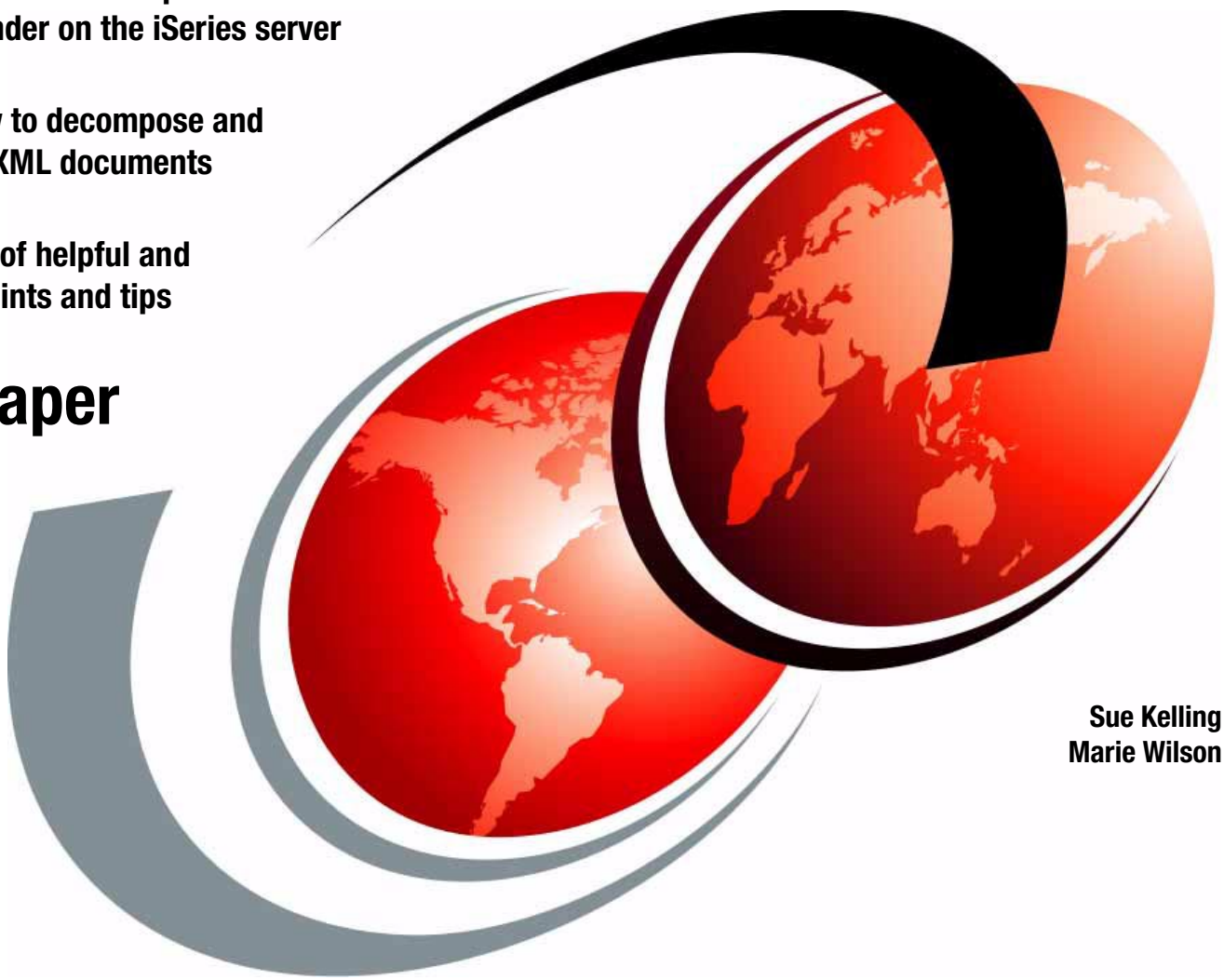
IBM

DB2 XML Extender

Hints and Tips for the IBM @server iSeries Server

- Understand how to implement DB2 XML Extender on the iSeries server
- Learn how to decompose and compose XML documents
- Make use of helpful and valuable hints and tips

Redpaper



Sue Kelling
Marie Wilson

ibm.com/redbooks

Redbooks



International Technical Support Organization

DB2 XML Extender
Hints and Tips for the IBM @server iSeries Server

August 2001

Take Note! Before using this information and the product it supports, be sure to read the general information in “Special notices” on page 83.

First Edition (August 2001)

This edition applies to DB2 UDB XML Extenders for AS/400 for use with the iSeries server OS/400 V5R1.

This document created or updated on August 24, 2001.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. JLU Building 107-2
3605 Highway 52N
Rochester, Minnesota 55901-7829

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 2001. All rights reserved.
Note to U.S Government Users - Documentation related to restricted rights - Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Preface v
The team that wrote this Redpaper v
Special notice v
IBM trademarks vi
Comments welcome vi

Chapter 1. Introduction 1
1.1 Introduction 2
1.2 Scenario overview 2
1.3 XML documents 3

Chapter 2. Getting started 5
2.1 DB2 XML Extender overview 6
2.1.1 XML column 6
2.1.2 XML collection 7
2.1.3 Choosing an access and storage method 8
2.2 Installation 8
2.3 Enabling DB 8

Chapter 3. Itinerary 11
3.1 Database details 12
3.2 Starting XML document 14
3.3 Discoveries 15
3.4 Final DTD 19
3.5 Final XML document 20
3.6 DTD mapping scheme 21
3.7 Final DAD 23
3.8 XML enablement commands 26
3.9 Trigger source 26
3.10 Java source 28

Chapter 4. Passengers 29
4.1 Database details 30
4.2 Starting XML document 32
4.3 Discoveries 33
4.4 Final DTD 34
4.5 Final XML document 35
4.6 DTD mapping scheme 35
4.7 Final DAD 36
4.8 XML enablement commands 38
4.9 Java source 38

Chapter 5. Invoice 41
5.1 Database details 42
5.2 Starting the XML document 43
5.3 Discoveries 44
5.4 Final XML document 48
5.5 DTD mapping scheme 49
5.6 Final DADs 51

© Copyright IBM Corp. 2001

5.7 XML enablement commands 53
5.8 Java source 53

Chapter 6. Payment 55
6.1 Composing paymentRequest 56
6.1.1 Database details 56
6.1.2 Starting XML document 57
6.1.3 Discoveries 58
6.1.4 Final DTD 59
6.1.5 Final XML document 60
6.1.6 DTD mapping scheme 61
6.1.7 Final DAD 63
6.1.8 XML enablement commands 66
6.1.9 Java source 66
6.2 Decomposing paymentRequest 67
6.2.1 Database details 67
6.2.2 XML document 68
6.2.3 Discoveries 68
6.2.4 Final DTD 70
6.2.5 Final XML document 70
6.2.6 DTD mapping scheme 70
6.2.7 Final DAD 72
6.2.8 XML enablement commands 73
6.2.9 Trigger source 73
6.2.10 Java source 74
6.3 Saving PaymentResponse 74
6.3.1 Database details 75
6.3.2 Discoveries 75
6.3.3 Final DTD 75
6.3.4 Final XML document 75
6.3.5 XML enablement commands 75
6.3.6 Java source 76

Chapter 7. Miscellaneous discoveries 77

Chapter 8. Final thoughts 81

Special notices 83

Preface

This Redpaper provides readers with a broad understanding of the new architecture made available by the DB2 XML Extender product. This content is intended for the following people:

- DB2 application programmers who are familiar with SQL and Java
- DB2 database administrators who are familiar with DB2 administration concepts, tools, and techniques
- Developers and programmers who work with XML data in DB2 applications and who are familiar with XML concepts

The team that wrote this Redpaper

This Redpaper is brought to you by a group of iSeries developers working in the IBM Rochester laboratory (called the Platform Evaluation Test team). Platform Evaluation Test provides a pre-GA readiness assessment of the iSeries server's ability to be installed and operated in customer-like environments. These test environments focus on the total solution, including current release of the operating system, new and existing hardware, and running customer-like scenarios. The scenarios, designed and constructed by the team, demonstrate how iSeries technology can be used to build e-business solutions.

Sue Kelling is a Staff Software Engineer at the IBM Rochester Laboratory. She has five years of experience in the software testing field.

Marie Wilson is a Staff Software Engineer at the IBM Rochester Laboratory. She has 15 years of experience in the software testing field.

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

Emily Chun
Jim Voge
Susan Malaika
Dung Nguyen
Angel Reyda
Laura McCrea
Mark Megarian
Kathy Johnson
Cindy Murch
Marilyn Dukart
Tom Eggebraaten
Dan Boyum
International Technical Support Organization, Rochester Center

Special notice

This publication is intended to help those interested in DB2 UDB XML Extenders for AS/400. The information in this publication is not intended as the specification of any programming interfaces that are provided by iSeries server. See the PUBLICATIONS section of the IBM Programming Announcement for iSeries server for more information about what publications are considered to be product documentation.

© Copyright IBM Corp. 2001

IBM trademarks

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

e (logo)® 
IBM ®
AS/400
AT
CT
DB2
DB2 Universal Database
IBM
OS/400
SP
WebSphere
Wizard
XT

Redbooks
Redbooks Logo 
400
Domino
Tivoli
Manage. Anything. Anywhere.
TME
NetView
Cross-Site
Tivoli Ready
Tivoli Certified
Planet Tivoli

Comments welcome

Your comments are important to us!

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

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

  `ibm.com/redbooks`

- Send your comments in an Internet note to:

  `redbook@us.ibm.com`

- Mail your comments to the address on page ii.

1

Introduction

This chapter contains:

- An introduction to the Redpaper
- A definition of the scenario
- An overview of XML documents used to perform business-to-business interactions.

© Copyright IBM Corp. 2001

1.1 Introduction

This Redpaper is an extension of the Redpaper *Business-to-Business Integration Guide*, REDP0139.

The document focuses on the hints, tips, and techniques that the Platform Evaluation Test (PET) team learned while integrating DB2 XML Extender into a scenario. For additional information on DB2 XML Extenders, see the *DB2 XML Extender: Overview* on the Web at:

http://www-4.ibm.com/software/data/db2/extenders/xmlext/

1.2 Scenario overview

The PET team designs, implements, evaluates, and deploys customer-like scenarios in a fashion similar to that used by information technology architects throughout the industry. Although the team is restricted to a test laboratory environment, great effort is made to reflect reality. While we know we cannot possibly cover every possible situation, we try to focus on those scenarios that our customers are most likely to use. With that in mind, we created the travel, flights, and cruise (TFC) scenario to test the user-to-online-buying and the business-to-business (B2B) business patterns.

TFC is composed of several fictitious companies, including a cruise company, an outsourced Web site, a travel agency, a flights company, and a bank. Each of these companies evolved slowly and separately and then linked together in several planned stages. During stage one, which was done prior to the OS/400 V5R1 release testing, the cruise company and the outsourced Web site were created. This stage tested the user-to-online-buying business pattern and focused on the integration of WebSphere Commerce Suite with a Domino back-end application. For more information, see the Redpaper *Integrating WebSphere Commerce Suite with Domino Back-End Application: iSeries 400 Edition,* REDP0141.

During stage two, which was run on OS/400 V5R1, the bank was added and an existing legacy application was separated into a travel agency and a flights company. The travel agency integrated with the cruise company to test the B2B business pattern. The flights company continued to run the set of legacy applications. The current plan is to update the flights company later this year to set up additional B2B relationships.

This section includes an overview of the TFC companies and the services they provide.

The cruise company:

- Maintains cruise and product information
- Handles order processing and billing statements
- Generates itineraries
- Maintains customer and passenger information
- Uses an outsourced Web site to handle the store front

The outsourced Web site:

- Hosts the cruise company's Web site
- Has separate interfaces for:
  - Regular customers
  - Travel agencies
- Provides payment processing for regular customers

The travel agency:

- Handles customer information, cruise searches, and cruise bookings
- Places orders through agents
- Provides itinerary information sent from the cruise company
- Handles billing statements sent from the cruise company
- Provides passenger listings to the cruise company
- Performs bank payment processing

The bank handles payment processing between the cruise company and travel agency.

The flights company:

- Runs the legacy flight test cases
- Handles airline and airport information
- Handles flight information
- Books flights
- Eventually integrates with the travel agency to allow flight booking

1.3 XML documents

This Redpaper describes how the travel agency uses XML to send information between the cruise company, the travel agency, and the bank. In all, five key XML documents were created and used:

- **Itinerary**:

  After the travel agency books cruises from the cruise company, they must obtain the itinerary information from the cruise company. To do this, the travel agency sends a request to the cruise company. The cruise company then sends the itinerary information to the travel agency in the form of an XML document. The travel agency takes the information from the XML document and decomposes it into database tables on their system.

- **Passenger**:

  When the travel agency sells the cruises that it bought from the cruise company, they must send the passenger information back to the cruise company. The travel agency composes an XML document based on the information in their database tables and sends it to the cruise company.

- **Invoice**:

  On a regularly scheduled interval, the cruise company bills the travel agency for the cruises that they booked. The cruise company creates an invoice and sends it to the travel agency as an XML document. The travel agency decomposes the XML document and stores it in their database tables.

- **Payment Request**:

  When the travel agency receives an invoice from the cruise company, they make a payment to the bank. The travel agency composes an XML document to send the payment and payment information to the bank.

  When the bank receives the information from the travel agency, they process the payment request and decompose the XML document into their database tables. The bank then sends a payment response XML document to the travel agency to confirm the payment.

- **Payment Response**:

  When the travel agency receives the XML response document from the bank, they store it in a database table. This allows the travel agency to verify what has been paid to the cruise company.

2

Getting started

This chapter contains:

- An overview of the DB2 XML Extender product
- Installation instructions for the DB2 XML Extender product
- Enabling instructions for the DB for DB2 XML Extender

© Copyright IBM Corp. 2001

2.1 DB2 XML Extender overview

The DB2 XML Extender product provides the ability to generate XML documents from existing data, insert rows into tables from XML documents, and store and access XML documents. It provides new data types, functions, and stored procedures to manage your XML data in DB2 databases. This product is available on OS/400 V5R1 of the iSeries server (5722DE1).

XML Extender also provides features to help manage XML data with DB2. These features include administration tools, storage and access methods, a data type definition (DTD) repository, and a mapping file called the Document Access Definition (DAD):

- **Administration tools**:
  - Enable database and table columns for XML
  - Map XML data to DB2 relational structures
- **Storage and access methods**:
  - XML Column
  - XML Collection
- **Store DTDs**:
  - When a database is enabled for XML, a DTD repository table dtd_ref is created in library DB2XML.
  - The DTDs are used for validating the structure of XML documents.
- **DAD file**:
  - Specifies how structured XML documents are to be processed by the XML Extender
  - Maps the XML document structure to a DB2 table
  - Used for storing XML documents in a column or when composing or decomposing XML data
  - Specifies whether you are storing documents using the XML column method, or defining an XML collection for composition or decomposition

For more detailed information, see *DB2 Universal Database for iSeries XML Extender Administration and Programming*, available on the Web at:

http://publib.boulder.ibm.com/pubs/html/as400/v5r1/ic2924/books/c2711720.pdf

2.1.1 XML column

An XML column is used when storing and accessing whole XML documents in the database. Using the XML column method, you can store the document XML file types, index the column in a side table, and then query or search the XML document. This storage method is useful for archiving applications in which documents are not frequently updated.

Before working with the XML extender to store your documents, you must understand the structure of the XML document so you can determine how to search the document. When planning how to search the document, you must determine:

- **The XML Extender data type to use**: These provide you with a way to store the XML document. The data types are:
  - XMLVarChar: for small documents stored in DB2
  - XMLCLOB: for large documents stored in DB2
  - XMLFILE: for documents stored outside DB2

- **The XML elements and attributes that you frequently search**: These provide the content that can be stored in side tables and indexed to improve performance.

To enable a column for XML extender and store an XML document in the column, you must perform the following tasks:

1. Insert the DTD for the XML document into the DTD reference table.
2. Prepare the DAD file for the XML column, which is used to store the XML document into DB2. It also maps the XML element and attribute contents to DB2 side tables used for indexing.
3. Add a column to the proper DB2 table with an XML user-defined type.
4. Enable the column for XML using the enable_column command or the dxxEnableColumn() stored procedure.
5. Index the side tables for a structural search.
6. Store the document using a user-defined function.

2.1.2 XML collection

An XML collection is used when mapping XML documents structures to DB2 tables. Using an XML collection allows you to compose XML documents from existing databases or decompose XML documents storing the untagged data in DB2 tables. This storage method is good for data interchange when the contents of XML documents are frequently updated.

When using XML collections, there are different considerations for composing documents from DB2 data or decomposing XML document into DB2 data. Perform the following steps to work with XML collections:

1. Determine the structure of the XML document and how it corresponds to the structure of your database data.
2. Create a DTD to describe the structure of the XML document.
3. Create a mapping scheme to show how the structure of the XML document relates to the DB2 tables that you use to compose or decompose.
4. Select a mapping scheme that defines how XML data is represented in a relational database. The mapping scheme is specified in the <Xcollection> element in the DAD file. The XML Extender provides two types of mapping schemes:
   - SQL mapping
   - Relational Database (RDB_node) mapping
5. Create the DAD file. The DAD file defines the relationship between the relational data and the XML document structure.
6. Determine if you want to validate your data against a DTD. If you choose to validate the data, store the DTD in the DTD repository.
7. Enable the collection for XML using the enable_collection command or dxxEnableCollection() stored procedure. You only need to enable the collection if you are using the dxxRetrieveXML() or dxxInsertXML() stored procedures.
8. Compose or decompose the XML document using the appropriate stored procedure.

