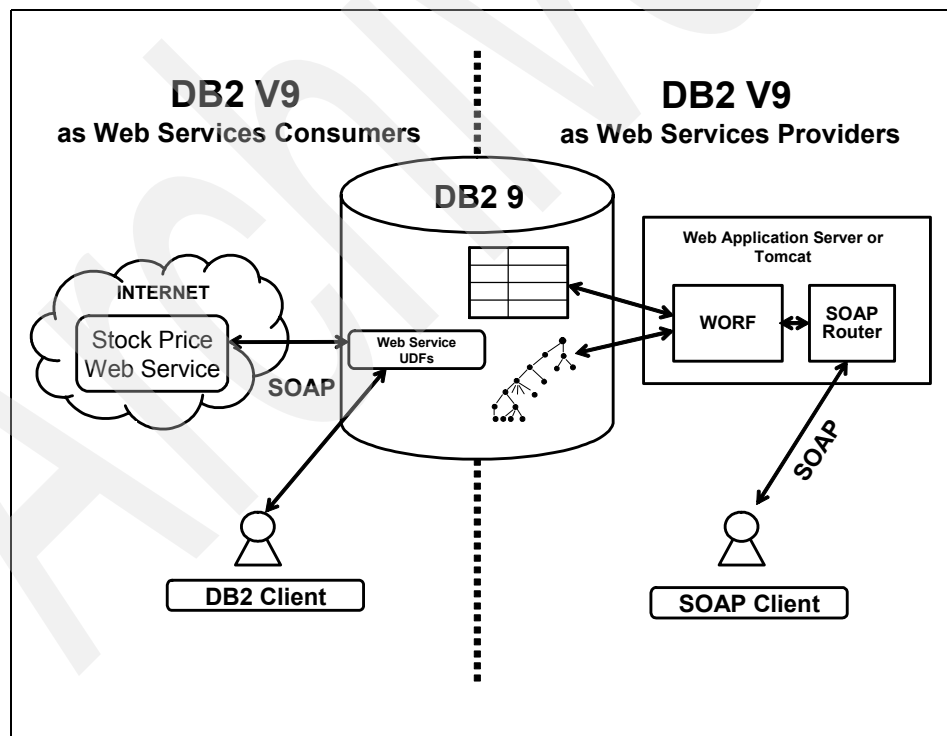


Figure 6-37 Web services in DB2 9 overview

Web services consumer

When utilizing DB2 9 as the Web services consumer, application developers can save the programming effort by using SQL to access Web service data. The data can be manipulated within the context of an SQL statement before that data is returned to the client application. DB2 provides UDFs to facilitate applications consuming and integrating Web services data.

Example 6-51 shows three SOAP UDFs that can be used to send SOAP requests and receive SOAP responses. These UDFs perform the same functions but return values in different type and length. The UDFs do the following actions:

- ▶ They compose a SOAP request.
- ▶ They post the request to the service endpoint.
- ▶ They receive the SOAP response.
- ▶ They return the content of the SOAP body.

These three UDFs have three arguments to be specified as follows:

- ▶ The first argument is the endpoint where the utilized Web service is running.
- ▶ The second argument is the SOAP action (if required).
- ▶ The third argument is the SOAP body where the method name and arguments for that Web service are specified.

Example 6-51 SOAP UDFs

```
db2xml.soaphttpv (  
    endpoint_url VARCHAR(256),  
    soap_action VARCHAR(256),  
    soap_body VARCHAR(3072)) | CLOB(1M))  
RETURNS VARCHAR(3072)
```

```
db2xml.soaphttpc (  
    endpoint_url VARCHAR(256),  
    soapaction VARCHAR(256),  
    soap_body VARCHAR(3072) | CLOB(1M))  
RETURNS CLOB(1M)
```

```
db2xml.soaphttpcl(  
    endpoint_url VARCHAR(256),  
    soapaction VARCHAR(256),  
    soap_body varchar(3072))  
RETURNS CLOB(1M) as locator
```

Example 6-52 shows the coding for invoking a SOAP UDF. Note that we have specified the method name *getLastName* and the argument *wEmpno* for that method as the third argument.

Example 6-52 an example of SOAP UDF

```
values db2xml.soaphttpv(

-- (1) ENDPOINT
'http://localhost:8080/services/emp/getLastName.dadx/SOAP',

-- (2) ACTION
'http://tempuri.org/emp/getLastName.dadx',

-- (3) SOAP BODY
'<m:getLastName xmlns:m="http://tempuri.org/emp/getLastName.dadx">
  <wEmpno xsi:type="xsd:string">000100</wEmpno>
</m:getWeather>Ae);
)
```

Web services provider

DB2 9 also can be the Web service provider. DB2 9 offers the Web services client application the ability to manipulate data inside the database through the WSDL interface. You can create a WSDL interface to access DB2 9 data by using the Web services Object Runtime Framework (WORF). Inside WORF, the Document Access Definition Extension (DADX) file is used to specify the services. In DADX files, you define the operation to access DB2 data. The DADX and its runtime environment then are deployed to a supported Web server for users to use the DB2 Web service.

Figure 6-38 shows an overview of the DB2 Web service provider architecture. The SOAP client calls a Web service (in Figure 6-38, *getName*). WORF looks into the DADX file to find the *getName* method and the SQL associated with it. The DB2 data retrieved by the SQL then is returned with a SOAP message.