2.1.3 Choosing an access and storage method

XML Extender provides XML column and XML collection as the two access and storage methods for XML documents. To decide which method best matches your application needs for accessing and manipulating XML data, consider the items listed in Table 2-1.

Table 2-1 Choosing an access and storage method

XML column	XML collection
Stores and retrieves entire XML documents	Decomposes or composes XML documents
Stores XML document for archival and auditing purposes	Stores untagged data in new or existing tables
The XML document is read but not updated	Stores pure data to be used by other pieces of the application
Stores XML documents external to DB2 and use DB2 for search and management	Data in the XML document is stored with collections of data that map well to relational tables
Range-based search on values of XML elements or attributes	Creates different views of your relational data
Document has elements with large text blocks that you want to search while keeping the document intact	Stores the data of the entire XML document but only wants to retrieve a subset of the data
Not frequently updated	Frequently updated

2.2 Installation

To use the XML Extender support on the iSeries server (V5R1), you must first install the DB2 Extender Licensed Program Product (5722DE1 - base and all of the options - and it's related PTFs). It installs the following items:

- DB2 UDB Extenders
- DB2 UDB Text Extender
- DB2 UDB XML Extender
- Text Search Engine

You also must install the International Components for Unicode Product (5722SS1 option 39). Also, ensure that user profiles that use the DB2 XML Extender product are set to the appropriate CCSID (65535 is not supported). Make sure that the user profiles' CCSID matches the CCSID of the XML documents.

2.3 Enabling DB

To store XML information in the database, you must enable it for the XML Extender. When you enable a database for XML, the XML Extender:

- Creates user-defined types (UDTs), user-defined functions (UDFs), and stored procedures
- Creates and populates control tables with the necessary metadata that the XML Extender requires
- Creates the DB2XML schema and assigns the necessary privileges

To enable the database on the iSeries server, issue the following command:

```
CALL PGM(QDBXM/QZXMADM) PARM(enable_db db_name)
```

On the iSeries server, the *db_name* should match the name of the Relational Database in the system's Relational Database Directory (WRKRDBDIRE) that has *LOCAL for the Remote Location.

3

Itinerary

This chapter describes the XML Extender pieces of the itinerary application that allows the travel agency to receive the itinerary XML document from the cruise company. The itinerary XML document contains all of the itinerary information for the cruises that were ordered by the travel agency based on an order number. The itinerary XML document is decomposed by the travel agency into several DB2 database tables.

The travel agency uses an XML collection for storage of the XML data and a custom designed DTD for the itinerary XML document since there were no suitable industry standard DTDs available.

© Copyright IBM Corp. 2001

3.1 Database details

The XML document is decomposed into five DB2 tables. The DB2 tables are:

- cruise_info
- rooms
- port_of_calls_temp
- activities_temp
- day_temp

The cruise_info table contains cruise information that is received from the cruise company. The layout of the cuise_info table is listed in Table 3-1.

Table 3-1 Cruise_info

Key	Field name	Alias name	Data type	Length	VarLen Alloc	Digits, DecPos	Description
P	BOOKING#	BOOKING_NBR	VARCHAR	20	10		Booking number
	CRUISECO	CRUISE_CO	VARCHAR	40	20		Cruise company name
	ORDER#	ORDER_NBR	VARCHAR	20	10		Order number
	PRODID	PRODUCT_ID	VARCHAR	20	10		Product id
	CRUISEID	CRUISE_ID	VARCHAR	20	10		Cruise id
	CRUISEDESC	CRUISE_DESC	VARCHAR	30	10		Cruise description (7 Day...)
	ROOMDESC	ROOM_DESC	VARCHAR	30	10		Room description (std, ocean, ...)
	COSTSING	COST_SINGLE	PACKED			7,2	Cost to the agency for single occupancy
	COSTDOUB	COST_DOUBLE	PACKED			7,2	Double price
	COSTADD	COST_ADDITIONAL	PACKED			7,2	Cost for each additional person over 2
	CUSTCOST	CUST_COST	PACKED			2,2	Cost to the customer (commission added) (percentage)
	DUR	DURATION	INT				Duration of cruise
	DEPTDATE	DEPARTURE_DATE	DATE				Departure date of cruise

The rooms table contains the room numbers that were bought from the cruise company. It is used to track the rooms that are sold to the travel agency customers. The layout of the rooms table is listed in Table 3-2.

Table 3-2 Rooms

Key	Field name	Alias name	Data type	Length	VarLen Alloc	Allow null	Description
P,F	BOOKING#	BOOKING_NBR	VARCHAR	20	10		Booking number
P	ROOMNBR	ROOM_NBR	VARCHAR	10	6		Room number

Key	Field name	Alias name	Data type	Length	VarLen Alloc	Allow null	Description
	STATUS	BOOKED_STATUS	CHAR	1		Y	Room status (O=open, B=booked, D=cruise has sailed)
	PDCRUISE	PD_CRUISE_LINE	CHAR	1		Y	Paid cruise line status (U=unpaid, P=paid)

The port_of_calls_temp table is a temporary table used for decomposing the XML document. The actual data is stored in the port_of_calls table. The port_of_calls table contains the same type of information. The layout of the port_of_calls_temp table is listed in Table 3-3.

Table 3-3 Port_of_calls_temp

Key	Field name	Alias name	Data type	Length	VarLen Alloc	Allow null	Description
	CRUISECO	CRUISE_CO	VARCHAR	40	20		Name of cruise company
	CRUISEID	CRUISE_ID	VARCHAR	20	10		Cruise id
	DAYNBR	DAY_NUMBER	INT				Day number (ie 1, 2, 3, ...)
	PORT	PORT_OF_CALL	VARCHAR	30	10		City name of port
	ARRTIME	ARRIVE_TIME	TIME			Y	Time of arrival at port
	DEPTTIME	DEPART_TIME	TIME			Y	Time of departure at port

The activities_temp table is a temporary table used for decomposing the XML document. The actual data is stored in the activities table. The activities table contains the same type of information. The layout of the activities_temp table is listed in Table 3-4.

Table 3-4 Activities_temp

Key	Field Name	Alias Name	Data Type	Length	VarLen Alloc	Allow Null	Description
	CRUISECO	CRUISE_CO	VARCHAR	40	20		Name of cruise company
	CRUISEID	CRUISE_ID	VARCHAR	20	10		Cruise id
	DAYNBR	DAY_NUMBER	INT				Day number (ie 1, 2, 3, ...)
	ACTTIME	ACTIVITY_TIME	TIME			Y	Time of activity
	DESC	DESCRIPTION	VARCHAR	100	20		Description of activity

The day_temp table contains the day number for the activities and ports of call. The layout of the day_temp table is listed in Table 3-5.

Table 3-5 Day_temp

Key	Field Name	Alias Name	Data Type	Description
	DAYNBR	DAY_NUMBER	INT	Day Number

3.2 Starting XML document

This section shows an example of the original XML document that the travel agency attempts to decompose into the above tables:

```
<?xml version="1.0"?>
   <!DOCTYPE ItineraryInformation SYSTEM "/cruiseinfo/itinerary.dtd">
     <ItineraryInformation orderNumber="123463" company="Cruise Company">
       <CruiseProduct productID="9899">
         <CruiseID>00011</CruiseID>
         <CruiseDescription>5-Day Caribbean</CruiseDescription>
         <RoomDescription>Oceanview</RoomDescription>
         <DepartureDate>10/12/2000</DepartureDate>
         <ReturnDate>10/15/2000></ReturnDate>
         <Duration>4</Duration>
         <SinglePrice>399</SinglePrice>
         <DoublePrice>199</DoublePrice>
         <ExtraOccupantPrice>199</ExtraOccupantPrice>
         <Schedule>
           <Day dayNumber="1">
             <Port location="Miami, Florida" depart="5:00 PM"/>
           </Day>
           <Day dayNumber="2">
             <Port location="AtSea"/>
             <Activity time="7:00 PM">Formal Dinner and Dance at the Windjammer
Cafe</Activity>
           </Day>
           <Day dayNumber="3">
             <Port location="Ocho Rios, Jamaica" arrive="8:00 AM" depart="5:00 PM"/>
             <Activity time="1:30 PM">Optional Tour of historical ruins of Ocho
Rios.</Activity>
             <Activity time="11:00 PM">Fireworks off starboard side of ship</Activity>
           </Day>
           <Day dayNumber="4">
             <Port location="Miami, Florida" arrive="8:30 AM"/>
           </Day>
         </Schedule>
         <NumberOfRooms>5</NumberOfRooms>
         <Rooms>
           <Room roomNumber="111"/>
           <Room roomNumber="112"/>
           <Room roomNumber="113"/>
           <Room roomNumber="114"/>
           <Room roomNumber="115"/>
         </Rooms>
       </CruiseProduct>
       <CruiseProduct productID="9899">
          ........
         <RoomDescription>Premium</RoomDescription>
          ........
       </CruiseProduct>
     </ItineraryInformation>
```

3.3 Discoveries

The following discoveries were encountered while creating the DAD file used for the itinerary XML document:

1. The encoding declaration is required on all XML, DAD, and DTD documents that are not in UTF-8. Otherwise, the parser rejects the documents. For the V5R1 version of DB2 XML extenders, this XML rule is not enforced strictly (however, it will be in the next version). We happened to be already using UTF-8, but we added it to our XML, DAD, and DTD documents for consistency.

   The following encoding declaration was added:

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

2. You must use valid DB2 column names within decomposition DAD files. If you use system column names, a DXXQ021E message is returned: “Table <table_name> does not have column <column_name>.” For example, in the cruise_info table, column name “booking_nbr” is valid in the DAD file, but column name “booking#” is not valid.

3. When decomposing an XML document into a database table, you must consider whether null values are allowed for a particular column. For example:

   – If the column is defined as *NOT NULL*, the element or attribute that has a mapping to that column must be present in the XML document, regardless of whether they have real values. If they do not have real values, they must be specified as an empty element or attribute (for example, <elem></elem> or attrib=””).

   – If the column is defined without the *NOT NULL* specification (null capable), the element or attribute that has a mapping to that column does not need to be present in the XML document.

   The rooms table was changed to allow null values for the status and pdcruise columns since that specific data was not included in the XML document. A trigger program (SetRoomDefaultValues) was then used to set the initial values.

   Note: The SQL source for the trigger program can be found in Section 3.9, “Trigger source” on page 26.

4. When creating a DAD file, only the primary key of a primary-key-foreign-key join condition must have a mapping in the DAD. In the following example, only the mapping for cruise_info is required because the join condition populates the activities_temp and port_of_calls_temp tables with the primary key cruise_co.

   ```
   ...
   <root_node>
    <element_node name="ItineraryInformation">
      <RDB_node>
        <table name="shoreline.cruise_info" key="booking_nbr"/>
        <table name="shoreline.rooms" key="booking_nbr room_nbr"/>
        <table name="shoreline.port_of_calls_temp" key="cruise_co cruise_id"/>
        <table name="shoreline.activities_temp" key="cruise_co cruise_id"/>
        <table name="shoreline.day_temp" key="day_number"/>
      <condition>
        shoreline.cruise_info.booking_nbr = shoreline.rooms.booking_nbr and
        shoreline.cruise_info.cruise_co = shoreline.port_of_calls_temp.cruise_co and
        shoreline.cruise_info.cruise_id = shoreline.port_of_calls_temp.cruise_id and
        shoreline.cruise_info.cruise_co =shoreline.activities_temp.cruise_co and
        shoreline.cruise_info.cruise_id =shoreline.activities_temp.cruise_id and
        shoreline.day_temp.day_number=shoreline.port_of_calls_temp.day_number and
        Shoreline.day_temp.day_number=shoreline.activities_temp.day_number
   ```

```
</condition>
</RDB_node>
 <element_node name="CruiseProduct" multi_occurrence="YES">
 ...
 <attribute_node name="company">
   <RDB_node>
     <table name="shoreline.cruise_info"/>
     <column name="cruise_co" type="varchar(40)"/>
   </RDB_node>
 </attribute_node>
 <!-- the lines below were removed -->
 <attribute_node name="company">
 <RDB_node>
 <table name="shoreline.activities"/>
 <column name="cruise_co" type="varchar(40)"/>
 </RDB_node>
 </attribute_node>
 <attribute_node name="company">
 <RDB_node>
 <table name="shoreline.port_of_calls"/>
 <column name="cruise_co" type="varchar(40)"/>
 </RDB_node>
 </attribute_node>
 <!-- end of lines removed -->
...
```

5. Attribute values or element content in an XML document must populate only one row of a table. To have two rows with the same data, that data must appear twice in the XML document.

   Within our original XML document, orderNumber and company were listed once. Within the cruise_info table, we were expecting this data to be listed for each order. Therefore, to have two rows with order_number="123456" and cruise_co="Cruise Company", that data had to appear twice in the XML document.

   There are three options that we considered:

   a. Have repeating child elements (for example, <CruiseProduct>) correspond to tables that are related via join conditions to the table associated with the parent element (for example, <ItineraryInformation>). The join condition causes a single value in the XML document to populate in more than one place. For example:

```
<?xml version="1.0"?>
<!DOCTYPE ItineraryInformation SYSTEM "/javateam/shoreline/dtd/itinerary.dtd">
<ItineraryInformation>
  <CruiseProduct>
    <OrderNumber>123456</OrderNumber>
    <Company>Cruise Company</Company>
    <ProductID>9899</ProductID>
    ...
  </CruiseProduct>
  <CruiseProduct>
    <OrderNumber>123456</OrderNumber>  <!-- data repeated for 2nd row -->
    <Company>Cruise Company</Company>   <!-- data repeated for 2nd row -->
    <ProductID>9899</ProductID>
   ...
  </CruiseProduct>
</ItineraryInformation>
...
```

b. Keep the original document structure (<ItineraryInformation> as the root element and only one <CruiseProduct> as the child element of <ItineraryInformation>), but use multiple XML documents.

c. Restructure the cruise_info table to resemble the document hierarchy in which <CruiseProduct> appears multiple times within <ItineraryInformation>.

We used option a. and added the child elements <OrderNumber>, <Company>, and <ProductID> to the <CruiseProduct> element. This change included updates to both the DTD and XML document.

6. If you plan to use decomposition and composition together (that is, decompose an XML document into tables, modify the data in the tables, and then run composition to re-create the XML document with the new data), the element and attribute ordering in the DAD file must match the original XML document. If it does not, the element or attribute ordering of the composed XML document will not match that of the original XML document. As a result, one of those two documents does not validate against the DTD.

   In our XML document, the <Port> tag preceded the <Activity> tag. When creating the DAD file, we listed the <Activity> tag first and then the <Port> tag. We changed the ordering within the DAD file to what is listed below so that it matched the XML document:

```
...
<element_node name="Port" >
  <element_node name="location">
   <text_node>
      <RDB_node>
       <table name="shoreline.port_of_calls"/>
       <column name="port_of_call" type="varchar(30)"/>
      </RDB_node>
    </text_node>
  </element_node>
  <element_node name="Activity" multi_occurrence="YES">
    <text_node>
      <RDB_node>
        <table name="shoreline.activities"/>
        <column name="description" type="varchar(100)"/>
      </RDB_node>
    </text_node>
    <attribute_node name="time">
      <RDB_node>
        <table name="shoreline.activities"/>
        <column name="activity_time" type="Time"/>
      </RDB_node>
    </attribute_node>
  </element_node>  <!-- end of element Activity -->
...
```

7. In the DAD file, no option exists to specify that only unique records should be inserted into the database tables. Within our XML document, for each cruise, there is a corresponding set of ports of call and activities listed. When the XML document is decomposed, the appropriate data is inserted into the port_of_calls and activities tables. We want to ensure that only one set of ports of call and activities is inserted into those tables for each cruise because we do not want these tables populated with duplicate records.

   Since there is no option in the DAD file to specify that duplicate rows should not be inserted, we created two temporary tables (port_of_calls_temp and activities_temp). We then set up INSERT triggers (Enter_Port_Of_Calls and Enter_Activities) on those two tables to insert the new row into the port_of_calls and activities tables if they did not

already exist. Two more triggers (Remove_Port_Of_Calls and Remove_Activities) were then used to delete the new row from the port_of_calls_temp and activities_temp tables (these tables were just used as temporary tables; the real data was saved in the port_of_calls and activities tables).

Note: The SQL statements used for the triggers can be found in 3.9, “Trigger source” on page 26.

8. When creating a DAD file, you must determine the structure of the XML document and how it corresponds to the structure of your database table. We took into consideration the handling of the <Day> element and how it mapped to our tables. For each <Day> element, we had zero or more <Port> child elements and zero or more <Activity> child elements. We initially started out with just the port_of_calls_temp and activities_temp tables into which to decompose this data. However, since port_of_calls_temp.day_number and activities_temp.day_number are not related by a join, there was no way to have <Day dayNumber=”...”> map to both columns.

   The element <Day> with its own attribute and two types of child elements, each mapping to tables, suggest that there should be a corresponding table for day (table day_temp). Even though there is no join relationship between port_of_calls_temp.day_number and activities_temp.day_number, the fact that both activity and port are grouped by days suggests that their respective tables are related indirectly (a join relationship between day_temp.day_number and port_of_calls_temp.day_number and a join between day_temp.day_number and activities_temp.day_number). Therefore, we created another table day_temp with a single column day_number and changed the DAD to look like the following example:

```
...
<element_node name="ItineraryInformation">
  <RDB_node>
  <table name="shoreline.cruise_info" key="booking_nbr"/>
  <table name="shoreline.rooms" key="booking_nbr room_nbr"/>
  <table name="shoreline.port_of_calls" key="cruise_co cruise_id"/>
  <table name="shoreline.activities" key="cruise_co cruise_id"/>
  <table name="shoreline.day_temp" key="day_number"/>
<condition>
   shoreline.cruise_info.booking_nbr = shoreline.rooms.booking_nbr and
   shoreline.cruise_info.cruise_co = shoreline.port_of_calls_temp.cruise_co and
   shoreline.cruise_info.cruise_id = shoreline.port_of_calls_temp.cruise_id and
   shoreline.cruise_info.cruise_co = shoreline.activities_temp.cruise_co and
   shoreline.cruise_info.cruise_id = shoreline.activities_temp.cruise_id AND
   shoreline.day_temp.day_nbr=shoreline.port_of_calls_temp.day_number and
   shoreline.day_temp.day_number=shoreline.activities_temp.day_number
  </condition>
 </RDB_node>
<...>
  <element_node name="Schedule" >
   <element_node name="Day" multi_occurrence="YES">
    <attribute_node name="dayNumber">
     <RDB_node>
      <table name="shoreline.day_temp"/>
     <column name="day_number" type="integer"/>
     </RDB_node>
    </attribute_node>
    <element_node name="Port">
      <element_node name="location">
        <RDB_node>
         <table name="shoreline.port_of_calls_temp"/>
```

```
        <column name="port_of_call" type="varchar(30)"/>
       </RDB_node>
    ...
  </element_node>
...
```

To clean up the day_temp table (no data is needed from this table), an INSERT trigger (RemoveRecord) was used to remove all the records from the day_temp table.

Note: The SQL source for this trigger can be found in Section 3.9, “Trigger source” on page 26.

9. When mapping an XML document to a database table, a one-to-one mapping of the XML elements or attributes to the database columns is required. Within the cruise_info and rooms tables, booking_nbr is defined as the primary key. This data is not contained in the XML document that the travel agency receives from the cruise company. The booking_nbr is a unique value that the travel agency generates for each cruise order. Since the booking_nbr was not contained in the original XML document, generate a new booking_nbr, parse the XML document, and modify the Document Object Model (DOM) structure to add the <BookingNumber> element. The modified XML document was then decomposed into the appropriate tables.

   Here is a Java code example that demonstrates how to modify the DOM and add in the <BookingNumber> element:

```
...
public static void addBookNbr(Document document, String bookingNumber) {
  NodeList nl = document.getElementsByTagName("CruiseID");
  for (int i=0 ; i < nl.getLength(); i++) {
    Element bookNbr = document.createElement("BookingNumber"); //add BookingNumber
element
    BookNbr.appendChild(document.createTextNode(bookingNumber)); //add the number to
element
    Node bookingNbr = (Node)bookNbr;
    Node parent = nl.item(i);
    try {
      parent.insertBefore(bookingNbr, parent.getFirstChild());//insert into the DOM
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
}
...
```

3.4 Final DTD

After incorporating all the discoveries listed above, the final DTD document appears as it is shown here:

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

<!ELEMENT ItineraryInformation (CruiseProduct)+>

<!ELEMENT CruiseProduct (OrderNumber, Company, ProductID, BookingNumber*, CruiseID,
CruiseDescription, RoomDescription, DepartureDate, ReturnDate, Duration, SinglePrice,
DoublePrice, ExtraOccupantPrice, Schedule, NumberOfRooms, Rooms)>
```

```
<!ELEMENT OrderNumber (#PCDATA)>
<!ELEMENT Company (#PCDATA)>
<!ELEMENT ProductID (#PCDATA)>
<!ELEMENT BookingNumber (#PCDATA)>
<!ELEMENT CruiseID (#PCDATA)>
<!ELEMENT CruiseDescription (#PCDATA)>
<!ELEMENT RoomDescription (#PCDATA)>
<!ELEMENT DepartureDate (#PCDATA)>
<!ELEMENT ReturnDate (#PCDATA)>
<!ELEMENT Duration (#PCDATA)>
<!ELEMENT SinglePrice (#PCDATA)>
<!ELEMENT DoublePrice (#PCDATA)>
<!ELEMENT ExtraOccupantPrice (#PCDATA)>
<!ELEMENT Schedule (Day)+>
<!ELEMENT NumberOfRooms (#PCDATA)>
<!ELEMENT Rooms (Room)+>

<!ELEMENT Day (Port*, Activity*)>
<!ATTLIST Day dayNumber CDATA #REQUIRED>

<!ELEMENT Port (#PCDATA)>
<!ATTLIST Port
  location  CDATA #REQUIRED
  arrive CDATA #IMPLIED
  depart CDATA #IMPLIED
>

<!ELEMENT Activity (#PCDATA)>
<!ATTLIST Activity time CDATA #IMPLIED>

<!ELEMENT Room (#PCDATA)>
<!ATTLIST Room roomNumber CDATA #REQUIRED>
```

3.5 Final XML document

This section shows an example of what the final XML document looks like after incorporating all the discoveries:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ItineraryInformation SYSTEM "/javateam/shoreline/dtd/itinerary.dtd">

<ItineraryInformation>

  <CruiseProduct>
    <OrderNumber>123463</OrderNumber>
    <Company>Cruise Company</Company>
    <ProductID>9899</ProductID>
    <BookingNumber>105</BookingNumber>
    <CruiseID>00011</CruiseID>
    <CruiseDescription>5-Day Caribbean</CruiseDescription>
    <RoomDescription>Oceanview</RoomDescription>
    <DepartureDate>10/12/2000</DepartureDate>
    <ReturnDate>10/15/2000></ReturnDate>
    <Duration>4</Duration>
    <SinglePrice>399</SinglePrice>
    <DoublePrice>199</DoublePrice>
    <ExtraOccupantPrice>199</ExtraOccupantPrice>
    <Schedule>
      <Day dayNumber="1">
```

```
      <Port location="Miami, Florida" depart="5:00 PM"/>
    </Day>
    <Day dayNumber="2">
      <Activity time="7:00 PM">Formal Dinner and Dance at the Windjammer Cafe</Activity>
    </Day>
    <Day dayNumber="3">
      <Port location="Ocho Rios, Jamaica" arrive="8:00 AM" depart="5:00 PM"/>
      <Activity time="1:30 PM">Optional Tour of historical ruins of Ocho Rios.</Activity>
      <Activity time="11:00 PM">Fireworks off starboard side of ship</Activity>
    </Day>
    <Day dayNumber="4">
      <Port location="Miami, Florida" arrive="8:30 AM"/>
    </Day>
  </Schedule>
  <NumberOfRooms>5</NumberOfRooms>
    <Rooms>
      <Room roomNumber="111"/>
      <Room roomNumber="112"/>
      <Room roomNumber="113"/>
      <Room roomNumber="114"/>
      <Room roomNumber="115"/>
   </Rooms>
 </CruiseProduct>
 <CruiseProduct>
   <OrderNumber>123463</OrderNumber>
   <Company>Cruise Company</Company>
   <ProductID>9899</ProductID>
   <BookingNumber>106</BookingNumber>
   ...
     <Schedule>
     ...
     </Schedule>
   ...
 </CruiseProduct>
</ItineraryInformation>
```

3.6 DTD mapping scheme

Figure 3-1 illustrates how the structure of the XML document relates to the DB2 tables that are used for the decomposition. This is helpful in the creation of the DAD file because it shows how the two structures compare.

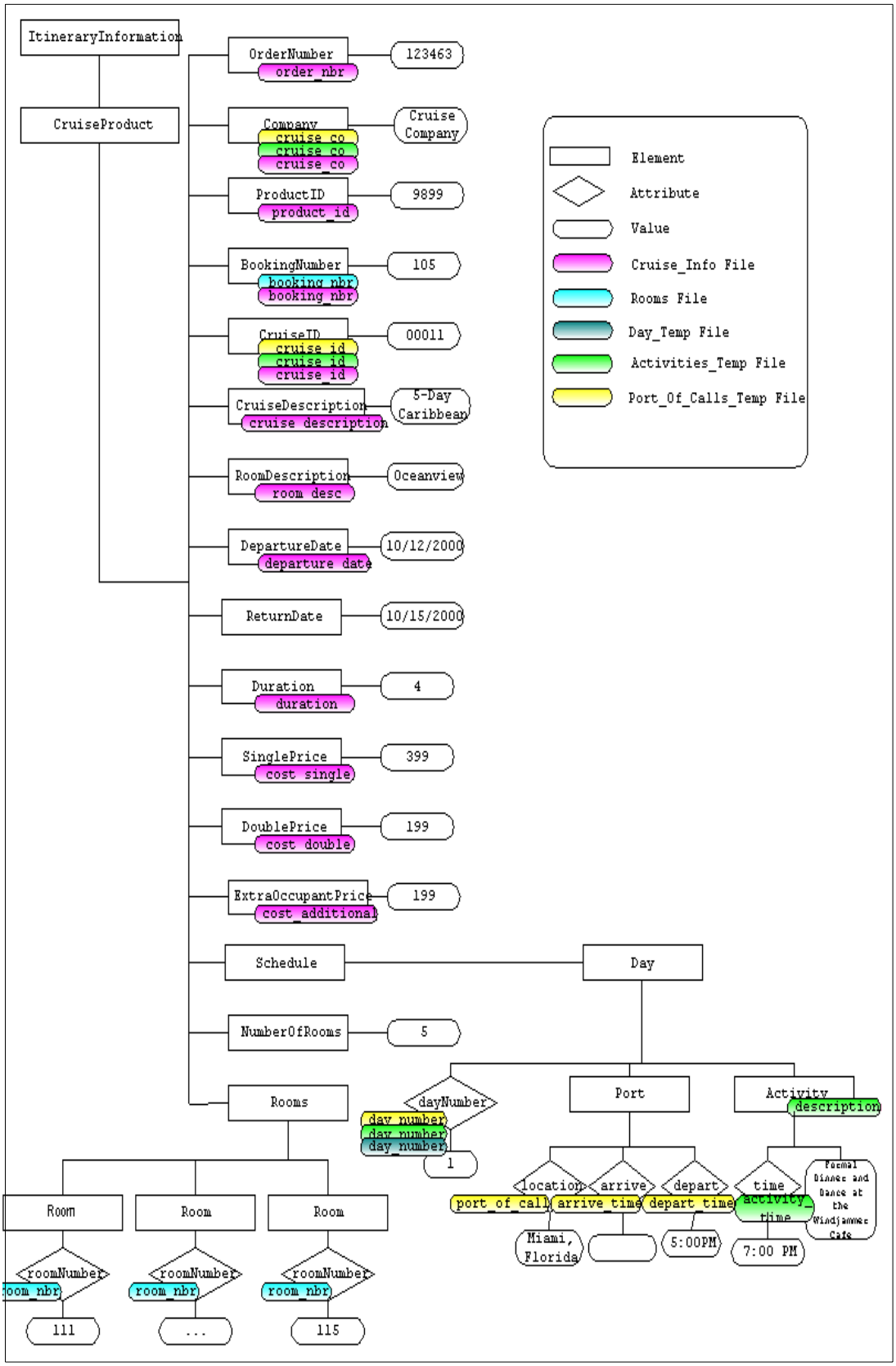


Figure 3-1 Itinerary mapping scheme

3.7 Final DAD

This section shows the final DAD document after incorporating all the discoveries. This DAD file for an XML collection uses RDB_node mapping:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DAD SYSTEM "/javateam/shoreline/dtd/dad.dtd">

<DAD>
<dtdid>/javateam/shoreline/dtd/itinerary.dtd</dtdid>
<validation>YES</validation>

<Xcollection>
<prolog>?xml version="1.0"?</prolog>
<doctype>!DOCTYPE ItineraryInformation SYSTEM
"/javateam/shoreline/dtd/itinerary.dtd"</doctype>

<root_node>
<element_node name="ItineraryInformation">
 <RDB_node>
  <table name="shoreline.cruise_info" key="booking_nbr"/>
  <table name="shoreline.rooms" key="booking_nbr room_nbr"/>
  <table name="shoreline.port_of_calls_temp" key="cruise_co cruise_id"/>
  <table name="shoreline.activities_temp" key="cruise_co cruise_id"/>
  <table name="shoreline.day_temp" key="day_number"/>
  <condition>
   shoreline.cruise_info.booking_nbr = shoreline.rooms.booking_nbr and
   shoreline.cruise_info.cruise_co = shoreline.port_of_calls_temp.cruise_co and
   shoreline.cruise_info.cruise_id = shoreline.port_of_calls_temp.cruise_id and
   shoreline.cruise_info.cruise_co =shoreline.activities_temp.cruise_co and
   shoreline.cruise_info.cruise_id =shoreline.activities_temp.cruise_id and
   shoreline.day_temp.day_number=shoreline.port_of_calls_temp.day_number and
   shoreline.day_temp.day_number=shoreline.activities_temp.day_number
  </condition>
 </RDB_node>
 <element_node name="CruiseProduct" multi_occurrence="YES">
    <element_node name="OrderNumber">
    <text_node>
      <RDB_node>
        <table name="shoreline.cruise_info"/>
        <column name="order_number" type="varchar(20)"/>
      </RDB_node>
    </text_node>
    </element_node>

    <element_node name="Company">
    <text_node>
      <RDB_node>
        <table name="shoreline.cruise_info"/>
        <column name="cruise_co" type="varchar(40)"/>
      </RDB_node>
    </text_node>
    </element_node>
    <element_node name="ProductID">
    <text_node>
      <RDB_node>
          <table name="shoreline.cruise_info"/>
     <column name="product_id" type="varchar(20)"/>
      </RDB_node>
    </text_node>
```

```
  </element_node>
<element_node name="BookingNumber">
<text_node>
 <RDB_node>
  <table name="shoreline.cruise_info"/>
  <column name="booking_nbr" type="varchar(20)"/>
 </RDB_node>
</text_node>
</element_node>
<element_node name="CruiseID">
<text_node>
 <RDB_node>
  <table name="shoreline.cruise_info"/>
  <column name="cruise_id" type="varchar(20)"/>
 </RDB_node>
</text_node>
</element_node>
<element_node name="CruiseDescription">
<text_node>
 <RDB_node>
  <table name="shoreline.cruise_info"/>
  <column name="cruise_desc" type="varchar(30)"/>
 </RDB_node>
</text_node>
</element_node>
<element_node name="RoomDescription">
<text_node>
 <RDB_node>
  <table name="shoreline.cruise_info"/>
  <column name="room_desc" type="varchar(30)"/>
 </RDB_node>
</text_node>
</element_node>
<element_node name="DepartureDate">
<text_node>
 <RDB_node>
  <table name="shoreline.cruise_info"/>
  <column name="departure_date" type="Date"/>
 </RDB_node>
</text_node>
</element_node>
<element_node name="Duration">
<text_node>
 <RDB_node>
  <table name="shoreline.cruise_info"/>
  <column name="duration" type="integer"/>
 </RDB_node>
</text_node>
</element_node>
<element_node name="SinglePrice">
<text_node>
 <RDB_node>
  <table name="shoreline.cruise_info"/>
  <column name="cost_single" type="Decimal"/>
 </RDB_node>
</text_node>
</element_node>
<element_node name="DoublePrice">
<text_node>
 <RDB_node>
```

```
  <table name="shoreline.cruise_info"/>
  <column name="cost_double" type="Decimal"/>
 </RDB_node>
</text_node>
</element_node>
<element_node name="ExtraOccupantPrice">
<text_node>
 <RDB_node>
  <table name="shoreline.cruise_info"/>
  <column name="cost_additional" type="Decimal"/>
 </RDB_node>
</text_node>
</element_node>
<element_node name="Schedule">
 <element_node name="Day" multi_occurrence="YES">
  <attribute_node name="dayNumber">
   <RDB_node>
    <table name="shoreline.day_temp"/>
    <column name="day_number" type="integer"/>
   </RDB_node>
  </attribute_node>
  <element_node name="Port">
   <attribute_node name="location">
    <RDB_node>
     <table name="shoreline.port_of_calls_temp"/>
     <column name="port_of_call" type="varchar(30)"/>
    </RDB_node>
   </attribute_node>
   <attribute_node name="arrive">
    <RDB_node>
     <table name="shoreline.port_of_calls_temp"/>
     <column name="arrive_time" type="Time"/>
    </RDB_node>
   </attribute_node>
   <attribute_node name="depart">
    <RDB_node>
     <table name="shoreline.port_of_calls_temp"/>
     <column name="depart_time" type="Time"/>
    </RDB_node>
   </attribute_node>
  </element_node>  <!-- end of element Port -->
   <element_node name="Activity" multi_occurence="YES">
   <text_node>
    <RDB_node>
     <table name="shoreline.activities_temp"/>
     <column name="description" type="varchar(100)"/>
    </RDB_node>
   </text_node>
   <attribute_node name="time">
    <RDB_node>
     <table name="shoreline.activities_temp"/>
     <column name="activity_time" type="Time"/>
    </RDB_node>
   </attribute_node>
  </element_node>  <!-- end of element Activity -->
 </element_node>  <!-- end of element Day -->
</element_node>  <!-- end of element Schedule-->
<element_node name="Rooms">
 <element_node name="Room" multi_occurrence="YES">
  <attribute_node name="roomNumber">
```