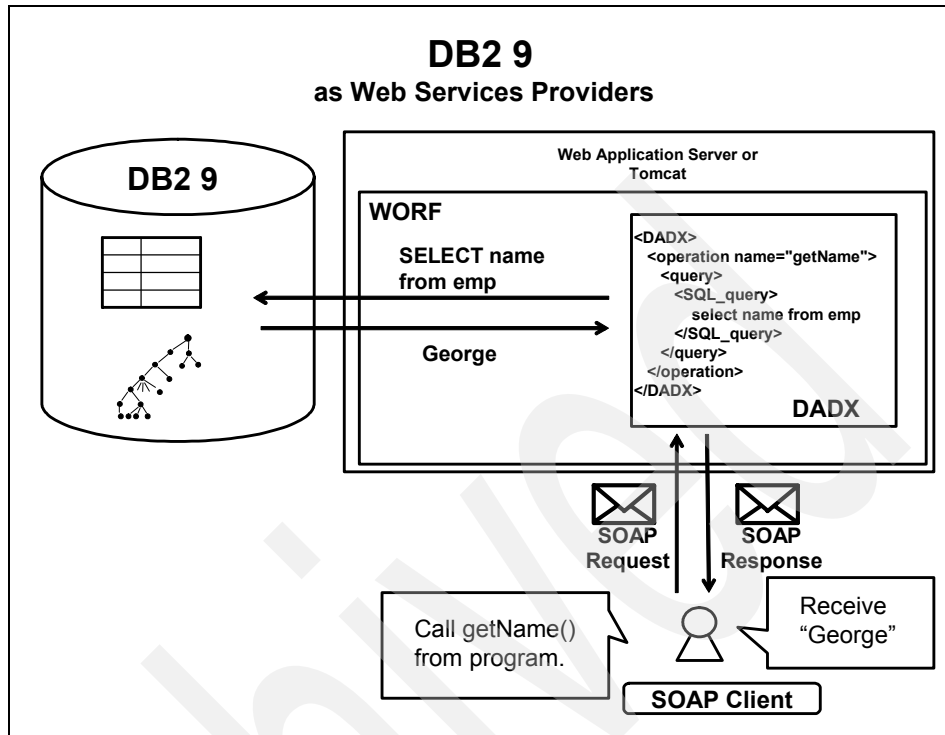


Figure 6-38 Web Services Provider (WORF) architecture overview

WORF

The *Web services Object Runtime Framework* (WORF) provides a user-friendly environment for creating simple XML based Web services to access DB2 data and stored procedures. The WORF application environment works on WebSphere Application Server and Apache Tomcat. By using the framework, the application developers can avoid writing programs to handle the details of creating the Web services.

The easy-to-use programming environment provided by WORF includes functions for the application to connect to the database, execute SQL statements, and call the stored procedures. You also can use WORF to generate WSDL, test pages, and XML Schema for Web services. In addition, WORF provides an automatic documentation feature and resource-based deployment. For constructing a simple WORF Web service, you only have to edit the DADX file and property file group.properties.

DADX file

DADX is an XML file describing Web service definitions. When the Web application server receives a SOAP request, WOF reads the DADX file to discern the method name that is being called, then executes the SQL statements or stored procedures corresponding to that method.

Example 6-53 shows a DADX file, in which a Web service method name, an argument, and a SELECT statement are defined. The `getLastName` method takes an employee number as an argument and returns the last name of the employee whose employee number matches with the argument.

Example 6-53 getLastName.dadx

```
<?xml version="1.0"?>
<DADX xmlns="http://schemas.ibm.com/db2/dxx/dadx"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:dtd1="http://schemas.myco.com/sales/getstart.dtd"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

  <operation name="getLastName">
    <wsdl:documentation>uranai</wsdl:documentation>
    <query>
      <SQL_query>
        select lastname from employee where empno=:wEmpno
      </SQL_query>
      <parameter name="wEmpno" type="xsd:string"/>
    </query>
  </operation>
</DADX>
```

Example 6-54 shows another use of DADX. This `doXQuery` method returns the result set of an SQL/XML query.

Example 6-54 doXQuery.dadx

```
<?xml version="1.0"?>
<DADX xmlns="http://schemas.ibm.com/db2/dxx/dadx"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:dtd1="http://schemas.myco.com/sales/getstart.dtd"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

  <operation name="doXQuery">
    <wsdl:documentation>uranai</wsdl:documentation>
    <query>
      <SQL_query>
```

```

SELECT xmlserialize(xmlquery('$c/Application/Customer/Name' passing
APPL_DOC as "c") as varchar(128))
FROM DB2ADMIN.LOAN_APPLICATION
WHERE xmlexists('$i/Application/Customer/Name[FirstName = "Ippei" or
FirstName="Ichiro" ]' passing APPL_DOC as "i")
    </SQL_query>
    </query>
  </operation>
</DADX>

```

group.properties file

In the group.properties file there is information that WORF must have to access DB2, such as the JDBC driver, database name, user ID, password.

Example 6-55 shows a group.properties file for getLastName.dadx.

Example 6-55 group.properties

```

# /dadx group properties
dbDriver=COM.ibm.db2.jdbc.app.DB2Driver
dbURL=jdbc:db2:sample
userID=xxxxx
password=xxxxxx
parserClass=org.apache.xerces.parsers.SAXParser
autoReload=true
reloadIntervalSeconds=5
initialContextFactory=com.ibm.websphere.naming.WsnInitialContextFactory
datasourceJNDI=jdbc/sample
groupNamespaceUri=http://schemas.ibm.com/employee

```

Testing WORF Web services

WORF comes with a testing framework so that you can easily verify your Web services.

When the methods are deployed, you can test the methods and acquire the WSDL and XML Schema. Now we have deployed two methods: getLastName and doXQuery. Figure 6-39 shows a test page of WORF after getLastName and doXQuery are deployed.

XML Redbook - Sample Web Services

- List the deployed services using the [SOAP Admin](#) page.
- Test the HTTP POST binding.
- View the automatically generated WSDL and XSD.

Sample Web Services					
Get Last Name From Employee	TEST	WSDL	WSDLservice	WSDLbinding	XSD
Do XQuery	TEST	WSDL	WSDLservice	WSDLbinding	XSD

Figure 6-39 WORF test page

Figure 6-40 shows a page resulting from executing the doXQuery methods. When you click the **Invoke** button in the right pane, the doXQuery method is executed and the result is displayed on the bottom pane. Note that the result set from the SELECT statement in the doXQuery.dadx is wrapped in the SOAP envelope.

The screenshot displays a web browser window with the address `http://localhost:8080/services/xmlrb/doXQuery.dadx/TEST`. The interface is divided into three main sections:

- Methods:** Lists the `doXQuery` method under the `http://tempuri.org/xmlrb/doXQuery.dadx Web Service`. A note states: "There is no documentation available for this service."
- Inputs:** Shows the `doXQuery Web Method` with a text input field containing `uranai` and an `Invoke` button.
- Result:** Displays the XML response wrapped in a SOAP envelope. The XML structure is as follows:

```
<?xml version="1.0" ?>
<xsd1:doXQueryResponse xmlns:xsd1="http://schemas.ibm.com/xmlrb/doXQuery.dadx/XSD"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <return>
    <xsd1:doXQueryResult xmlns:xsd1="http://schemas.ibm.com/xmlrb/doXQuery.dadx/XSD"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <doXQueryRow>
        <_x0031_><Name><FirstName>Ippei</FirstName><LastName>Komi</LastName></Name></_x0031_>
      </doXQueryRow>
      <doXQueryRow>
        <_x0031_><Name><FirstName>Ichiro</FirstName><LastName>Ohta</LastName></Name></_x0031_>
      </doXQueryRow>
    </xsd1:doXQueryResult>
  </return>
</xsd1:doXQueryResponse>
```

Figure 6-40 WORF test result page

Sample data

This appendix provides the following sample materials:

- ▶ DDLs for creating database and tables used in sample application XMLoan.
- ▶ The sample XML data used in Chapter 5, “Managing XML data” on page 173.

A.1 Creating XMLoan database

This section provides the DDLs for the FAMDI bank loan application described in Chapter 2, “Sample scenario description” on page 21.

A.1.1 Creating database

XML data is stored in code set UTF-8/code page 1208 on DB2 V9.1. In order to use the XML type, you must create a UTF-8 database.

Example A-1 shows creating a UTF-8 database called xmlrb.