```
     <RDB_node>
      <table name="shoreline.rooms"/>
      <column name="room_nbr" type="varchar(10)"/>
     </RDB_node>
    </attribute_node>
   </element_node>  <!-- end of element Room -->
  </element_node>  <!-- end of element Rooms -->
 </element_node>  <!-- end of element CruiseProduct -->
</element_node>  <!-- end of element ItineraryInformation -->
</root_node>
</Xcollection>
</DAD>
```

3.8 XML enablement commands

This section lists the commands that must be run to decompose the Itinerary.xml document into the database tables:

1. When using the DTD to validate XML data in the XML collection (VALIDATION set to YES in the DAD file), the following SQL statement must be issued to store the Itinerary.dtd in the DTD repository table:

   ```
   INSERT INTO DB2XML.DTD_REF VALUES('/javateam/shoreline/dtd/itinerary.dtd',
   DB2XML.XMLCLOBFROMFILE('/javateam/shoreline/dtd/itinerary.dtd'), 0, 'author_name',
   'creator_name', NULL)
   ```

2. Since the dxxInsertXML() stored procedure is used to decompose the document, enable the XML collection using the following command:

   ```
   CALL PGM(QDBXM/QZXMADM) PARM (ENABLE_COLLECTION db_name SHORELINE.CRUISE_INFO
   '/javateam/shoreline/dad/itinerary.dad')
   ```

3.9 Trigger source

The SQL statements used to add the trigger programs to the specified tables are listed in Table 3-6.

Table 3-6 Trigger source

Table	Description	SQL statement
CRUISE_INFO	After insert, sets the commission rate for the new row.	`CREATE TRIGGER SHORELINE/SET_COMMISSION_RATE AFTER INSERT ON SHORELINE/CRUISE_INFO REFERENCING NEW AS NEWROW FOR EACH ROW MODE DB2ROW BEGIN ATOMIC UPDATE SHORELINE/CRUISE_INFO SET CUST_COST = .15 WHERE BOOKING_NBR = NEWROW.BOOKING_NBR`
DAY_TEMP	After insert, deletes records from the table.	`CREATE TRIGGER SHORELINE/REMOVERECORD AFTER INSERT ON SHORELINE/DAY_TEMP BEGIN ATOMIC DELETE FROM SHORELINE/DAY_TEMP; END`

Table	Description	SQL statement
PORT_OF_CALLS_TEMP	After insert, adds distinct records to Port_Of_Calls table.	CREATE TRIGGER SHORELINE/ENTER_PORT_OF_CALLS AFTER INSERT ON SHORELINE/PORT_OF_CALLS_TEMP REFERENCING NEW AS NEW_ROW FOR EACH ROW MODE DB2ROW BEGIN IF NEW_ROW.ARRIVE_TIME IS NULL AND NEW_ROW.DEPART_TIME IS NULL THEN INSERT INTO SHORELINE/PORT_OF_CALLS (SELECT X.CRUISE_CO, X.CRUISE_ID, X.DAY_NUMBER, X.PORT_OF_CALL, X.ARRIVE_TIME, X.DEPART_TIME FROM SHORELINE/PORT_OF_CALLS_TEMP X EXCEPTION JOIN SHORELINE/PORT_OF_CALLS Y ON X.CRUISE_CO = Y.CRUISE_CO AND X.CRUISE_ID = Y.CRUISE_ID AND X.DAY_NUMBER = Y.DAY_NUMBER AND X.PORT_OF_CALL = Y.PORT_OF_CALL AND X.ARRIVE_TIME IS NULL AND X.DEPART_TIME IS NULL); ELSEIF NEW_ROW.ARRIVE_TIME IS NULL THEN INSERT INTO SHORELINE/PORT_OF_CALLS (SELECT X.CRUISE_CO, X.CRUISE_ID, X.DAY_NUMBER, X.PORT_OF_CALL, X.ARRIVE_TIME, X.DEPART_TIME FROM SHORELINE/PORT_OF_CALLS_TEMP X EXCEPTION JOIN SHORELINE/PORT_OF_CALLS Y ON X.CRUISE_CO = Y.CRUISE_CO AND X.CRUISE_ID = Y.CRUISE_ID AND X.DAY_NUMBER = Y.DAY_NUMBER AND X.PORT_OF_CALL = Y.PORT_OF_CALL AND X.ARRIVE_TIME IS NULL AND X.DEPART_TIME = Y.DEPART_TIME); ELSEIF NEW_ROW.DEPART_TIME IS NULL THEN INSERT INTO SHORELINE/PORT_OF_CALLS (SELECT X.CRUISE_CO, X.CRUISE_ID, X.DAY_NUMBER, X.PORT_OF_CALL, X.ARRIVE_TIME, X.DEPART_TIME FROM SHORELINE/PORT_OF_CALLS_TEMP X EXCEPTION JOIN SHORELINE/PORT_OF_CALLS Y ON X.CRUISE_CO = Y.CRUISE_CO AND X.CRUISE_ID = Y.CRUISE_ID AND X.DAY_NUMBER = Y.DAY_NUMBER AND X.PORT_OF_CALL = Y.PORT_OF_CALL AND X.ARRIVE_TIME = Y.ARRIVE_TIME AND X.DEPART_TIME IS NULL); ELSEIF NEW_ROW.ARRIVE_TIME IS NOT NULL AND NEW_ROW.DEPART_TIME IS NOT NULL THEN INSERT INTO SHORELINE/PORT_OF_CALLS (SELECT X.CRUISE_CO, X.CRUISE_ID, X.DAY_NUMBER, X.PORT_OF_CALL, X.ARRIVE_TIME, X.DEPART_TIME FROM SHORELINE/PORT_OF_CALLS_TEMP X EXCEPTION JOIN SHORELINE/PORT_OF_CALLS Y ON X.CRUISE_CO = Y.CRUISE_CO AND X.CRUISE_ID = Y.CRUISE_ID AND X.DAY_NUMBER = Y.DAY_NUMBER AND X.PORT_OF_CALL = Y.PORT_OF_CALL AND X.ARRIVE_TIME = Y.ARRIVE_TIME AND X.DEPART_TIME = Y.DEPART_TIME); END IF; END
PORT_OF_CALLS_TEMP	After insert, deletes record from Port_Of_Calls_Temp table.	CREATE TRIGGER SHORELINE/REMOVE_PORT_OF_CALLS AFTER INSERT ON SHORELINE/PORT_OF_CALLS_TEMP REFERENCING NEW AS NEW_ROW FOR EACH ROW MODE DB2ROW BEGIN DELETE FROM SHORELINE/PORT_OF_CALLS_TEMP; END
ACTIVITIES_TEMP	After insert, adds distinct records to the Activities table.	CREATE TRIGGER SHORELINE/ENTER_ACTIVITIES AFTER INSERT ON SHORELINE/ACTIVITIES_TEMP REFERENCING NEW AS NEW_ROW FOR EACH ROW MODE DB2ROW BEGIN IF NEW_ROW.ACTIVITY_TIME IS NULL THEN INSERT INTO SHORELINE/ACTIVITIES (SELECT X.CRUISE_CO, X.CRUISE_ID, X.DAY_NUMBER, X.ACTIVITY_TIME, X.DESCRIPTION FROM SHORELINE/ACTIVITIES_TEMP X EXCEPTION JOIN SHORELINE/ACTIVITIES Y ON X.CRUISE_CO = Y.CRUISE_CO AND X.CRUISE_ID = Y.CRUISE_ID AND X.DAY_NUMBER = Y.DAY_NUMBER AND X.ACTIVITY_TIME IS NULL AND X.DESCRIPTION = Y.DESCRIPTION); else INSERT INTO SHORELINE/ACTIVITIES (SELECT X.CRUISE_CO, X.CRUISE_ID, X.DAY_NUMBER, X.ACTIVITY_TIME, X.DESCRIPTION FROM SHORELINE/ACTIVITIES_TEMP X EXCEPTION JOIN SHORELINE/ACTIVITIES Y ON X.CRUISE_CO = Y.CRUISE_CO AND X.CRUISE_ID = Y.CRUISE_ID AND X.DAY_NUMBER = Y.DAY_NUMBER AND X.ACTIVITY_TIME = Y.ACTIVITY_TIME AND X.DESCRIPTION = Y.DESCRIPTION); END IF; END

Table	Description	SQL statement
ACTIVITIES_TEMP	After insert, deletes record from the Activities_ Temp table.	CREATE TRIGGER SHORELINE/REMOVE_ACTIVITIES AFTER INSERT ON SHORELINE/ACTIVITIES_TEMP REFERENCING NEW AS NEW_ROW FOR EACH ROW MODE DB2ROW BEGIN DELETE FROM SHORELINE/ACTIVITIES_TEMP; END
ROOMS	Before insert, checks to see whether values are NULL. If so, set them to specified values (‘O’ and ‘U’).	CREATE TRIGGER SHORELINE/SETROOMDEFAULTVALUES BEFORE INSERT ON SHORELINE/ROOMS REFERENCING NEW AS NEW_ROW FOR EACH ROW MODE DB2ROW BEGIN IF NEW_ROW.BOOKED_STATUS IS NULL THEN SET NEW_ROW.BOOKED_STATUS='O'; END IF; IF NEW_ROW.PD_CRUISE_LINE IS NULL THEN SET NEW_ROW.PD_CRUISE_LINE = ‘U’; END IF; END

3.10 Java source

The following code snippet decomposes the itinerary XML document into the database tables with the defined DAD:

```
...
// Read the XML file
String = null;
byte buf[] = new byte[5000];
try
{
  FileInputStream in = new FileInputStream("/javateam/shoreline/xml/TestItinerary.xml");
  in.read(buf, 0, 5000);
}
catch (Exception e)
{
  System.out.println("Error: " + e.toString());
}
xmlItinerary = new String(buf, 0);

// Call the stored procedure
cs = con.prepareCall("CALL db2xml.dxxInsertXML(?, ?, ?, ?)");

// Register the input parameter: collection name
cs.setString(1, "SHORELINE.CRUISE_INFO");

// Register the input parameter: xml document
cs.setObject(2, xmlItinerary);

// Register the output parameter: return code
cs.registerOutParameter(3, Types.INTEGER);

// Register the output parameter: return message
cs.registerOutParameter(4, Types.VARCHAR);

// Run the stored procedure
cs.execute();
System.out.println("return code: " + cs.getInt(3));
System.out.println("message text: " + cs.getString(4));
...
```

4

Passengers

This chapter describes the XML Extender pieces of the passenger application that allows the travel agency to send a passenger XML document to the cruise company. The passenger XML document contains all the passenger information for the booked rooms that have been reserved by customers of the travel agency. The passenger XML document is composed from two DB2 database tables by using SQL mapping.

The travel agency uses an XML collection for storage of the XML data and a custom designed DTD for the passenger XML document since there are no suitable industry standard DTDs available.

© Copyright IBM Corp. 2001

4.1 Database details

The XML document is composed of data from two existing DB2 tables. The two DB2 tables are cruise_info and passengers. The xml_passenger table is used to store the XML document.

The cruise_info table contains cruise information that is received from the cruise company. The layout of the cuise_info table is listed in Table 4-1.

Table 4-1 Cruise_Info

Key	Field name	Alias name	Data type	Length	VarLen Alloc	Digits, DecPos	Description
P	BOOKING#	BOOKING_NBR	VARCHAR	20	10		Booking number
	CRUISECO	CRUISE_CO	VARCHAR	40	20		Cruise company name
	ORDER#	ORDER_NBR	VARCHAR	20	10		Order number
	PRODID	PRODUCT_ID	VARCHAR	20	10		Product id
	CRUISEID	CRUISE_ID	VARCHAR	20	10		Cruise id
	CRUISEDESC	CRUISE_DESC	VARCHAR	30	10		Cruise description (7 Day...)
	ROOMDESC	ROOM_DESC	VARCHAR	30	10		Room description (std, ocean, ...)
	COSTSING	COST_SINGLE	PACKED			7,2	Cost to the agency for single occupancy
	COSTDOUB	COST_DOUBLE	PACKED			7,2	Double price
	COSTADD	COST_ADDITIONAL	PACKED			7,2	Cost for each additional person over 2
	CUSTCOST	CUST_COST	PACKED			2,2	Cost to the customer (commission added) (percentage)
	DUR	DURATION	INT				Duration of cruise
	DEPTDATE	DEPARTURE_DATE	DATE				Departure date of cruise

The passengers table contains passenger information for a specific room on a cruise. The layout of the passengers table is listed in Table 4-2.

Table 4-2 Passengers

Key	Field name	Alias name	Data type	Length	VarLen Alloc	Allow null	Description
F	BOOKING#	BOOKING_NBR	VARCHAR	20	10		Booking number
F	ROOMNBR	ROOM_NBR	VARCHAR	10	6		Room number
	NAME1	NAME_1	VARCHAR	50	30		Passenger 1 name
	ADDR1	ADDRESS_1	VARCHAR	40	20		Passenger 1 address

Key	Field name	Alias name	Data type	Length	VarLen Alloc	Allow null	Description
	CITY1	CITY_1	VARCHAR	15	10		Passenger 1 city
	STATE1	STATE_1	CHAR	2			Passenger 1 state
	ZIP1	ZIP_1	CHAR	9			Passenger 1 zip
	COUNTRY1	COUNTRY_1	CHAR	15			Passenger 1 country
	PHONENBR1	PHONE_NBR_1	CHAR	14			Passenger 1 phone number
	DOB1	DATE_OF_BIRTH_1	DATE				Passenger 1 date of birth
	NAME2	NAME_2	VARCHAR	50	30	Y	Passenger 2 name
	ADDR2	ADDRESS_2	VARCHAR	40	20	Y	Passenger 2 address
	CITY2	CITY_2	VARCHAR	15	10	Y	Passenger 2 city
	STATE2	STATE_2	CHAR	2		Y	Passenger 2 state
	ZIP2	ZIP_2	CHAR	9		Y	Passenger 2 zip
	COUNTRY2	COUNTRY_2	CHAR	15		Y	Passenger 2 country
	PHONENBR2	PHONE_NBR_2	CHAR	14		Y	Passenger 2 phone number
	DOB2	DATE_OF_BIRTH_2	DATE			Y	Passenger 2 date of birth
	NAME3	NAME_3	VARCHAR	50	30	Y	Passenger 3 name
	ADDR3	ADDRESS_3	VARCHAR	40	20	Y	Passenger 3 address
	CITY3	CITY_3	VARCHAR	15	10	Y	Passenger 3 city
	STATE3	STATE_3	CHAR	2		Y	Passenger 3 state
	ZIP3	ZIP_3	CHAR	9		Y	Passenger 3 zip
	COUNTRY3	COUNTRY_3	CHAR	15		Y	Passenger 3 country
	PHONENBR3	PHONE_NBR_3	CHAR	14		Y	Passenger 3 phone number
	DOB3	DATE_OF_BIRTH_3	DATE			Y	Passenger 3 date of birth
	NAME4	NAME_4	VARCHAR	50	30	Y	Passenger 4 name

Key	Field name	Alias name	Data type	Length	VarLen Alloc	Allow null	Description
	ADDR4	ADDRESS_4	VARCHAR	40	20	Y	Passenger 4 address
	CITY4	CITY_4	VARCHAR	15	10	Y	Passenger 4 city
	STATE4	STATE_4	CHAR	2		Y	Passenger 4 state
	ZIP4	ZIP_4	CHAR	9		Y	Passenger 4 zip
	COUNTRY4	COUNTRY_4	CHAR	15		Y	Passenger 4 country
	PHONENBR4	PHONE_NBR_4	CHAR	14		Y	Passenger 4 phone number
	DOB4	DATE_OF_BIRTH_4	DATE			Y	Passenger 4 date of birth
	PROCIND	PROCESSED_INDICATOR	CHAR	1			Indicates if passengers list have been sent to cruise company (N=not sent, P=sent)

The xml_passenger table is used to store the XML document content. This table is used for composing the passenger XML document. The layout of the xml_passenger table is listed in Table 4-3.

Table 4-3 XML_passenger

Key	Field name	Alias name	Data type	Description
	VALID00001	VALID_DOCUMENT	INT	Valid document indicator
	XML_D00001	XML_DOCUMENT	XMLCLOB	Passengers XML document

4.2 Starting XML document

This section shows an example of the original XML document that the travel agency attempts to compose using tables presented in the previous section:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE PassengerUpdate SYSTEM "/javateam/shoreline/dtd/passengerUpdate.dtd">
<PassengerUpdate agency="Shoreline Travel">
  <Confirmation sendConfirm="No"></Confirmation>
  <Cruise cruiseID="1">
    <Room roomNumber="1    ">
      <Passenger1 name="Jane Doe" address="123 Main St, City, State 11111"/>
      <Passenger2 name="John Doe" address="123 Main St, City, State 11111"/>
      <Passenger3></Passenger3>
      <Passenger4></Passenger4>
    </Room>
    <Room roomNumber="3    ">
      <Passenger1 name="Al Jones" address="234 Main St, City, State 33333"/>
      <Passenger2></Passenger2>
      <Passenger3></Passenger3>
      <Passenger4></Passenger4>
```

```
   </Room>
  </Cruise>
  <Cruise cruiseID="6">
    <Room roomNumber="2    ">
      <Passenger1 name="Amanda Smith" address="987 Main St, City, State 55555"/>
      <Passenger2 name="Joe Smith" address="987 Main St, City, State 55555"/>
      <Passenger3 name="Ashley Doe" address="123 Elmwood Dr., City, State 22222"/>
      <Passenger4 name="Bill Doe" address="123 Elmwood Dr., City, State 22222"/>
   </Room>
  </Cruise>
</PassengerUpdate>
```

4.3 Discoveries

This section lists some discoveries we encountered while creating the DAD file used for the passenger XML document:

1. As documented in the *DB2 Universal Database for iSeries XML Extender Administration and Programming Guide* (Chapter 2, “Creating the XML collection: Preparing the DAD file”):
   - Columns listed in the SQL statement have to be specified in top-down order according to the hierarchy of the XML document structure.
   - Columns belonging to the same table must be grouped together.
   - Columns that represent the object ID, the first column in each group, are listed in top-down order in the ORDER BY clause.