Example: A-1 create database command

```
create database xmlrb using codeset UTF-8 territory US
```

A.1.2 Creating tables

There are seven tables. The data model in Chapter 2, “Sample scenario description” on page 21 shows the relationship among them. Following are the table names and brief descriptions of the tables:

- ▶ The APPLICATION_STATUS table has static data of an application; for instance, new, in progress, accepted, or rejected.
- ▶ The LOAN_APPLICATION table contains customers' loan applications. The loan application is stored in an XML form.
- ▶ The CAMPAIGN table stores the campaigns for loan products; for instance, a campaign can be TV, radio, or newspaper.
- ▶ The LOAN table has the loan information; for instance, starting date of the loan.
- ▶ The PAYMENT table stores the customer's payments.
- ▶ The PRODUCT table stores all the loan products.
- ▶ The FEEDBACK table stores the customer's feedback. The customer can submit feedback on-line. The feedback is stored in an XML form.

Creating tables and primary keys

The following examples show creating tables and primary keys.

Example A-2 shows creating table APPLICATION_STATUS, and STATUS_ID is the primary key.

Example: A-2 Create table APPLICATION_STATUS

```
CREATE TABLE "DB2ADMIN"."APPLICATION_STATUS" (  
    "STATUS_ID" INTEGER NOT NULL ,  
    "STATUS_DESC" CHAR(50) NOT NULL );  
  
ALTER TABLE "DB2ADMIN"."APPLICATION_STATUS"  
    ADD CONSTRAINT "CC1155357143051" PRIMARY KEY  
    ("STATUS_ID");
```

Example A-3 shows creating LOAN_APPLICATION, and APPL_ID is the primary key. The column APPL_DOC has an XML type.

Example: A-3 Create table LOAN_APPLICATION

```
CREATE TABLE "DB2ADMIN"."LOAN_APPLICATION" (  
    "APPL_ID" BIGINT NOT NULL GENERATED ALWAYS AS IDENTITY (  
        START WITH +0  
        INCREMENT BY +1  
        MINVALUE +0  
        MAXVALUE +9223372036854775807  
        NO CYCLE  
        NO CACHE  
        NO ORDER ) ,  
    "APPL_DOC" XML ,  
    "APPL_STATUS" INTEGER NOT NULL WITH DEFAULT 1 ,  
    "PROD_ID" INTEGER WITH DEFAULT NULL );  
  
ALTER TABLE "DB2ADMIN"."LOAN_APPLICATION"  
    ADD CONSTRAINT "CC1155685528218" PRIMARY KEY  
    ("APPL_ID");
```

Example A-4 shows creating table CAMPAIGN, and CAMP_ID is the primary key.

Example: A-4 Create table CAMPAIGN

```
CREATE TABLE "DB2ADMIN"."CAMPAIGN" (  
    "CAMP_ID" INTEGER NOT NULL ,  
    "CAMP_DESC" CHAR(30) NOT NULL );  
  
ALTER TABLE "DB2ADMIN"."CAMPAIGN"  
    ADD CONSTRAINT "CC1155685220956" PRIMARY KEY  
    ("CAMP_ID");
```

Example A-5 shows creating table LOAN, and LOAN_ID is the primary key.

Example: A-5 Create table LOAN

```
CREATE TABLE "DB2ADMIN"."LOAN" (  
    "START_DATE" DATE NOT NULL ,  
    "LOAN_ID" INTEGER NOT NULL ,  
    "PYMT_STATUS" CHAR(10) ,  
    "PYMT_COUNT" INTEGER );  
  
ALTER TABLE "DB2ADMIN"."LOAN"  
    ADD CONSTRAINT "CC1155686490131" PRIMARY KEY  
    ("LOAN_ID");
```

Example A-6 shows creating table PAYMENT.

Example: A-6 Create table command for table PAYMENT

```
CREATE TABLE "DB2ADMIN"."PAYMENT" (  
    "APPL_ID" INTEGER NOT NULL ,  
    "PYMT_DATE" DATE NOT NULL );
```

Example A-7 shows creating table PRODUCT, and PROD_ID is the primary key.

Example: A-7 Create table PRODUCT

```
CREATE TABLE "DB2ADMIN"."PRODUCT" (  
    "PROD_ID" INTEGER NOT NULL ,  
    "PROD_DESC" CHAR(50) NOT NULL ,  
    "RATE" DECIMAL(6,3) ,  
    "AMOUNT" DECIMAL(14,2) ,  
    "TERM" INTEGER );  
  
ALTER TABLE "DB2ADMIN"."PRODUCT"  
    ADD CONSTRAINT "CC1155685582066" PRIMARY KEY  
    ("PROD_ID");
```

Example A-8 shows creating table FEEDBACK. The column COMMENT has an XML type.

Example: A-8 Create table command for FEEDBACK

```
CREATE TABLE "DB2ADMIN"."FEEDBACK" (  
    "APPL_ID" INTEGER NOT NULL ,  
    "COMMENT" XML );
```

Creating foreign keys

Foreign keys provide a way to define relationships among tables. The following examples show the commands to define foreign keys.

Example A-9 shows that column APPL_STATUS in table LOAN_APPLICATION is a foreign key that refers to column STATUS_ID in table APPLICATION_STATUS. The column PROD_ID in table LOAN_APPLICATION is a foreign key that refers to PROD_ID in table PRODUCT.

Example: A-9 Define foreign keys for table LOAN_APPLICATION

```
ALTER TABLE "DB2ADMIN"."LOAN_APPLICATION"
  ADD CONSTRAINT "CC1155687456882" FOREIGN KEY
    ("APPL_STATUS")
  REFERENCES "DB2ADMIN"."APPLICATION_STATUS"
    ("STATUS_ID")
  ON DELETE NO ACTION
  ON UPDATE NO ACTION
  ENFORCED
  ENABLE QUERY OPTIMIZATION;

ALTER TABLE "DB2ADMIN"."LOAN_APPLICATION"
  ADD CONSTRAINT "CC1155687530557" FOREIGN KEY
    ("PROD_ID")
  REFERENCES "DB2ADMIN"."PRODUCT"
    ("PROD_ID")
  ON DELETE NO ACTION
  ON UPDATE NO ACTION
  ENFORCED
  ENABLE QUERY OPTIMIZATION;
```

Example A-10 shows that column LOAN_ID in table LOAN is a foreign key that refers to column APPL_ID in table LOAN_APPLICATION.

Example: A-10 Define foreign key for table LOAN

```
ALTER TABLE "DB2ADMIN"."LOAN"
  ADD CONSTRAINT "CC1155686626047" FOREIGN KEY
    ("LOAN_ID")
  REFERENCES "DB2ADMIN"."LOAN_APPLICATION"
    ("APPL_ID")
  ON DELETE NO ACTION
  ON UPDATE NO ACTION
  ENFORCED
  ENABLE QUERY OPTIMIZATION;
```

Example A-11 shows that column APPL_ID in table PAYMENT is a foreign key that refers to column LOAN_ID in table LOAN.