   Our goal was for the rooms for each cruiseID to be separate. When we first created the DAD, all of the rooms were placed in one cruise.

   There is a problem in the SQL statement of our DAD file. It lacks the agency_name from the ORDER BY clause:

   ```
   ...
   <SQL_stmt>
      SELECT 'Shoreline Travel' AS AGENCY_NAME, 'No' AS SEND_CONFIRM, CRUISE_ID,
      SHORELINE.PASSENGERS.BOOKING_NBR as BOOK, ROOM_NBR, NAME_1, (ADDRESS_1 CONCAT ', '
      CONCAT CITY_1 CONCAT ', ' CONCAT STATE_1 CONCAT ' ' CONCAT ZIP_1) AS
      COMPLETE_ADDRESS_1, NAME_2, (ADDRESS_2 CONCAT ', ' CONCAT CITY_2 CONCAT ', ' CONCAT
      STATE_2 CONCAT ' ' CONCAT ZIP_2) AS COMPLETE_ADDRESS_2, NAME_3, (ADDRESS_3 CONCAT ',
      ' CONCAT CITY_3 CONCAT ', ' CONCAT STATE_3 CONCAT ' ' CONCAT ZIP_3) AS
      COMPLETE_ADDRESS_3, NAME_4, (ADDRESS_4 CONCAT ', ' CONCAT CITY_4 CONCAT ', ' CONCAT
      STATE_4 CONCAT ' ' CONCAT ZIP_4) AS COMPLETE_ADDRESS_4 FROM SHORELINE.CRUISE_INFO,
      SHORELINE.PASSENGERS WHERE CRUISE_CO = 'Cruise Company' AND PROCESSED_INDICATOR =
      'N' AND SHORELINE.CRUISE_INFO.BOOKING_NBR = SHORELINE.PASSENGERS.BOOKING_NBR ORDER
      BY CRUISE_ID, BOOK, ROOM_NBR
   </SQL_stmt>
   ...
   ```

   The first two conditions of the documentation are met, but there is an object ID missing from the ORDER BY clause:

   ```
   ORDER BY AGENCY_NAME, CRUISE_ID, BOOK, ROOM_NBR
   ```

   Cruise_id is really the second rather than the first object ID because the <Cruise> element repeats within the <PassengerUpdate> element. The first object ID is represented by the dummy column agency_name.

2. The content() user-defined function writes output to the Integrated File System (IFS) using the character set defined for the job. To generate the XML document, we used dxxRetrieveXML() and CONTENT(). We then attempted to send this via e-mail to another

user. When we tried to read it within our Java application using FileInputStream, we received garbage characters because the file was stored in EBCDIC.

To force Java to convert the file to ASCII, we used the IFSTextFileInputStream API included in the AS/400 Toolbox for Java.

Note: An example of this source is contained in 4.9, “Java source” on page 38.

3. When calling the dxxRetrieveXML() stored procedure, the XML collection name must match the name contained in the xml_usage table in the library DB2XML. The name is case sensitive.

4.4 Final DTD

After incorporating all the discoveries listed above, the final DTD document appears as it is shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--  This DTD defines the format for sending passenger updates from a travel agency to a
cruise line company. -->

<!ELEMENT PassengerUpdate (Confirmation?, Cruise+)>
<!ATTLIST PassengerUpdate agency CDATA #REQUIRED>

<!ELEMENT Cruise (Room)+>
<!ATTLIST Cruise cruiseID CDATA #REQUIRED>

<!ELEMENT Confirmation (URL?)>
<!ATTLIST Confirmation sendConfirm (Yes | No) "No">

<!ELEMENT URL (#PCDATA)>

<!ELEMENT Room (Passenger1+, Passenger2?, Passenger3?, Passenger4?)>
<!ATTLIST Room roomNumber CDATA #REQUIRED>

<!ELEMENT Passenger1 (#PCDATA)>
<!ATTLIST Passenger1
  name    CDATA #REQUIRED
  address CDATA #REQUIRED
>

<!ELEMENT Passenger2 (#PCDATA)>
<!ATTLIST Passenger2
  name    CDATA #IMPLIED
  address CDATA #IMPLIED
>

<!ELEMENT Passenger3 (#PCDATA)>
<!ATTLIST Passenger3
  name    CDATA #IMPLIED
  address CDATA #IMPLIED
>

<!ELEMENT Passenger4 (#PCDATA)>
<!ATTLIST Passenger4
  name    CDATA #IMPLIED
  address CDATA #IMPLIED
>
```

4.5 Final XML document

This section shows an example of what the final XML document looks like after incorporating all of the discoveries:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE PassengerUpdate SYSTEM "/javateam/shoreline/dtd/passengerUpdate.dtd">
<PassengerUpdate agency="Shoreline Travel">
  <Confirmation sendConfirm="No"></Confirmation>
  <Cruise cruiseID="1">
    <Room roomNumber="1    ">
      <Passenger1 name="Jane Doe" address="123 Main St, City, State 11111"/>
      <Passenger2 name="John Doe" address="123 Main St, City, State 11111"/>
      <Passenger3></Passenger3>
      <Passenger4></Passenger4>
    </Room>
    <Room roomNumber="3    ">
      <Passenger1 name="Al Jones" address="234 Main St, City, State 33333"/>
      <Passenger2></Passenger2>
      <Passenger3></Passenger3>
      <Passenger4></Passenger4>
    </Room>
  </Cruise>
  <Cruise cruiseID="6">
    <Room roomNumber="2    ">
      <Passenger1 name="Amanda Smith" address="987 Main St, City, State 55555"/>
      <Passenger2 name="Joe Smith" address="987 Main St, City, State 55555"/>
      <Passenger3 name="Ashley Doe" address="123 Elmwood Dr., City, State 22222"/>
      <Passenger4 name="Bill Doe" address="123 Elmwood Dr., City, State 22222"/>
   </Room>
  </Cruise>
</PassengerUpdate>
```

4.6 DTD mapping scheme

Figure 4-1 on page 36 illustrates how the structure of the XML document relates to the DB2 tables that are used for the composition. This was helpful in the creation of the DAD file because it shows how the two structures compare.

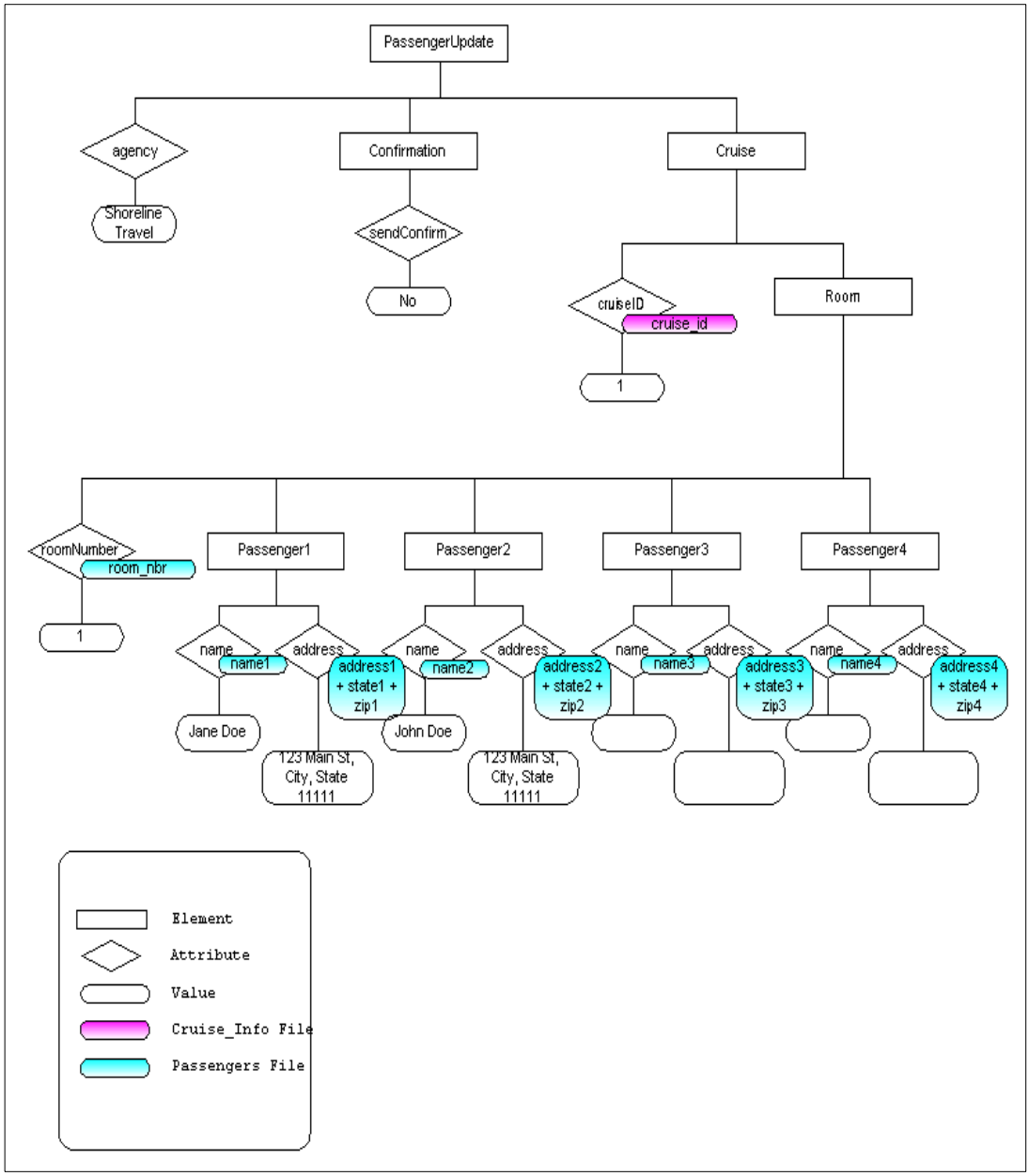


Figure 4-1 Passenger mapping scheme

4.7 Final DAD

This section shows the final DAD document after incorporating all the discoveries. This DAD file uses SQL mapping:

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "/javateam/shoreline/dtd/dad.dtd">
<DAD>
<dtdid>/JavaTeam/Shoreline/dtd/passengerUpdate.dtd</dtdid>
<validation>NO</validation>

<Xcollection>
```

```
<SQL_stmt>
   SELECT 'Shoreline Travel' AS AGENCY_NAME, 'No' AS SEND_CONFIRM, CRUISE_ID,
   SHORELINE.PASSENGERS.BOOKING_NBR as BOOK, ROOM_NBR, NAME_1, (ADDRESS_1 CONCAT ', '
   CONCAT CITY_1 CONCAT ', ' CONCAT STATE_1 CONCAT ' ' CONCAT         ZIP_1) AS
   COMPLETE_ADDRESS_1, NAME_2, (ADDRESS_2 CONCAT ', ' CONCAT CITY_2 CONCAT ', ' CONCAT
   STATE_2 CONCAT ' ' CONCAT         ZIP_2) AS COMPLETE_ADDRESS_2, NAME_3, (ADDRESS_3
   CONCAT ', ' CONCAT CITY_3 CONCAT ', ' CONCAT STATE_3 CONCAT ' ' CONCAT         ZIP_3) AS
   COMPLETE_ADDRESS_3, NAME_4, (ADDRESS_4 CONCAT ', ' CONCAT CITY_4 CONCAT ', ' CONCAT
   STATE_4 CONCAT ' ' CONCAT         ZIP_4) AS COMPLETE_ADDRESS_4 FROM
   SHORELINE.CRUISE_INFO, SHORELINE.PASSENGERS WHERE CRUISE_CO = 'Cruise Company' AND
   PROCESSED_INDICATOR = 'N' AND SHORELINE.CRUISE_INFO.BOOKING_NBR =
   SHORELINE.PASSENGERS.BOOKING_NBR ORDER BY AGENCY_NAME, CRUISE_ID, BOOK, ROOM_NBR
</SQL_stmt>

<prolog>?xml version="1.0" encoding="UTF-8"?</prolog>
<doctype>!DOCTYPE PassengerUpdate SYSTEM
"/JavaTeam/Shoreline/dtd/passengerUpdate.dtd"</doctype>
<root_node>
  <element_node name="PassengerUpdate">
    <attribute_node name="agency">
      <column name="AGENCY_NAME"/>
    </attribute_node>
  <element_node name="Confirmation">
    <attribute_node name="sendConfirm">
      <column name="SEND_CONFIRM"/>
    </attribute_node>
  </element_node>
  <element_node name="Cruise" multi_occurrence="YES">
    <attribute_node name="cruiseID">
      <column name="CRUISE_ID"/>
    </attribute_node>
    <element_node name="Room">
      <attribute_node name="roomNumber">
        <column name="ROOM_NBR"/>
      </attribute_node>
      <element_node name="Passenger1">
        <attribute_node name="name">
          <column name="NAME_1"/>
        </attribute_node>
        <attribute_node name="address">
          <column name="COMPLETE_ADDRESS_1"/>
        </attribute_node>
      </element_node>
      <element_node name="Passenger2">
        <attribute_node name="name">
          <column name="NAME_2"/>
        </attribute_node>
        <attribute_node name="address">
          <column name="COMPLETE_ADDRESS_2"/>
        </attribute_node>
      </element_node>
      <element_node name="Passenger3">
        <attribute_node name="name">
          <column name="NAME_3"/>
        </attribute_node>
        <attribute_node name="address">
          <column name="COMPLETE_ADDRESS_3"/>
        </attribute_node>
      </element_node>
```

```
      <element_node name="Passenger4">
        <attribute_node name="name">
          <column name="NAME_4"/>
        </attribute_node>
        <attribute_node name="address">
          <column name="COMPLETE_ADDRESS_4"/>
        </attribute_node>
      </element_node>
    </element_node><!-- end Room -->
  </element_node>       <!-- end Cruise -->
 </element_node><!-- end Passenger-->
</root_node>
</Xcollection>
</DAD>
```

4.8 XML enablement commands

This section lists the commands that must be run to compose the Passenger.xml document from the database tables:

1. The following SQL statement must be issued to store the passengerUpdate.dtd in the DTD repository table:

   ```
   INSERT INTO DB2XML.DTD_REF VALUES('/javateam/shoreline/dtd/passengerUpdate.dtd',
   DB2XML.XMLCLOBFROMFILE('/JavaTeam/Shoreline/dtd/passengerUpdate.dtd'), 0, 'author_name',
   'creator_name', NULL)
   ```

2. Since the dxxRetrieveXML() stored procedure is used to compose the document, enable the XML collection using the following command:

   ```
   CALL PGM(QDBXM/QZXMADM) PARM(ENABLE_COLLECTION db_name SHORELINE.XML_PASSENGER
   '/javateam/shoreline/dad/passengers.dad')
   ```

4.9 Java source

The following code snippet composes the passenger XML document from the database tables with the defined DAD. It then writes the XML document to the IFS file and proceeds to read it so the program can later send it in an e-mail message:

```
...
// Remove old records from file by deleting records from XML_PASSENGER
stmt.executeUpdate("DELETE FROM SHORELINE.XML_PASSENGER");

// Call the stored procedure
cs = con.prepareCall("CALL db2xml.dxxRetrieveXML(?, ?, ?, ?, ?, ?, ?, ?, ?, ?)");

// Register the input parameter: collection name
cs.setString(1, "SHORELINE.XML_PASSENGER");

// Register the input parameter: result table name
cs.setString(2, "SHORELINE.XML_PASSENGER");

// Register the input parameter: result column name
cs.setString(3, "xml_document");

// Register the input parameter: valid column name
cs.setString(4, "valid_document");

// Register the input parameter: override type
```

```
cs.setInt(5, 0);

// Register the input parameter: override
cs.setString(6, "NO_OVERRIDE");

// Register the input parameter: max # of rows
cs.setInt(7, 500);

// Register output parameter: actual # of rows
cs.registerOutParameter(8, Types.INTEGER);

// Register output parameter: return code
cs.registerOutParameter(9, Types.INTEGER);

// Register output parameter: error message text
cs.registerOutParameter(10, Types.VARCHAR);

// Run the stored procedure
cs.execute();
System.out.println("# of Rows: " + cs.getInt(8));
System.out.println("return code: " + cs.getInt(9));
System.out.println("message text: " + cs.getString(10));

if (!(cs.getInt(9) == 0))
throw new Exception("Error creating XML document -- error code: " + cs.getInt(9));

// Create the XML document -- /javateam/shoreline/passengers.xml
stmt.executeQuery("SELECT DB2XML.CONTENT(XML_DOCUMENT,
'/JAVATEAM/SHORELINE/passengers.xml') FROM SHORELINE.XML_PASSENGER");