Example: A-11 Define foreign key for table PAYMENT

```
ALTER TABLE "DB2ADMIN"."PAYMENT"  
  ADD CONSTRAINT "CC1155686683650" FOREIGN KEY  
    ("APPL_ID")  
  REFERENCES "DB2ADMIN"."LOAN"  
    ("LOAN_ID")  
  ON DELETE NO ACTION  
  ON UPDATE NO ACTION  
  ENFORCED  
  ENABLE QUERY OPTIMIZATION;
```

Example A-12 shows that column APPL_ID in table FEEDBACK is a foreign key that refers to column APPL_ID in table APPLICATION_APPLICATION.

Example: A-12 Define foreign key for table FEEDBACK

```
ALTER TABLE "DB2ADMIN"."FEEDBACK"  
  ADD CONSTRAINT "CC1155687388774" FOREIGN KEY  
    ("APPL_ID")  
  REFERENCES "DB2ADMIN"."LOAN_APPLICATION"  
    ("APPL_ID")  
  ON DELETE NO ACTION  
  ON UPDATE NO ACTION  
  ENFORCED  
  ENABLE QUERY OPTIMIZATION;
```

A.2 contactInfo.xsd

Example A-13 shows the XML schema used in 5.3.1, "IMPORT" on page 208.


```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="work" type="xsd:string"/>
  <xsd:element name="mobile" type="xsd:string"/>
  <xsd:element name="State" type="xsd:string"/>
  <xsd:element name="ContactInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="Address" minOccurs="1" maxOccurs="2"/>
        <xsd:element ref="Phone" minOccurs="1" maxOccurs="3"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="Zip" type="zipType"/>
  <xsd:element name="Address">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="Street"/>
        <xsd:element ref="City"/>
        <xsd:element ref="State"/>
        <xsd:element ref="Zip"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="City" type="xsd:string"/>
  <xsd:element name="Street" type="xsd:string"/>
  <xsd:element name="Phone">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="work"/>
        <xsd:element ref="home"/>
        <xsd:element ref="mobile"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="home" type="xsd:string"/>
  <xsd:simpleType name="zipType">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[0-9]{5}(-[0-9]{4})?"></xsd:pattern>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

A.3 Sample XML data

Example A-14 is an XML file used in 5.4.3, “Node-level access control” on page 241.

Example: A-14 employee002.xml

```
<?xml version="1.0"?>
<Employee>
  <Name>John Smith2</Name>
  <EmpNo>002</EmpNo>
  <Title>Engineer</Title>
  <DateOfBirth>2/22/1967</DateOfBirth>
  <SSN>892-76-0002</SSN>
  <Address country="US">
    <Street>2 East Main Street</Street>
    <City>Los Gatos</City>
    <State>CA</State>
    <Zip>95034</Zip>
  </Address>
  <Phone type="work">312-964-0002</Phone>
  <Phone type="home">678-181-0002</Phone>
  <Email>john.smith2@my.com</Email>
  <Salary>20000</Salary>
</Employee>
```

Example A-15 is an XML file used in 5.4.3, “Node-level access control” on page 241.

Example: A-15 employee003.xml

```
<?xml version="1.0"?>
<Employee>
  <Name>John Smith3</Name>
  <EmpNo>003</EmpNo>
  <Title>Architect</Title>
  <DateOfBirth>2/23/1967</DateOfBirth>
  <SSN>892-76-0003</SSN>
  <Address country="US">
    <Street>3 East Main Street</Street>
    <City>Los Gatos</City>
    <State>CA</State>
    <Zip>95034</Zip>
  </Address>
  <Phone type="work">312-964-0003</Phone> <Phone
type="home">678-181-0003</Phone>
```

```
<Email>john.smith3@my.com</Email>
<Salary>30000</Salary>
</Employee>
```

Example A-16 is an XML file used in 5.4.3, “Node-level access control” on page 241

Example: A-16 employee004.xml

```
<?xml version="1.0"?>
<Employee>
  <Name>John Smith4</Name>
  <EmpNo>004</EmpNo>
  <Title>Director</Title>
  <DateOfBirth>2/24/1967</DateOfBirth>
  <SSN>892-76-0004</SSN>
  <Address country="US">
    <Street>4 East Main Street</Street>
    <City>Los Gatos</City>
    <State>CA</State>
    <Zip>95034</Zip>
  </Address>
  <Phone type="work">312-964-0004</Phone>
  <Phone type="home">678-181-0004</Phone>
  <Email>john.smith4@my.com</Email>
  <Salary>40000</Salary>
</Employee>
```

Example A-17 is an XML file used in 5.4.3, “Node-level access control” on page 241.

Example: A-17 employee005.xml

```
<?xml version="1.0"?>
<Employee>
  <Name>John Smith5</Name>
  <EmpNo>005</EmpNo>
  <Title>CEO</Title>
  <DateOfBirth>2/25/1967</DateOfBirth>
  <SSN>892-76-0005</SSN>
  <Address country="US">
    <Street>5 East Main Street</Street>
    <City>Los Gatos</City>
    <State>CA</State>
    <Zip>95034</Zip>
```

```

</Address>
<Phone type="work">312-964-0005</Phone>
<Phone type="home">678-181-0005</Phone>
<Email>john.smith5@my.com</Email>
<Salary>50000</Salary>
</Employee>

```

Example A-18 shows the INSERT statements for the EMP table, which is used in 5.4.3, “Node-level access control” on page 241.

Example: A-18 Insert statement for John Smith2 - John Smith5

```

INSERT INTO EMP VALUES ('002', XMLPARSE( DOCUMENT
'<?xml version="1.0"?>
<Employee>
  <Name>John Smith2</Name>
  <EmpNo>002</EmpNo>
  <Title>Engineer</Title>
  <Phone type="work">312-964-0002</Phone>
  <Email>john.smith2@my.com</Email>
</Employee>'),
XMLPARSE( DOCUMENT
'<?xml version="1.0"?>
<Employee>
  <Name>John Smith2</Name>
  <EmpNo>002</EmpNo>
  <DateOfBirth>2/22/1967</DateOfBirth>
  <SSN>892-76-0002</SSN>
  <Address country="US">
    <Street>2 East Main Street</Street>
    <City>Los Gatos</City>
    <State>CA</State>
    <Zip>95034</Zip>
  </Address>
  <Phone type="home">678-181-0002</Phone>
  <Salary>20000</Salary>
</Employee>'),
SECLABEL_BY_NAME('EMP_POLICY', 'PUBLIC'));

INSERT INTO EMP VALUES ('003', XMLPARSE( DOCUMENT
'<?xml version="1.0"?>
<Employee>
  <Name>John Smith3</Name>
  <EmpNo>003</EmpNo>
  <Title>Architect</Title>

```

```

        <Phone type="work">312-964-0003</Phone>
        <Email>john.smith3@my.com</Email>
    </Employee>'),
    XMLPARSE( DOCUMENT
    '<?xml version="1.0"?>
    <Employee>
        <Name>John Smith3</Name>
        <EmpNo>003</EmpNo>
        <DateOfBirth>2/23/1967</DateOfBirth>
        <SSN>892-76-0003</SSN>
        <Address country="US">
            <Street>3 East Main Street</Street>
            <City>Los Gatos</City>
            <State>CA</State>
            <Zip>95034</Zip>
            <Phone type="home">678-181-0003</Phone>
        </Address>
        <Salary>30000</Salary>
    </Employee>'),
    SECLABEL_BY_NAME('EMP_POLICY', 'PUBLIC'));

INSERT INTO EMP VALUES ('004', XMLPARSE( DOCUMENT
    '<?xml version="1.0"?>
    <Employee>
        <Name>John Smith4</Name>
        <EmpNo>004</EmpNo>
        <Title>Director</Title>
        <Phone type="work">312-964-0004</Phone>
        <Email>john.smith4@my.com</Email>
    </Employee>'),
    XMLPARSE( DOCUMENT
    '<?xml version="1.0"?>
    <Employee>
        <Name>John Smith4</Name>
        <EmpNo>004</EmpNo>
        <DateOfBirth>2/24/1967</DateOfBirth>
        <SSN>892-76-0004</SSN>
        <Address country="US">
            <Street>4 East Main Street</Street>
            <City>Los Gatos</City>
            <State>CA</State>
            <Zip>95034</Zip>
            <Phone type="home">678-181-0004</Phone>
        </Address>
        <Salary>40000</Salary>

```

```

</Employee>'),
SECLABEL_BY_NAME('EMP_POLICY', 'HR_ONLY'));