// Get XML document into string. Use Toolbox API so that the data is converted to ASCII.
IFSTextFileInputStream in = null;
AS400 as400 = new AS400(sysName, dbUser, dbPassword);
in = new IFSTextFileInputStream(as400, "/javateam/shoreline/passengers.xml");
xmlString = in.read(in.available());
in.close();
...
```

5

Invoice

This chapter describes the XML Extender pieces of the invoice application that allows the travel agency to receive an invoice XML document from the cruise company. The invoice XML document contains all of the invoice and payment information for any cruises that have sailed. This invoice XML document is then decomposed into two temporary DB tables by the travel agency based on the statement ID. The information within these tables is used by another application to process the statement. The original invoice XML document is also stored in a database table using XML columns for archival purposes.

The DTD used for this XML document is based off of an existing industry standard DTD (VISA Invoice DTD Version 1.0) found at:

http://www.visa.com/ut/dnld/spec.ghtml

© Copyright IBM Corp. 2001

5.1 Database details

The XML document is decomposed into invoice_temp1 and invoice_temp2 DB2 tables. It is then stored in the accounts_payable DB2 table.

The invoice_temp1 table contains the overall statement information that is received from the cruise company. The layout of the invoice_temp1 table is listed in Table 5-1.

Table 5-1 Invoice_temp1

Key	Field name	Alias name	Data type	Length	VarLen Alloc	Digits, DecPos	Description
	STMT_ID	STATEMENT_ID	VARCHAR	30	10		Statement ID
	CRUISECOM	CRUISECO	VARCHAR	40	20		Cruise company name
	TOTALPRC	TOTALPRICE	DECIMAL			8,2	Total price of statement

The invoice_temp2 table contains the information for the individual rooms within the statement. The layout of the invoice_temp2 table is listed in Table 5-2.

Table 5-2 Invoice_temp2

Key	Field name	Alias name	Data type	Length	VarLen Alloc	Digits, DecPos	Description
	STATMENTID	STATEMENT_ID	VARCHAR	30	10		Statement ID
	PRODUCT	PRODID	VARCHAR	20	20		Product ID
	ORDERNBR	ORDER_NBR	VARCHAR	20	10		Order number
	ROOMPRICE	ROOM_PRICE	DECIMAL			8,2	Room price
	RMNBR	ROOMNBR	VARCHAR	10	6		Room number
	CRUISE	CRUISEID	VARCHAR	20	10		Cruise ID

The accounts_payable table contains the statement and payment response XML documents along with the paid status indicator. The layout of the accounts_payable table is listed in Table 5-3.

Table 5-3 Accounts_payable

Key	Field name	Alias name	Data type	Length	Allow Nulls	Description
	PDSTATUS	PAID_STATUS	CHAR	1		Paid Status ► O=unprocessed ► P=paid ► R=rejected ► S=send
	XMLSTMT	XML_STATEMENT	XMLCLOB			XML Statement Document
	XMLPAYRES	XML_PAYMENT_RESPONSE	XMLVARCHAR		Y	XML Payment Response Document

The invoice_data table is a side table used when the XML document is stored in the accounts_payable table. The layout of the invoice_data table is listed in Table 5-4.

Table 5-4 Invoice_data

Field name	Data type	Length	Allow null	Digits, DecPos	Description
DXXROOT_ID	CHAR	13			Root ID
STATEMENT_ID	VARCHAR	20	Y		Statement ID
INVOICE_DATE	DATE		Y		Invoice Date
TOTAL_PRICE	DECIMAL		Y	10, 2	Total Price

The invoice_company table is a side table used when the XML document is stored in the accounts_payable table. The layout of the invoice_company table is listed in Table 5-5.

Table 5-5 Invoice_company

Field name	Data type	Length	Allow null	Description
DXXROOT_ID	CHAR	13		Root ID
DXX_SEQNO	INT	9	Y	Sequence number
CRUISE_CO	VARCHAR	40	Y	Cruise company name

The invoice_company_type table is a side table used when the XML document is stored in the accounts_payable table. The layout of the invoice_company_type table is listed in Table 5-6.

Table 5-6 Invoice_company_type

Field name	Data type	Length	Allow null	Description
DXXROOT_ID	CHAR	13		Root ID
DXX_SEQNO	INT	9	Y	Sequence number
TYPE	VARCHAR	5	Y	Company type

5.2 Starting the XML document

This section shows an example of the original XML document that the travel agency attempts to decompose and store into the tables presented in the previous section:

```
<?xml version="1.0"?>
<!DOCTYPE Invoice SYSTEM "/tfc/invoice.1.0.dtd">
<Invoice sectorUsageVersion="1">
  <InvoiceHeader>
    <InvoiceType stdValue="380"/>
    <InvoiceStatus stdValue="9"/>
    <TaxTreatment stdValue="GIL"/>
    <InvoiceTreatment stdValue="E"/>
    <InvoiceNumber>s000000006</InvoiceNumber>
    <InvoiceDate>05/16/2001</InvoiceDate>
    <Currency stdValue="USD"/>
    <Party stdValue="SU">
      <PartyID>A4321</PartyID>
      <Name>
        <Name1>Cruise Lines</Name1>
```

```
      </Name>
    </Party>
    <Party stdValue="BY">
      <PartyID>A4325</PartyID>
      <Name>
        <Name1>Travel Agency</Name1>
      <Name>
    </Party>
  </InvoiceHeader>
  <InvoiceDetails>
    <BaseItemDetail>
      <LineItemNum>1</LineItemNum>
      <PartNumDetail>
        <PartNum>23</PartNum>
        <PartDesc>4-Day Caribbean Standard Inside</PartDesc>
      </PartNumDetail>
      <Quantity>
        <Qty>4</Qty>
        <UnitOfMeasure stdValue="EA"/>
      </Quantity>
    </BaseItemDetail>
    <UnitPrice>399</UnitPrice>
    <POLineNum>70698</POLineNum>
    <LineItemSubtotal>1398.0</LineItemSubtotal>
    <Date stdValue="STRT" stdName="VISA:DATE">05/16/2001T00:00:00</Date>
    <Date stdValue="END" stdName="VISA:DATE">05/20/2001T00:00:00</Date>
    <Ref stdValue="FLNO" stdName="VISA:REF">29</Ref>
    <Ref stdValue="SRVC" stdName="VISA:REF">Standard Inside</Ref>
    <Ref stdValue="RMNO" stdName="VISA:REF">1</Ref>
    <Ref stdValue="FBC" stdName="VISA:REF">Single</Ref>
  </InvoiceDetails>
  <InvoiceDetails>
    ...
  </InvoiceDetails>
  <InvoiceSummary>
    <TaxSummary>
      <Tax>
        <TaxFunction stdValue="7"/>
        <TaxType stdValue="GST"/>
        <TaxCategory stdValue="S"/>
        <TaxPercent>7.5</TaxPercent>
        <TaxableAmount>20071.00</TaxAmount>
        <TaxAmount>1505.33</TaxAmount>
      </Tax>
    </TaxSummary>
    <InvoiceTotals>
      <NetValue>20071.00</NetValue>
      <TaxValue>1505.33</TaxValue>
      <GrossValue>21576.33</GrossValue>
    </InvoiceTotals>
  </InvoiceSummary>
</Invoice>
```

5.3 Discoveries

This section lists discoveries we encountered while creating the DAD file used for the invoice XML document:

1. The DAD must have tags that are unique. Within the XML document, any tag with the same name must map to the same column in the DAD. Any tag changes in the DAD also require tag changes in the DTD. For example, the second <name> tag must be changed to <name4> so that it can map to a separate column in the DB2 table:

```
...
<Party stdValue="SU">
  <PartyID>A4321</PartyID>
  <Name>
    <Name1>Cruise Company</Name1>
  </Name>
</Party>
<Party2 stdValue="BY">
  <PartyID2>A4325</PartyID2>
  <Name4>
    <Name5>Travel Agency</Name5>
  </Name4>
</Party2>
...
```

2. Side tables are intended to help you search for relevant XML documents. They do not provide for construction of hierarchies. When we started to look at how we wanted to use the information contained within this XML document, we were going to save the document using XML columns and set up side tables that would hold the data that we wanted processed by another piece of our application. To retain the hierarchy structure, we had to decompose the XML document into database tables. It was then determined that we would use the invoice_temp1 and invoice_temp2 tables for the decomposition of the XML document.

3. When using an XML collection, you must understand how the XML data is represented in the relational database. Since there is multiple room information within an XML statement, the XML document must be decomposed into two separate tables; one that contains the overall information for the statement and one that contains all of the room information. We created invoice_temp1 (overall statement information) and invoice_temp2 (room information) to store the information.

4. When an element has multiple child element types, multi_occurrence must be specified in the DAD file. The <InvoiceHeader> definition needs multi_occurrence specified because of the child types <InvoiceNumber>, <Party>, <Name>, and <Name1> that map to the same table. The <Invoice> definition also needs multi_occurrence for the same reason.

```
...
<root_node>
  <element_node name="Invoice" multi_occurrence="YES">
    <RDB_node>
      <table name="shoreline.invoicetemp" key="statement_id"/>
    </RDB_node>
  <element_node name="InvoiceHeader" multi_occurrence="YES">
...
```

5. Sets of repeating elements must be enclosed within another element (called a wrapper element). Since <InvoiceDetails> is a repeating element, we added <InvoiceDetailsList> to the DTD and to the XML document:

```
...
<InvoiceDetailsList>
  <InvoiceDetails> ....
  </InvoiceDetails>
  <InvoiceDetails> ....
  </InvoiceDetails>
</InvoiceDetailsList>
```

In the DAD:

```
...
<element_node name="InvoiceDetailsList">
    <element_node name="InvoiceDetails"  multi_occurrence="YES">
      ....
        <element_node name="Ref3">
          <text_node>
            <RDB_node>
              <table name="shoreline.invoice_temp2"/>
              <column name="roomnbr" type="char(5)"/>
            </RDB_node>
          </text_node>
        </element_node>   <!-- Ref3 -->
    </element_node>     <!-- InvoiceDetails -->
</element_node>     <!-- InvoiceDetailsList -->
...
```

6. For a given table, if there are no attribute_node's mappings for the table, the first element_node that has a mapping to the table must be directly below the element_node that specifies multi_occurrence. When first creating the DAD, the data is decomposed as shown in Table 5-7.

Table 5-7 Discovery 6 starting data

STATEMENT_ID	PRODID	ORDER_NBR	ROOM_PRICE	ROOMNBR	CRUISEID
S12345	545,454	-	-	-	-
S12345	545,456	-	-	-	-
S12345	-	123,123,123	-	-	-
S12345	-	123,123,124	-	-	-
S12345	-	-	-		1,234,366
S12345	-	-	-		1,234,366
S12345	-	-	-	101	-
S12345	-	-	-	106	-

Table 5-8 shows how we wanted the data to look.

Table 5-8 Discovery 6 final data

STATEMENT_ID	PRODID	ORDER_NBR	ROOM_PRICE	ROOMNBR	CRUISE_ID
S12345	545,454	123,123,123	-	101	1,234,366
S12345	545,456	123,123,124	-	106	1,234,366

The problem is that <PartNum> is the first element_node that maps to invoice_temp2. However, it is nested too deeply within <InvoiceDetails>, which has a multi_occurrence specification to indicate that multiple child elements map to the same table. To solve the problem, you can change the DAD using a couple of methods:

- Remove the two levels of nesting of <PartNum> within <InvoiceDetails> (that is, remove <BaseItemDetail> and <PartNumDetail>).
- Move <POLineNum> ahead of the block of <BaseItemDetail> and its child elements. <POLineNum> is mapped to a column of invoice_temp2 and is a direct child element of <InvoiceDetails>.

```
...
<InvoiceDetails>
<POLineNum>70698</POLineNum>
<BaseItemDetail>
...
```

We implemented the second option by updating the DTD and the XML document.

7. The XML Extender allows you to store DTDs in the dtd_ref table in library DB2XML. The DTDs are used for validating the structure of XML documents. If changes are made to the DTD after a row has been inserted in the dtd_ref table for that particular DTD, the row should be deleted and re-inserted into the dtd_ref table to pick up the new changes in the DTD file.
8. When enabling a column, you must specify any side tables. We stored the XML document in the accounts_payable table and did not require the use of side tables. We just wanted to save the document and also validate it. You cannot enable a column to simply have the validation done on an insert or update.

   When we enabled the column, we ended up specifying three side tables for the data (invoice_data, invoice_company and invoice_company_type). We selected data for the side tables that we thought may be useful in future searches.

9. When enabling a column, the extender creates the side tables using the name specified in the DAD file. If the side table name in the DAD is not fully qualified, the default DB2 schema naming rules apply. In this case, the job's user ID becomes the default schema. You can specify schema names in the DAD file. This creates the side tables where you want them. The extender does not put the side tables in the same schema as the base table by default. Use the *SQL naming convention to specify the table name (such as 'mydb.app1', not 'mydb/app1').
10. When enabling a column, use the following suggestions:
    - Fully-qualify the side table names in the DAD file.
    - When creating collections, use STRSQL and a statement like ‘CREATE COLLECTION WXYZ’. This provides full journaling support for subsequent CREATE TABLE statements and allows the side tables to be journaled.
    - Make sure that, when you create tables in the collection, the tables are journaled.
11. A null string is not a valid XML document. If you have a column that is enabled using a DAD file that has validation set to YES, and you try to insert a null value into the column, you receive an error indicating that the XML document is invalid. We made a change to the xml_statement column in the accounts_payable field to not allow null values.
12. If you have a table that has been enabled (for side tables) and you drop the table without disabling it, you must drop the side tables and remove the record from the xml_usage table.
13. You must use unique column names for the side tables when creating a default view with the -v parameter of the enable_column option of the administration command. This means that, for a given XML column, you should use different column names in each of the side tables to prevent collisions when the side table column names are combined in the CREATE VIEW statement. If you plan to create your own view for the side tables, instead of using the -v parameter, this restriction does not apply.

5.4 Final XML document

This section shows an example of what the final XML document looks like after incorporating all the discoveries:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Invoice SYSTEM "/JavaTeam/Shoreline/dtd/invoice.1.0.dtd">
<Invoice sectorUsageVersion="1">
  <InvoiceHeader>
    <InvoiceType stdValue="380"/>
    <InvoiceStatus stdValue="9"/>
    <TaxTreatment stdValue="GIL"/>
    <InvoiceTreatment stdValue="E"/>
    <InvoiceNumber>s000000006</InvoiceNumber>
    <InvoiceDate>05/16/2001</InvoiceDate>
    <Currency stdValue="USD"/>
    <Party stdValue="SU">
      <PartyID>A4321</PartyID>
      <Name>
        <Name1>Cruise Company</Name1>
      </Name>
    </Party>
    <Party2 stdValue="BY">
      <PartyID2>A4325</PartyID2>
      <Name4>
        <Name5>Travel Agency</Name5>
      </Name4>
    </Party2>
  </InvoiceHeader>
  <InvoiceDetailsList>
    <InvoiceDetails>
      <POLineNum>70698</POLineNum>
      <BaseItemDetail>
        <LineItemNum>1</LineItemNum>
        <PartNumDetail>
          <PartNum>23</PartNum>
          <PartDesc>4-Day Caribbean Standard Inside</PartDesc>
        </PartNumDetail>
        <Quantity>
          <Qty>4</Qty>
          <UnitOfMeasure stdValue="EA"/>
        </Quantity>
      </BaseItemDetail>
      <UnitPrice>399</UnitPrice>
      <LineItemSubtotal>1398.0</LineItemSubtotal>
      <Date stdValue="STRT" stdName="VISA:DATE">05/16/2001T00:00:00</Date>
      <Date stdValue="END" stdName="VISA:DATE">05/20/2001T00:00:00</Date>
      <Ref1 stdValue="FLNO" stdName="VISA:REF">29</Ref1>
      <Ref2 stdValue2="SRVC" stdName2="VISA:REF">Standard Inside</Ref2>
      <Ref3 stdValue3="RMNO" stdName3="VISA:REF">1</Ref3>
      <Ref4 stdValue4="FBC" stdName4="VISA:REF">Single</Ref4>
    </InvoiceDetails>
    <InvoiceDetails>
      <POLineNum>70698</POLineNum>
      <BaseItemDetail>
        <LineItemNum>2</LineItemNum>
        ...
      </BaseItemDetail>
      ...
    </InvoiceDetails>
```

```
    <InvoiceDetails>
      ...
    </InvoiceDetails>
  </InvoiceDetailsList>
  <InvoiceSummary>
    <TaxSummary>
      <Tax>
        <TaxFunction stdValue="7"/>
        <TaxType stdValue="GST"/>
        <TaxCategory stdValue="S"/>
        <TaxPercent>7.5</TaxPercent>
        <TaxableAmount>20071.00</TaxableAmount>
        <TaxAmount>1505.33</TaxAmount>
      </Tax>
    </TaxSummary>
    <InvoiceTotals>
      <NetValue>20071.00</NetValue>
      <TaxValue>1505.33</TaxValue>
      <GrossValue>21576.33</GrossValue>
    </InvoiceTotals>
  </InvoiceSummary>
</Invoice>
```

5.5 DTD mapping scheme

Figure 5-1 on page 50 illustrates how the structure of the XML document relates to the DB2 tables that are used for the decomposition. This was helpful in the creation of the DAD file because it shows how the two structures compare.