INSERT INTO EMP VALUES ('005', XMLPARSE( DOCUMENT
'<?xml version="1.0"?>
<Employee>
  <Name>John Smith5</Name>
  <EmpNo>005</EmpNo>
  <Title>Manager</Title>
  <Phone type="work">312-964-0005</Phone>
  <Email>john.smith5@my.com</Email>
</Employee>'),
XMLPARSE( DOCUMENT
'<?xml version="1.0"?>
<Employee>
  <Name>John Smith5</Name>
  <EmpNo>005</EmpNo>
  <DateOfBirth>2/25/1967</DateOfBirth>
  <SSN>892-76-0005</SSN>
  <Address country="US">
    <Street>5 East Main Street</Street>
    <City>Los Gatos</City>
    <State>CA</State>
    <Zip>95034</Zip>
  <Phone type="home">678-181-0005</Phone>
  </Address>
  <Salary>50000</Salary>
</Employee>'),
SECLABEL_BY_NAME('EMP_POLICY', 'HR_ONLY'));

```

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/SG24-7315-01>

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, SG24-7315-01.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

<i>File name</i>	<i>Description</i>
ch5sampledata.zip	Zipped sample data used in chapter 5
setup.zip	Zipped sample application code and data

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space:	3 MB
Operating System:	Windows/Linux
Processor:	486 or higher
Memory:	512 MB

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder.

Abbreviations and acronyms

ACORD	Association for Cooperative Operations Research and Development	SMTP	Simple Mail Transfer Protocol
		SOA	Service Oriented Architecture
		SOAP	Simple Object Access Protocol
ANSI	American National Standards Institute	UDDI	Universal Description, Discovery and Integration
API	Application Program Interface	UDF	User Defined Functions
BLOB	Binary Large Object	UML	Unified Modeling Language
CLI	Command Line Interface	URI	Uniform Resource Identifier
CLOB	Character Large Object	URL	Uniform Resource Locator
CLP	Command Line Processor	UTC	Coordinated Universal Time
DADX	Document Access Definition Extension	WSDL	Web Services Description Language
DBMS	Database Management System	XDA	XML Data Area
DDL	Data Definition Language	XDM	XQuery/XPath Data Model
DMS	Database Management Space	XDS	XML Data Specifier
DTD	Document Type Definition	XML	eXtensible Markup Language
DWB	Developer WorkBench	XRS	XML Schema Repository
FLWOR	ffor, let, where, order by, and return	XSD	XML Schema Definition
HADR	High Availability and Disaster Recovery	XSR	XML Schema Repository
		IWORF	Web services Object Runtime Framework
HTML	Hyper Text Markup Language		
HTTP	Hypertext Transfer Protocol		
IXF	Integrated Exchanged Format		
JDK	Java Development Kit		
LBAC	Label-based Access Control		
LOB	Large Object		
ODBC	Open Database Connectivity		
RSS	Really Simple Syndication		
SMS	System Managed Space		

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information about ordering these publications, see “How to get IBM Redbooks” on page 380. Note that some of the documents referenced here may be available in softcopy only.

- ▶ *DB29: pureXML Overview and Fast Start*, SG24-7298
- ▶ *DB2 Express-C: The Developer Handbook for XML, PHP, C/C++, Java, and .NET*, SG24-7301

Other publications

These publications are also relevant as further information sources:

IBM - DB2 9

- ▶ *What's New*, SC10-4253
- ▶ *Administration Guide: Implementation*, SC10-4221
- ▶ *Administration Guide: Planning*, SC10-4223
- ▶ *Administrative API Reference*, SC10-4231
- ▶ *Administrative SQL Routines and Views*, SC10-4293
- ▶ *Call Level Interface Guide and Reference, Volume 1*, SC10-4224
- ▶ *Call Level Interface Guide and Reference, Volume 2*, SC10-4225
- ▶ *Command Reference*, SC10-4226
- ▶ *Data Movement Utilities Guide and Reference*, SC10-4227
- ▶ *Data Recovery and High Availability Guide and Reference*, SC10-4228
- ▶ *Developing ADO.NET and OLE DB Applications*, SC10-4230
- ▶ *Developing Embedded SQL Applications*, SC10-4232
- ▶ *Developing Java Applications*, SC10-4233

- ▶ *Developing Perl and PHP Applications*, SC10-4234
- ▶ *Getting Started with Database Application Development*, C10-4252
- ▶ *Getting started with DB2 installation and administration on Linux and Windows*, GC10-4247
- ▶ *Message Reference Volume 1*, SC10-4238
- ▶ *Message Reference Volume 2*, SC10-4239
- ▶ *Migration Guide*, GC10-4237
- ▶ *Performance Guide*, SC10-4222
- ▶ *Query Patroller Administration and User's Guide*, GC10-4241
- ▶ *Quick Beginnings for DB2 Clients*, GC10-4242
- ▶ *Quick Beginnings for DB2 Servers*, GC10-4246
- ▶ *Spatial Extender and Geodetic Data Management Feature User's Guide and Reference*, SC18-9749
- ▶ *SQL Guide*, SC10-4248
- ▶ *SQL Reference, Volume 1*, SC10-4249
- ▶ *SQL Reference, Volume 2*, SC10-4250
- ▶ *System Monitor Guide and Reference*, SC10-4251
- ▶ *Troubleshooting Guide*, GC10-4240
- ▶ *Visual Explain Tutorial*, SC10-4319
- ▶ *XML Extender Administration and Programming*, SC18-9750
- ▶ *XML Guide*, SC10-4254
- ▶ *XQuery Reference*, SC18-9796
- ▶ *DB2 Connect User's Guide*, SC10-4229
- ▶ *Quick Beginnings for DB2 Connect Personal Edition*, GC10-4244
- ▶ *Quick Beginnings for DB2 Connect Servers*, GC10-4243

IBM - DB2 8.2

- ▶ *What's New V8*, SC09-4848-01
- ▶ *Administration Guide: Implementation V8*, SC09-4820-01
- ▶ *Administration Guide: Performance V8*, SC09-4821-01
- ▶ *Administration Guide: Planning V8*, SC09-4822-01
- ▶ *Application Development Guide: Building and Running Applications V8*, SC09-4825-01

- ▶ *Application Development Guide: Programming Client Applications V8*, SC09-4826-01
- ▶ *Application Development Guide: Programming Server Applications V8*, SC09-4827-01
- ▶ *Call Level Interface Guide and Reference, Volume 1, V8*, SC09-4849-01
- ▶ *Call Level Interface Guide and Reference, Volume 2, V8*, SC09-4850-01
- ▶ *Command Reference V8*, SC09-4828-01
- ▶ *Data Movement Utilities Guide and Reference V8*, SC09-4830-01
- ▶ *Data Recovery and High Availability Guide and Reference V8*, SC09-4831-01
- ▶ *Guide to GUI Tools for Administration and Development*, SC09-4851-01
- ▶ *Installation and Configuration Supplement V8*, GC09-4837-01
- ▶ *Quick Beginnings for DB2 Clients V8*, GC09-4832-01
- ▶ *Quick Beginnings for DB2 Servers V8*, GC09-4836-01
- ▶ *Replication and Event Publishing Guide and Reference*, SC18-7568
- ▶ *SQL Reference, Volume 1, V8*, SC09-4844-01
- ▶ *SQL Reference, Volume 2, V8*, SC09-4845-01
- ▶ *System Monitor Guide and Reference V8*, SC09-4847-01
- ▶ *Data Warehouse Center Application Integration Guide Version 8 Release 1*, SC27-1124-01
- ▶ *DB2 XML Extender Administration and Programming Guide Version 8 Release 1*, SC27-1234-01
- ▶ *Federated Systems PIC Guide Version 8 Release 1*, GC27-1224-01

Online resources

These Web sites are also relevant as further information sources:

- ▶ DB2 XML wiki
<http://www.ibm.com/developerworks/wikis/display/db2xml/Home>
- ▶ DB2 Information Center
<http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp>
- ▶ DB2 Express-C
<http://www.ibm.com/software/data/db2/udb/db2express/>

- ▶ C. M. Saracco. *Managing XML for Maximum Return*, IBM Whitepaper, October 2005.
<ftp://ftp.software.ibm.com/software/data/pubs/papers/managingxml.pdf>
- ▶ Matthias Nicola and Bert Van der Linden. *Native XML Support in DB2 Universal Database*, Proceedings of the 31st Annual VLDB, 2005.
<http://www.vldb2005.org/program/paper/thu/p1164-nicola.pdf>
- ▶ Matthias Nicola. *15 best practices for pureXML performance in DB2 9*, IBM developerWorks, October 2006.
<http://www.ibm.com/developerworks/db2/library/techarticle/dm-0610nicola/>
- ▶ C. M. Saracco. *What's New in DB2 Viper: XML to the Core*, IBM developerWorks, February 2006.
<http://www.ibm.com/developerworks/db2/library/techarticle/dm-0602saracco/>
- ▶ Holger Seubert and Sabine Perathoner-Tschaffler. XML full-text search in DB2, IBM developerWorks article, June 2006.
<http://www.ibm.com/developerworks/db2/library/techarticle/dm-0606seubert/index.html>
- ▶ Hardeep Singh. *XML application migration from DB2 8.x to DB2 Viper, Part 1: Partial updates to XML documents in DB2 Viper*, IBM developerWorks, May 2006.
<http://www.ibm.com/developerworks/db2/library/techarticle/dm-0605singh/>

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

Archived

Index

Symbols

!= 91

>= 91

A

access control 248

access plan 10, 186, 189–190, 192, 194–195

Add-In 250, 253, 321–322

ADO.NET 249, 321–322

API 250, 290, 318–319, 322

APIs 6, 317

application 6, 11, 15, 21–29, 31–38, 40, 47–48, 52–53, 55, 57, 61, 63, 66, 77, 145, 156, 160, 162, 164, 167, 171, 176, 207, 217, 223, 249–250, 268–269, 271, 273, 275–280, 286–291, 294–296, 299, 301–302, 304, 306, 309–310, 314–319, 321, 342, 344, 349, 354–356, 361–362

application programming interfaces 268

argument 88–89, 97, 99–101, 126, 144, 150–151, 159, 161, 271, 276, 282, 315, 319, 354, 357

array 237, 311–312, 315, 319

attribute 45, 50, 64, 67, 74, 76–80, 86, 93–94, 110, 122, 137–138, 140, 150, 158, 161, 167–168, 170–171, 201, 206–208, 210, 213, 217, 219–220, 223, 225, 243, 266, 274, 287

axis 80

B

bind-in 43

bind-out 43

BLOB 5, 149–150, 204, 271–272, 294–297, 299–300, 304–305

Boolean operators 162

boundary whitespace 272

B-Tree index 176

buffer 69, 282, 284, 287

business logic 349

C

Call Level Interface 249, 273, 278–279

carriage returns 272

character string 180, 297

character type 282, 287, 306, 311–312, 314

class 4, 51, 53, 68, 222, 303–304, 319, 344

CLI 11, 268, 271, 273, 278–280, 282, 286–287, 290–291, 341–343

CLI driver 278

CLOB 5, 43, 45, 149, 174, 271, 294–299, 304–306, 343–345, 354

COBOL 268, 289, 338, 343

code page 67, 70, 145, 217, 223, 275, 282, 287, 306, 362

colon 80

column path Index 182

column path index 183

Command Line Processor 253

comment() 80

communication 351

complex data type 51, 53, 55

complex type element 50

components 71, 145, 233, 349–350

constraint 180

constructor 89–90, 93, 142, 150

CONTAIN() 159

D

data access 10, 243, 248, 322

data model 13–14, 19, 23, 76, 149, 222, 236, 362

data security 234

data source 342

data type 6–7, 12, 21, 40, 43–44, 50, 53, 55, 66, 68–69, 88–89, 95, 97, 100, 127, 132, 142, 144–145, 149–150, 178–181, 196, 198–199, 208, 212, 216, 222, 226, 229, 233, 251–255, 269–270, 273, 278, 282, 284, 288–289, 294, 297–299, 302, 307–308, 313–314, 322, 324, 338–339, 343–346

database objects 42, 70, 198, 253–254

DB2 Developer Workbench 12, 254

DB2 Express-C 37

db2cli.ini 273, 279

db2cli.lst 279

DB2Connection 349

db2-fn

sqlcolumn 103

sqlquery 103

decimal 179
decomposition 11, 146, 148
default namespace 98, 137, 139, 141, 151, 166
delimiters 297
descendant 79
distribution 229, 232
document model 167, 169, 171
document-node() 80
dynamic 17, 77, 96, 270, 276, 278, 299–300, 315, 340

E

e-business 21
element 10–11, 47, 49, 51, 53–54, 61, 64, 67, 74, 76–78, 80, 83–85, 87, 93–94, 101–102, 110, 113–114, 116–117, 120, 122, 124, 136–138, 140, 142, 147–148, 150–151, 154, 159–162, 164–167, 169–170, 186–188, 190, 192, 196, 199–200, 206–207, 215, 229, 237–238, 243, 274, 277, 288, 300–301, 328, 340, 342, 367
embedded SQL 73, 126–127, 133–134, 136, 271–272, 278, 280, 289–291, 299
environment 5, 14, 21, 36–37, 87, 155, 158, 189, 250–251, 278–279, 281, 292, 302, 316, 321, 342, 355–356
eq 91
explicit validation 65
export 222–224, 226–227, 252–253, 310, 322, 325
expression 10, 47, 74, 78, 80–81, 84, 86–87, 90–92, 94–96, 103, 105–108, 112, 115, 118–119, 126–127, 129–130, 161, 168–169, 178, 197, 229, 266, 269, 276, 297, 299, 302, 310–315, 319, 340
extensions 6, 278, 315, 318

F

fetch 276, 289, 300–301, 319
FLWOR 90, 103, 106–108, 112, 119, 126, 266
fn
 abs 99
 ceiling 100
 codepoints-to-string 98
 compare 98
 concat 98
 contains 98
 empty 101
 ends-with 98
 exist 101
 floor 100

implicit-timezone 100
last 101
local-name 102
lower-case 98
matches 98
max 100
min 100
node-name 102
normalize-space 98
number 100
remove 101
round 100
starts-with 98
string-join 98
string-length 99
substring-before 99
tokenize 99
translate 99
upper-case 99

forward-only 319
function 43, 48, 61, 68, 78, 81, 86, 88–90, 96–98, 115, 122, 126, 128–130, 133, 136, 138, 140–141, 143–144, 149–153, 155, 159–160, 170, 197, 233, 243, 245, 248, 269–271, 274, 277–278, 282, 284, 286–287, 294, 297, 299–300, 302, 314, 319–320, 339

G

ge 91
global catalog path table 176
GRANT 237–239, 246

H

handle 2, 17, 40, 180, 288, 341–342, 356
host language 289
host variable 269, 271, 274, 276, 283, 294–295, 298

I

IBM.Data.DB2 322, 343
ibm_db2 316, 318–319
implicit parsing 269, 285
implicit validation 61, 63, 65
import 42–43, 51, 60–61, 164, 200–202, 208–213, 216–217, 221, 322, 325, 344
index access 176
installation 37, 250, 302–303, 316–317, 321

J

Java 11, 38, 122, 249, 254, 268, 274, 277,
302–306, 310, 338, 343–347, 349
JDBC 11–12, 268, 302–304, 307, 310, 345–346,
349, 358

K

keyword 94, 105, 252, 288, 299

L

Label-based access control 234
LBAC 234
le 91
let 103
LOB 42–44, 149, 222–224, 294, 297, 299, 301
local complex type 50
local name 75, 79, 102, 137, 140–141
location 84
logical index 181
lt 91

M

metadata 302, 318
method 12, 15, 18, 155, 159, 282, 305–307, 315,
346, 348–349, 352, 354–355, 357, 359
modular 315, 349
monitor 68–69

N

namespace 50, 61–62, 64, 67, 73, 75, 79, 87–88,
97–98, 102, 123, 137–141, 150–151, 166, 169, 200,
203, 255, 267, 277, 288, 300–301, 322, 340, 342,
344
namespace prefix 102, 137, 139
native data type 40
NCName 79
ne 91
Net Search Extender 11, 73, 154–155, 157, 159,
162, 167
node 7, 67, 74–80, 83–84, 86, 90, 92–93, 96, 102,
113, 122–123, 140–141, 144, 150, 152, 154, 168,
174, 178, 181, 185, 196, 229, 244, 247–248, 251,
255, 264–265, 279
node value 180
NSE 154–155, 157, 159–160, 164, 167, 169
numeric attribute 159

O

ODBC 11, 278–279, 318–319
OLE DB 249, 321
optimizer 40, 155, 186, 229
options 10, 39, 70, 123, 133–134, 143, 185, 208,
216, 221, 223–224, 226, 231, 268–269, 297, 319,
333
order by 103
overhead 5, 7, 45, 47–48, 60, 199, 282, 318, 351

P

package 316–317
parameter 123, 135–136, 170, 209, 212–213,
219–220, 228, 268–270, 276, 280, 282, 285, 291,
306, 308, 314, 319, 322, 324, 338–340, 343–346,
348, 357
parameter marker 269–270, 277
parser 218
path ID 176
PDO_ODBC 318–319
performance 5, 7, 10, 17, 19, 40, 44, 47–49, 63, 65,
69, 173, 176, 197, 199, 230, 319
Perl 249, 316
PHP 11, 249, 268, 315, 317–321
physical index 181
precompile 280, 291, 297
precompiler 278, 297
predicate 86, 186
predicate expression 86
prefix 75, 79, 88, 97, 102, 123, 137–138, 140–141,
166
PREPARE 299–300, 340
primary schema 62
privilege 71, 239, 246
procedures 251–252, 254–255, 268, 303,
338–340, 343–344, 346, 348–349, 356
processing-instruction() 80
programming interface 250, 278, 318
programming language 250, 268, 276, 315,
349–350
protocol 43, 57, 349, 351–352
pureXML 1, 4, 6, 11–13, 17–19, 21, 40, 43, 66, 68,
154, 173–174, 253, 268, 353
pureXML storage 67

Q

QName 75, 77, 79, 102, 137, 139, 141
qualified name 75, 78, 102, 137, 166, 169

query 176, 178–179, 183, 185–187, 189–190, 192,
194–196, 203–204, 240, 248

R

Redbooks Web site 380
 Contact us xv
registry 351
repository 10, 18, 57, 71, 198–199, 202, 251, 255,
348
result set 49, 97, 192, 240, 247–248, 286, 288,
319–320, 341, 357, 359
ResultSet 277, 307–308, 313, 345, 347
return 103
runtime 10, 229, 253, 278, 301, 310, 312–313, 355

S

schema 10–11, 13, 15–16, 19, 42, 44–45, 47,
49–54, 56, 59–64, 66–67, 70, 75, 138, 144–145,
147–148, 173, 180, 185, 198–208, 210, 212–213,
215, 219–220, 222, 225, 251, 253, 255, 262, 319,
333, 348–349, 366–367
schema document 51, 53, 55, 61, 63, 144, 146,
199, 201–202, 204, 348
schema validation 208, 212–213, 219–220
scripts 253–254, 279–280, 290, 292, 319
SECADM authority 233
security label 237–238
security label component 237
security policy 237
security structure 237
serialized string 149, 151, 269, 275, 282, 286–287,
304, 307, 310, 313
setup 21, 36–37, 122, 233, 237, 243–244,
247–248, 278–279, 289, 321
shredding 5, 7, 15, 146–147, 149
simple type element 50
source file 210, 280, 291, 304, 309, 322, 344
SQL 1, 6, 8, 10–11, 38, 40, 43, 45, 47, 49, 61–63,
71, 73, 97, 103, 110, 115, 117, 120–121, 123–124,
126, 129–130, 132–135, 142–143, 155, 159, 173,
176, 178–181, 186–190, 192, 194, 196, 201–203,
205, 213, 226, 234, 249, 251, 253–254, 269–271,
273, 276–278, 280, 289–291, 294–302, 319–320,
334–335, 338–341, 343–345, 353–357
SQL statement 127, 300, 320
SQL_C_BINARY 282–283, 285, 287, 289
SQL_HANDLE_DBC 341–342
SQL_HANDLE_ENV 341

SQL_HANDLE_STMT 288, 342
SQLCA 289, 300
SQLCODE 215
SQLJ 11, 38, 122, 254, 268, 273, 302–304,
309–310, 313, 346, 348–349
sqlj 309–310, 312–313
sqlname.data 301
SQLSTATE 202, 215, 241
SQLVAR 301
square bracket 86
statement 8–10, 43, 45, 61, 63, 65, 68, 124,
126–127, 129, 142–143, 157, 177, 179, 181, 183,
197, 234, 238–240, 246–247, 269, 272, 276–277,
283–284, 288–289, 299–301, 315, 319–320, 335,
339–340, 342–347, 354, 357, 359, 370
static 271, 276–277, 298–300, 347, 362
static embedded SQL 272
statistics 12, 34, 59, 185–186, 189, 229–231, 252
stemming search 159, 163
storage model 66
stored procedure 37–38, 122, 124–125, 270, 281,
293, 324, 341, 345–348
string value 77, 99, 143
SYSCAT.INDEXES. Even though the XML column
path Index and the XML regions Index 182

T

tables 251–254, 268, 299, 323
text node 80, 124, 153
text() 80
thesaurus search 159
toolbar 331
tree structure 40, 43, 67

U

Unicode 7, 43, 98
Uniform Resource Identifier 137
URI 75, 79, 102, 137–138, 140–141
UTF-8 217, 223

V

VARCHAR 5, 38, 42–45, 122, 130, 159, 179–181,
183, 187–188, 190, 192, 196, 203–204, 208, 246,
286, 294, 339–340, 354
variable 37, 43, 78, 90, 95, 103–108, 152, 158, 269,
271, 274, 276, 282–283, 285, 287–288, 293–295,
298, 302, 306, 338–341

varying-length string 180
views 43, 48–49, 70–71, 199, 243–244, 247, 253, 323

W

well-formed XML 10, 43, 66, 143, 208, 269, 282
whitespace 48, 62, 65, 98, 143, 157, 218, 272, 274, 294, 306, 311–312, 340
wildcard search 164

X

XDA 174
XML 1–2, 4–8, 10–13, 15–21, 24, 28, 31, 38, 40–51, 53, 55–60, 62–64, 66–67, 69–70, 73–75, 77–78, 80–81, 83–85, 88, 90, 93, 97, 103, 109–110, 112, 114–118, 120–124, 126, 128–130, 132–136, 138–139, 141–145, 147–149, 151–152, 154–155, 158–161, 164–167, 169, 173–179, 181, 184–187, 189–190, 195–199, 201–206, 208–210, 212–213, 215–225, 227, 229–231, 233, 235–236, 239, 242, 244, 247–252, 254–255, 257, 260–264, 266, 268–271, 273–275, 277–279, 282–285, 287–290, 294–296, 299–308, 310, 312–315, 322–323, 325–329, 331, 333, 337–341, 343–349, 351–353, 356–358, 361–364, 366, 368–369
XML column path index 176
XML Data Area 174
XML data descriptor 174
XML index 174, 176, 178–179, 181, 183, 187–188, 192, 195–197
XML schema 10, 13, 42, 44–45, 47, 49–51, 56, 60, 62–63, 66–67, 71, 75, 138, 144, 146, 148, 180, 198–199, 201–204, 206, 208, 210, 212–213, 221–222, 225, 251, 255, 348–349
XML schema repository 10, 198–199, 251, 255, 348
XML value index 195
XMLATTRIBUTES 150
XMLCONCAT 152
XMLELEMENT 150
XMLEXISTS 127
XMLFOREST 151
xmlns 137, 139, 147–148, 152, 155–156, 165–166
XMLPARSE 43, 81, 109, 111, 121, 139, 143–144, 218, 220–221, 239, 269–273, 282, 294, 297, 299, 306, 311–312, 339–340, 370–372
xmlpattern 10, 174, 177, 179, 181, 183, 197
XMLQUERY, 127

XMLSERIALIZE 43, 149, 297, 314
XMLTABLE 127, 151
XMLUPDATE 122
XPath 10, 74, 78, 86–87, 123, 134, 149, 160, 167–169, 176, 178, 190, 229, 353
XQUERY 81, 84–87, 93, 104–108, 112–117, 119–121, 126–127, 129, 140–141, 144, 160, 165, 178, 188, 192, 196, 252, 276, 288, 299–300, 302, 307, 320, 340, 342, 344
XSCAN 186
XSR 60, 70, 148, 199, 202–203, 205, 251, 333, 348–349



DB2 9 pureXML Guide

(0.5" spine)
0.475" <-> 0.875"
250 <-> 459 pages



DB2 9 pureXML Guide



Redbooks

**Learning SQL/XML,
XQuery, XPath with
working examples**

**Developing XML
applications with
DB2 pureXML**

**Managing XML for
maximum return**

IBM DB2 9 for Linux, UNIX, and Windows marks a new stage in the evolution of data servers. IBM has continually led the data management industry with the release of innovative technology. DB2 9 is a new generation data server with revolutionary pureXML technology. This technology in DB2 9 fundamentally transforms the way XML information is managed for maximum return while seamlessly integrating XML with relational data.

In this IBM Redbook we discuss the pureXML data store, hybrid database design, and administration. We describe XML schemas, industry standards, and how to manage schemas. We also cover SQL/XML, XQuery, and XPath using easy-to-understand examples. Lastly, we show how to use XML technology efficiently in business applications.

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