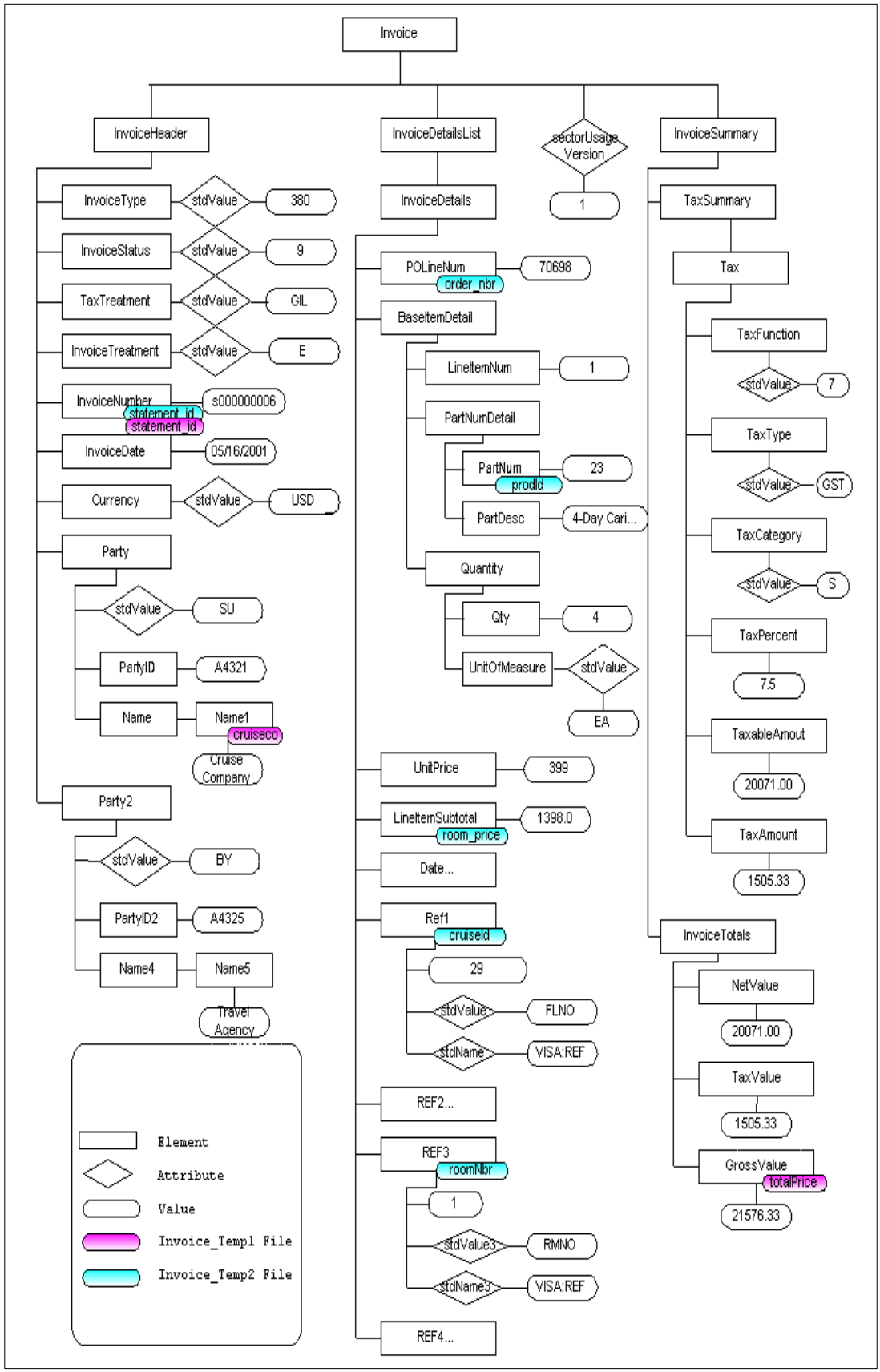


Figure 5-1 Invoice mapping scheme

5.6 Final DADs

This section shows the final DAD document used in the decomposition of the XML document after incorporating all the discoveries. This DAD file for an XML collection uses RDB_node mapping:

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "/javateam/shoreline/dtd/dad.dtd">
<DAD>
<dtdid>/javateam/shoreline/dtd/invoice.1.0.dtd</dtdid>
<validation>YES</validation>
<Xcollection>
<prolog>?xml version="1.0"?</prolog>
<doctype>!DOCTYPE Invoice SYSTEM "/javateam/shoreline/dtd/invoice.1.0.dtd"</doctype>
<root_node>
  <element_node name="Invoice" multi_occurrence="YES">
    <RDB_node>
      <table name="shoreline.invoice_temp1" key="statement_id"/>
      <table name="shoreline.invoice_temp2" key="statement_id"/>
      <condition>
       shoreline.invoice_temp1.statement_id = shoreline.invoice_temp2.statement_id
      </condition>
    </RDB_node>
    <element_node name="InvoiceHeader" multi_occurrence="YES">
      <element_node name="InvoiceNumber">
        <text_node>
          <RDB_node>
            <table name="shoreline.invoice_temp1"/>
            <column name="statement_id" type="varchar(30)"/>
          </RDB_node>
        </text_node>
      </element_node>       <!-- InvoiceNumber -->
      <element_node name="Party">
        <element_node name="Name">
          <element_node name="Name1">
            <text_node>
              <RDB_node>
                <table name="shoreline.invoice_temp1"/>
                <column name="cruiseco" type="varchar(40)"/>
              </RDB_node>
            </text_node>
          </element_node>   <!-- Name1 -->
        </element_node>     <!-- Name -->
      </element_node>       <!-- Party -->
    </element_node>       <!--InvoiceHeader -->
    <element_node name="InvoiceDetailsList">
    <element_node name="InvoiceDetails" multi_occurrence="YES">
      <element_node name="POLineNum">
        <text_node>
          <RDB_node>
            <table name="shoreline.invoice_temp2"/>
            <column name="order_nbr" type="varchar(20)"/>
          </RDB_node>
        </text_node>
      </element_node>   <!-- POLineNum -->
      <element_node name="BaseItemDetail">
        <element_node name="PartNumDetail">
          <element_node name="PartNum">
            <text_node>
              <RDB_node>
```

```
              <table name="shoreline.invoice_temp2"/>
              <column name="prodid" type="varchar(20)"/>
            </RDB_node>
          </text_node>
        </element_node>   <!-- PartNum -->
      </element_node>     <!-- PartNumDetail -->
    </element_node>       <!-- BaseItemDetail -->
    <element_node name="LineItemSubtotal">
      <text_node>
        <RDB_node>
          <table name="shoreline.invoice_temp2"/>
          <column name="room_price" type="decimal(8,2)"/>
        </RDB_node>
      </text_node>
    </element_node>   <!-- LineItemSubtotal -->
    <element_node name="Ref1">
      <text_node>
        <RDB_node>
          <table name="shoreline.invoice_temp2"/>
          <column name="cruiseid" type="varchar(20)"/>
        </RDB_node>
      </text_node>
    </element_node>   <!-- Ref1 -->
    <element_node name="Ref3">
      <text_node>
        <RDB_node>
          <table name="shoreline.invoice_temp2"/>
          <column name="roomnbr" type="varchar(10)"/>
        </RDB_node>
      </text_node>
   </element_node>   <!-- Ref3 -->
   </element_node>     <!-- InvoiceDetails -->
   </element_node>  <!-- InvoiceDetailsList -->
       <element_node name="InvoiceSummary">
    <element_node name="InvoiceTotals">
      <element_node name="GrossValue">
        <text_node>
          <RDB_node>
            <table name="shoreline.invoice_temp1"/>
            <column name="totalprice" type="Decimal(8,2)"/>
          </RDB_node>
        </text_node>
      </element_node> <!-- GrossValue -->
    </element_node>   <!-- InvoiceTotals -->
  </element_node>     <!-- InvoiceSummary -->
</element_node>     <!-- Invoice -->
</root_node>
</Xcollection>
</DAD>
```

Here is the final DAD document (InvoiceSideTable.dad) used to perform the validation of the XML document and to create the side tables. This DAD file for an XML column uses SQL mapping:

```
<?xml version="1.0"?>
<!DOCTYPE Invoice SYSTEM "/javateam/shoreline/dad/dad.dtd">
<DAD>
  <dtdid>/javateam/shoreline/dtd/invoice.1.0.dtd</dtdid>
  <validation>YES</validation>
  <Xcolumn>
    <table name="shoreline.invoice_data">
```

```
      <column name="statement_id"
              type="varchar(20)"
              path="/Invoice/InvoiceHeader/InvoiceNumber"
              multi_occurrence="NO"/>
      <column name="invoice_date"
              type="DATE"
              path="/Invoice/InvoiceHeader/InvoiceDate"
              multi_occurrence="NO"/>
      <column name="total_price"
              type="decimal(10,2)"
              path="/Invoice/InvoiceSummary/InvoiceTotals/GrossValue"
              multi_occurrence="NO"/>
    </table>
    <table name="shoreline.invoice_company">
      <column name="cruise_co"
              type="varchar(40)"
              path="/Invoice/InvoiceHeader/Party/Name/Name1"
              multi_occurrence="YES"/>
    </table>
    <table name="shoreline.invoice_company_type">
      <column name="type"
              type="varchar(5)"
              path="/Invoice/InvoiceHeader/Party/@stdValue"
              multi_occurrence="YES"/>
    </table>
  </Xcolumn>
</DAD>
```

5.7 XML enablement commands

This section lists the commands that must be run to decompose the Invoice.xml document into the database tables and to store the original XML document in the accounts_payable table:

1. When using the DTD to validate XML data in the XML collection (VALIDATION set to YES in the DAD file), the following SQL statement must be issued to store the Invoice.dtd in the DTD repository table:

   ```
   INSERT INTO DB2XML.DTD_REF VALUES('/javateam/shoreline/dtd/invoice.1.0.dtd',
   DB2XML.XMLCLOBFROMFILE('/javateam/shoreline/dtd/invoice.1.0.dtd'), 0, 'author_name',
   'creator_name', NULL)
   ```

2. Enable the xml_statement column in the accounts_payable table using the following command:

   ```
   CALL PGM(QDBXM/QZXMADM) PARM(ENABLE_COLUMN db_name SHORELINE.ACCOUNTS_PAYABLE
   XML_STATEMENT '/javateam/shoreline/dad/invoicesidetable.dad')
   ```

5.8 Java source

The following code snippet decomposes the invoice XML document into the database tables with the defined DAD:

```
...
// read dad file
byte buf[] = new byte[15000];
try {
    FileInputStream in = new FileInputStream("/javateam/shoreline/dad/Invoice.dad");
    in.read(buf, 0, 15000);
```

```
}
catch (Exception e) {
    System.out.println("Error: " + e.toString());
}
dad = new String(buf, 0);

// Call the stored procedure
cs = con.prepareCall("CALL db2xml.dxxShredXML(?, ?, ?, ?)");

// Register the input parameter: dad file
cs.setObject(1, dad);

// Register the input parameter: xml file
cs.setObject(2, xmlDoc);

// Register the output parameter: return code
cs.registerOutParameter(3, Types.BIGINT);

// Register the output parameter: return message
cs.registerOutParameter(4, Types.CHAR);

// Run the stored procedure
cs.execute();
System.out.println("return code: " + cs.getInt(3));
System.out.println("message text: " + cs.getString(4));
...
```

The following code snippet inserts the invoice XML document into a column in the accounts_payable database table:

```
...
try {

   // creating Byte array from String
   byte buf[] = new byte[15000];
   buf = invoiceString.getBytes();

   // creating output stream

   // Work with /Javateam/Shoreline/xml/invoice.xml on the system.
   AS400 as400 = new AS400(SysName, DbUser, DbPassword);
   IFSFileOutputStream file = new IFSFileOutputStream(as400,
   "/javateam/shoreline/xml/Invoice.xml", 437);

   // Write a String to the file (don't convert characters).
   file.write(buf);
   // Close the file
   file.close();

   // inserting into DB
   con = ds.getConnection(DbUser, DbPassword);
   stmt = con.createStatement();

   stmt.executeUpdate("Insert into shoreline.accounts_payable values('0',
   db2xml.xmlclobfromfile('/javateam/shoreline/xml/Invoice.xml'), null, null)");
   ...
```

6

Payment

This chapter describes the XML Extender pieces of the payment request application that allows the travel agency to send a payment request XML document to the bank authorizing payment for a specific statement. The payment request XML document contains the payment information to transfer funds from the travel agency to the cruise company. The payment request XML document is composed from several DB2 database tables. The travel agency uses an XML collection because they have data in their existing relational tables and want to compose XML documents based on a certain DTD.

The bank receives the payment request XML document and decomposes it into a single DB2 database table. When the processing is complete, a payment response XML document is sent back to the travel agency. The bank decides to use an XML collection for storage of the payment response XML document.

The travel agency receives the payment response XML document and stores it in an XML column so they know the payment has been made.

The DTD used for this XML document is based off of an existing industry standard DTD (Bank Internet Payment System (BIPS)), which is available at:

http://www.fstc.org/projects/bips/

© Copyright IBM Corp. 2001

6.1 Composing paymentRequest

This section describes the composition of the paymentRequest XML document.

6.1.1 Database details

The XML document is composed from four DB2 tables. The DB2 tables are:

- payment_info1
- invoice_temp1
- payment_info
- accounts_payable

The xml_payment table is used to store the XML document.

The payment_info1 table contains the email address and the sequence number for the payment request document. The layout of the payment_info1 table is listed in Table 6-1.

Table 6-1 Payment_info1

Key	Field name	Alias name	Data types	Length	VarLen Alloc	Description
	EMAIL		VARCHAR	30	15	Email Address
	SEQUENCE		INT			Sequence Number

The invoice_temp1 table contains the overall statement information that is received from the cruise company. The layout of the invoice_temp1 table is listed in Table 6-2.

Table 6-2 Invoice_temp1

Key	Field name	Alias name	Data types	Length	VarLen Alloc	Digits, DecPos	Description
	STMT_ID	STATEMENT_ID	VARCHAR	30	10		Statement ID
	CRUISECOM	CRUISECO	VARCHAR	40	20		Cruise company name
	TOTALPRC	TOTALPRICE	DECIMAL			8,2	Total price of statement

The payment_info table contains the overall payment information that is sent to the bank. The layout of the payment_info table is listed in Table 6-3.

Table 6-3 Payment_info

Key	Field name	Alias name	Data types	Length	VarLen Alloc	Description
	BANKNAME		VARCHAR	20	10	Bank name
	TRAVELCO		VARCHAR	30	20	Travel company name
	PAYOR_TYPE		CHAR	3		Payor type
	PAYOR_CODE		VARCHAR	20	10	Payor code
	PAYOR_NAME		VARCHAR	20	10	Payor name
	P_ACT_TYPE		CHAR	9		Payor account type
	PAYOR00001	PAYOR_ACCOUNT	CHAR	6		Payor account

Key	Field name	Alias name	Data types	Length	VarLen Alloc	Description
	CRUISECO		VARCHAR	30	20	Cruise company name
	PAYTO_TYPE		CHAR	3		Pay to type
	PAYTO_CODE		VARCHAR	20	10	Pay to code
	PAYTO_NAME		VARCHAR	20	10	Pay to name
	C_ACT_TYPE		CHAR	9		Cruise account type
	PAYTO00001	PAYTO_ACCOUNT	CHAR	6		Pay to account
	ENCODING		CHAR	3		Encoding
	PERSON		VARCHAR	20	10	Person
	CERT		VARCHAR	10	5	Certificate

The accounts_payable table contains the statement and payment response XML documents along with the paid status indicator. The layout of the accounts_payable table is listed in Table 6-4.

Table 6-4 Accounts_payable

Key	Field name	Alias name	Data type	Length	Allow Nulls	Description
	PDSTATUS	PAID_STATUS	CHAR	1		Paid Status ► O=unprocessed ► P=paid ► R=rejected ► S=send
	XMLSTMT	XML_STATEMENT	XMLCLOB			XML Statement Document
	XMLPAYRES	XML_PAYMENT_RESPONSE	XMLVARCHAR		Y	XML Payment Response Document

The xml_payment table is used to store the XML document content. This table is used when composing the payment request XML document. The layout of the xml_payment table is listed in Table 6-5.

Table 6-5 XML_payment

Key	Field Name	Alias Name	Data Type	Description
	VALID00001	VALID_DOCUMENT	INT	Valid document indicator
	XML_D00001	XML_DOCUMENT	XMLCLOB	Payment request XML document

6.1.2 Starting XML document

This section shows an example of the original XML document that the travel agency attempts to compose from the tables presented in the previous section:

```
<?xml version="1.0"?>
  <!DOCTYPE bips SYSTEM "/tfc/bips.dtd">
  <bips>
```

```
<message-id sender-id="tfc@shoreline.com" date="2001-02-15" sequence="1"/>
<payment-request>
  <payment-request-id>S000012350</payment-request-id>
  <execution-date>
    <month>1</month>
    <day>22</day>
    <year>2001</year>
  </execution-date>
  <payment-network>
    <other network-name="Bank"></other>
  </payment-network>
  <amount>10456.50</amount>
  <payor>
    <entity>
      <name>Travel Agency</name>
    </entity>
    <bank-customer-info>
      <bank>
        <bank-code type="trn">123456789</bank-code>
        <entity>
          <name>Bank</name>
        </entity>
      </bank>
      <account type="corporate">432432</account>
    </bank-customer-info>
  </payor>
  <payto>
    <entity>
      <name>Cruise Company</name>
    </entity>
    <bank-customer-info>
      <bank>
        <bank-code type="trn">123456789</bank-code>
        <entity>
          <name>Bank</name>
        </entity>
      </bank>
    <account type="corporate">909090</account>
  </bank-customer-info>
 </payto>
</payment-request>
</bips>
```

6.1.3 Discoveries

This section lists some discoveries we encountered while creating the DAD file used for the paymentRequest XML document:

1. For enable_collection and decomposition, column type must be specified in the DAD. For example:

   ```
   ...
   <column name="email"  type="varchar(20)"/>
   ...
   ```

   The enable_collection command uses the column type specifications to create the tables in the collection if the tables do not exist. If the tables do exist, the type specified in the DAD must match the actual column type.

2. Insert bips1.dtd into the dtd_ref table in library DB2XML before enabling the collection. The value inserted into the dtdid column should match the contents of the <dtdid> element in the DAD.
3. In the DAD file, <current month>, <current day>, and <current year> must repeat within <payment-request> along with the other columns of invoice_temp1 to allow multiple <payment-request> elements. In the column list of the SQL statement, the following statements should follow the object id column of invoice_temp1, statement_id.

   ```
   strip(char(month(current timestamp)), both) as currentmonth,
   strip(char(day(current timestamp)), both) as currentday,
   strip(char(year(current timestamp)), both) as currentyear,
   ```

   After moving the statement_id, the select statement resembles the following example:

   ```
   SELECT email, strip(char(sequence),both) as sequence,(current date) as todaysdate,
   STATEMENT_ID, strip(char(month(current timestamp)), both) as currentmonth,
   strip(char(day(current timestamp)), both) as currentday,
   strip(char(year(current timestamp)), both) as currentyear,
   strip(char(totalprice), both) as totalprice, bankname, travelco,
   payor_type, PAYOR_CODE, PAYOR_NAME, P_ACT_TYPE, PAYOR_ACCOUNT,
   shoreline.payment_info.cruiseco as cruise_company,
   PAYTO_TYPE, PAYTO_CODE,PAYTO_NAME, C_ACT_TYPE,PAYTO_ACCOUNT, ENCODING, PERSON, CERT,
   shoreline.invoice_temp1.cruiseco
   FROM shoreline.payment_info1, shoreline.invoice_temp1, shoreline.payment_info
   WHERE shoreline.invoice_temp1.cruiseco = shoreline.payment_info.cruiseco
   ORDER BY EMAIL, statement_id, bankname
   ```

4. A paid_status indicator of ‘S’ (send) was added to the accounts_payable table to allow the travel agency to determine which statement ID to send to the bank. This allows the travel agency to be able to compose a paymentRequest.xml document for only those statements that the travel agency approves.

6.1.4 Final DTD

After incorporating all of the discoveries listed in the previous section, as well as the discoveries listed in 6.2, “Decomposing paymentRequest” on page 67, the final DTD document is shown here:

```
 <!ELEMENT bips    ( message-id, payment-request+, signature-info+ ) >
     <!ATTLIST bips
             version                       CDATA "1.0" >
     <!ELEMENT message-id                    EMPTY >
     <!ATTLIST message-id
               sender-id                     CDATA #REQUIRED
               sequence                      CDATA #REQUIRED
               date                          CDATA #REQUIRED >
     <!ELEMENT payment-request   (execution-date, payment-request-id, amount,
payment-network?, payor+, payto+) >
     <!ELEMENT payment-request-id            (#PCDATA) >
     <!ELEMENT payor       ( entity,  bank-customer-info) >
     <!ELEMENT payto           (entity, bank-customer-info ) >
     <!ELEMENT bank-customer-info            (bank, account ) >
     <!ELEMENT bank                          ( bank-code,  entity?  ) >
     <!ELEMENT bank-code                     (#PCDATA) >
     <!ATTLIST bank-code
               type                          ( trn  ) #REQUIRED >
     <!ELEMENT account  (id*, key-id*) >
     <!ATTLIST account
               type                          CDATA #IMPLIED >
     <!ELEMENT id (#PCDATA)>
     <!ELEMENT key-id  (#PCDATA)>
```

```
<!ELEMENT entity          ( name*, contact* ) >
<!ELEMENT name                          (#PCDATA) >
<!ELEMENT contact                       (#PCDATA) >
<!ELEMENT execution-date                ( month, day,  year ) >
<!ELEMENT amount                        (#PCDATA) >
<!ELEMENT payment-network               (other ) >
<!ELEMENT other                         EMPTY >
<!ATTLIST other
          network-name                  CDATA #REQUIRED >
<!ELEMENT month                         (#PCDATA) >
<!ELEMENT day                           (#PCDATA) >
<!ELEMENT year                          (#PCDATA) >
<!ELEMENT signature-info                ( signature, certificate ) >
<!ELEMENT signature    (#PCDATA) >
<!ATTLIST signature
          algorithm                     ( RSA ) #REQUIRED >
<!ELEMENT certificate                  (#PCDATA) >
```

6.1.5 Final XML document

This section shows an example of what the final XML document looks like after incorporating all of the discoveries:

```
<?xml version="1.0"?>
  <!DOCTYPE bips SYSTEM "/javateam/shoreline/dtd/bips1.dtd">
  <bips>
    <message-id sender-id="tfc@shoreline.com" sequence="1" date="2001-02-15"/>
    <payment-request>
      <execution-date>
        <month>1</month>
        <day>22</day>
        <year>2001</year>
      </execution-date>
      <payment-request-id>S000012350</payment-request-id>
      <amount>10456.50</amount>
      <payment-network>
        <other network-name="Bank"></other>
      </payment-network>
      <payor>
        <entity>
          <name>Travel Agency</name>
        </entity>
        <bank-customer-info>
          <bank>
            <bank-code type="trn">123456789</bank-code>
            <entity>
              <name>Bank</name>
            </entity>
          </bank>
          <account type="corporate">
            <id>432432</id>
          </account>
        </bank-customer-info>
      </payor>
      <payto>
        <entity>
          <contact>Cruise Company</contact>
        </entity>
        <bank-customer-info>
          <bank>
```

```
          <bank-code type="trn">123456789</bank-code>
          <entity>
            <name>Bank</name>
          </entity>
        </bank>
        <account type="corporate">
          <key-id>909090</key-id>
        </account>
      </bank-customer-info>
    </payto>
  </payment-request>
  <signature-info>
    <signature algorithm="RSA">"joe"</signature>
    <certificate>"Cert"</certificate>
  </signature-info>
</bips>
```

6.1.6 DTD mapping scheme

Figure 6-1 on page 62 illustrates how the structure of the XML document relates to the DB2 tables that are used for the composition. This was helpful in the creation of the DAD file because it shows how the two structures compare.

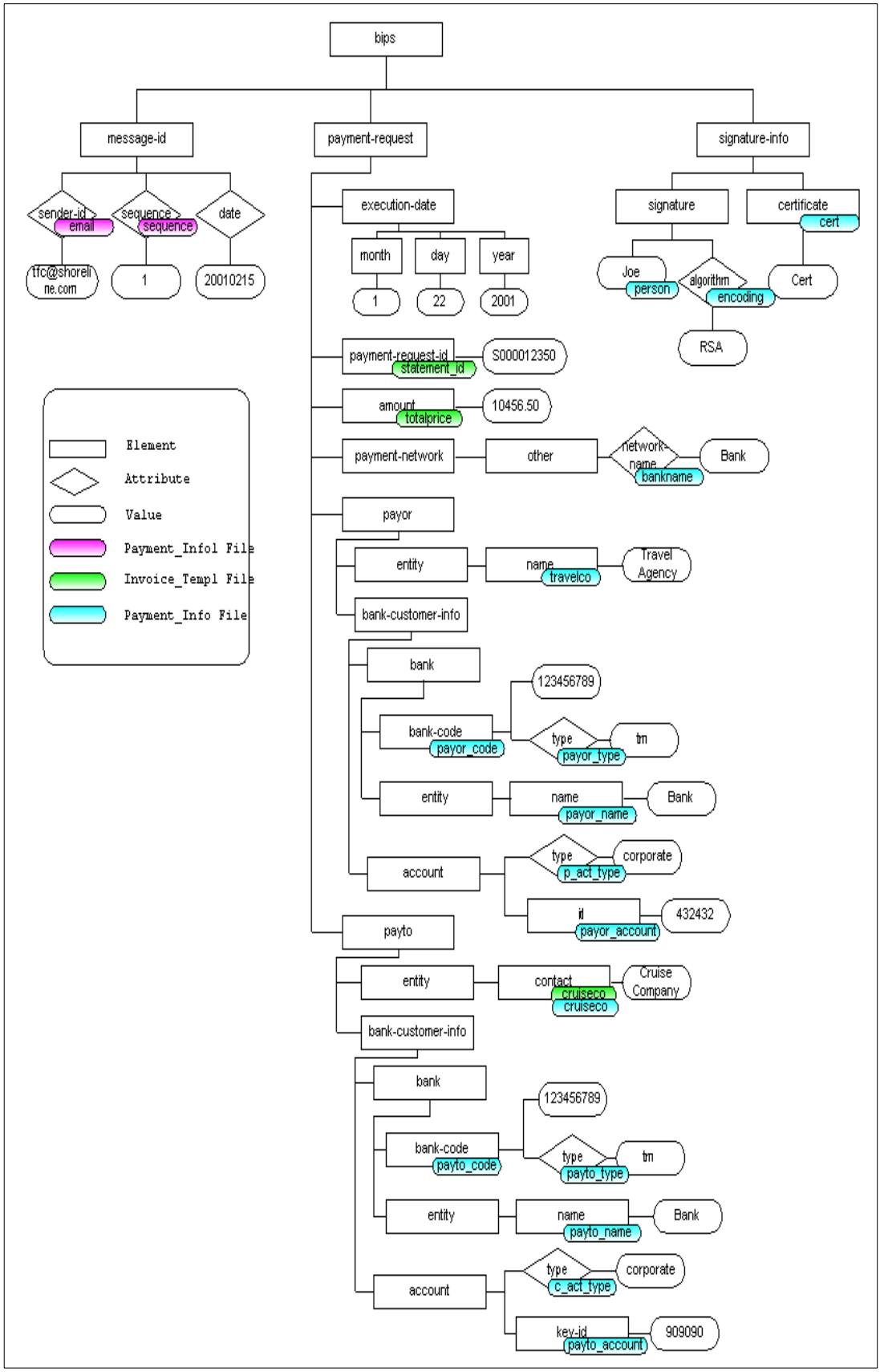


Figure 6-1 Composing passenger mapping scheme

6.1.7 Final DAD

This section shows the final DAD document used to compose the XML document after incorporating all of the discoveries. This DAD file for an XML collection uses SQL mapping:

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "/javateam/shoreline/dtd/dad.dtd">

<DAD>
<dtdid>/javateam/shoreline/dtd/bips1.dtd</dtdid>
<validation>YES</validation>

<Xcollection>

<SQL_stmt>
SELECT email, strip(char(sequence),both) as sequence,
(current date) as todaysdate, STATEMENT_ID,
strip(char(month(current timestamp)), both) as currentmonth,
strip(char(day(current timestamp)), both) as currentday,
strip(char(year(current timestamp)), both) as currentyear,
strip(char(totalprice), both) as totalprice, bankname, travelco,
payor_type, PAYOR_CODE, PAYOR_NAME, P_ACT_TYPE, PAYOR_ACCOUNT,
shoreline.payment_info.cruiseco as cruise_company,
PAYTO_TYPE, PAYTO_CODE,
PAYTO_NAME, C_ACT_TYPE,PAYTO_ACCOUNT, ENCODING, PERSON, CERT,
shoreline.invoice_temp1.cruiseco, shoreline.accounts_payable.paid_status
FROM shoreline.payment_info1, shoreline.invoice_temp1, shoreline.payment_info,
shoreline.accounts_payable
WHERE shoreline.invoice_temp1.cruiseco = shoreline.payment_info.cruiseco
and shoreline.invoice_temp1.statement_id = db2xml.extractvarchar(xml_statement,
'/Invoice/InvoiceHeader/InvoiceNumber') and shoreline.accounts_payable.paid_status = 'S'
ORDER BY EMAIL, statement_id, bankname, paid_status
</SQL_stmt>

<prolog>?xml version="1.0"?</prolog>
<doctype>!DOCTYPE bips SYSTEM "/javateam/shoreline/dtd/bips1.dtd"</doctype>
<root_node>
  <element_node name="bips">

  <element_node name="message-id">
    <attribute_node name="sender-id">
      <column name="email" type="varchar(30)"/>
    </attribute_node>
    <attribute_node name="sequence">
      <column name="sequence" type="int"/>
    </attribute_node>
    <attribute_node name="date">
      <column name="todaysdate" type="date"/>
    </attribute_node>
  </element_node><!-- message id -->

  <element_node name="payment-request" multi_occurrence="YES">
    <element_node name="execution-date">
      <element_node name="month">
        <text_node>
          <column name="currentmonth" type="int"/>
        </text_node>
      </element_node>
      <element_node name="day">
        <text_node>
           <column name="currentday" type="int"/>
```

```
      </text_node>
  </element_node>
  <element_node name="year">
    <text_node>
      <column name="currentyear" type="int"/>
    </text_node>
  </element_node>
</element_node><!-- execution date -->

<element_node name="payment-request-id">
  <text_node>
    <column name="STATEMENT_ID" type="varchar(30)"/>
  </text_node>
</element_node>

<element_node name="amount">
  <text_node>
    <column name="totalprice" type="decimal(8,2)"/>
  </text_node>
</element_node><!-- amount -->

<element_node name="payment-network">
  <element_node name="other">
  <attribute_node name="network-name">
      <column name="bankname" type="varchar(20)"/>
    </attribute_node>
  </element_node>
</element_node><!-- payment network -->

<element_node name="payor">
  <element_node name="entity">
    <element_node name="name">
      <text_node>
        <column name="travelco" type="varchar(30)"/>
      </text_node>
    </element_node>
  </element_node><!-- entity -->

  <element_node name="bank-customer-info">
    <element_node name="bank">
      <element_node name="bank-code">
        <attribute_node name="type">
       <column name="payor_type" type="char(3)"/>
        </attribute_node>
        <text_node>
      <column name="payor_code" type="varchar(20)"/>
        </text_node>
      </element_node><!-- bank code -->

      <element_node name="entity">
        <element_node name="name">
          <text_node>
            <column name="payor_name" type="varchar(20)"/>
          </text_node>
        </element_node>
      </element_node><!-- entity -->
    </element_node><!-- bank -->

    <element_node name="account">
    <attribute_node name="type">
```

```
           <column name="p_act_type" type="char(9)"/>
         </attribute_node>

         <element_node name="id">
           <text_node>
             <column name="payor_account" type="char(6)"/>
           </text_node>
         </element_node>     <!-- id -->
        </element_node><!-- account -->
    </element_node><!-- bank customer info -->
  </element_node><!-- payor -->

  <element_node name="payto">
    <element_node name="entity">
      <element_node name="contact">
        <text_node>
          <column name="cruise_company" type="varchar(30)"/>
        </text_node>
      </element_node>
    </element_node><!-- entity -->

    <element_node name="bank-customer-info">
      <element_node name="bank">
        <element_node name="bank-code">
          <attribute_node name="type">
          <column name="payto_type" type="char(3)"/>
          </attribute_node>
          <text_node>
               <column name="payto_code" type="varchar(20)"/>
          </text_node>
        </element_node><!-- bank code -->

        <element_node name="entity">
          <element_node name="name">
            <text_node>
              <column name="payto_name" type="varchar(20)"/>
            </text_node>
          </element_node>
        </element_node><!-- entity -->
      </element_node><!-- bank -->

      <element_node name="account">
       <attribute_node name="type">
          <column name="c_act_type" type="char(9)"/>
        </attribute_node>

        <element_node name="key-id">
          <text_node>
            <column name="payto_account" type="char(6)"/>
          </text_node>
        </element_node>
       </element_node><!-- account -->
     </element_node><!-- bank customer info -->
  </element_node><!-- payto -->
</element_node>       <!-- end payment-request -->

<element_node name="signature-info">
  <element_node name="signature">
    <attribute_node name="algorithm">
     <column name="encoding" type="char(3)"/>
```

```
       </attribute_node>
      <text_node>
         <column name="person" type="varchar(20)"/>
      </text_node>
    </element_node><!-- signature -->

    <element_node name="certificate">
      <text_node>
        <column name="Cert" type="varchar(10)"/>
      </text_node>
    </element_node><!-- certificate -->
  </element_node><!-- signature info -->

 </element_node><!-- end bips-->
</root_node>
</Xcollection>
</DAD>
```

6.1.8 XML enablement commands

This section lists the commands that must be run to compose the PaymentRequest.xml document from the database tables:

1. The following SQL statement must be issued to store the bips1.dtd in the DTD repository table:

   ```
   INSERT INTO DB2XML.DTD_REF VALUES('/javateam/shoreline/dtd/bips1.dtd',
   DB2XML.XMLCLOBFROMFILE('/javateam/shoreline/dtd/bips1.dtd'), 0, 'author_name',
   'creator_name', NULL)
   ```

2. Since the dxxRetrieveXML() stored procedure is used to compose the XML document, enable the XML collection using the following commands:

   ```
   CALL PGM(QDBXM/QZXMADM) PARM(ENABLE_COLLECTION db_name SHORELINE.PAYMENT_INFO1
   '/javateam/shoreline/dad/paymentRequestCompose.dad')

   CALL PGM(QDBXM/QZXMADM) PARM(ENABLE_COLLECTION db_name SHORELINE.XML_PAYMENT
   '/javateam/shoreline/dad/paymentRequestCompose.dad')
   ```

6.1.9 Java source

This section shows the code snippet that composes the payment request XML document from the database tables with the defined DAD. It then writes the XML document to the IFS file and proceeds to read it so the program can later send it to the bank for processing:

```
...
// Remove old records from file by deleting records from XML_PAYMENT
stmt.executeUpdate("DELETE FROM SHORELINE.XML_PAYMENT");

// Call the stored procedure
cs = con.prepareCall("CALL db2xml.dxxRetrieveXML(?, ?, ?, ?, ?, ?, ?, ?, ?, ?)");

// Register the input parameter: collection name
cs.setString(1, "SHORELINE.XML_PAYMENT");

// Register the input parameter: result table name
cs.setString(2, "SHORELINE.XML_PAYMENT");

// Register the input parameter: result column name
cs.setString(3, "xml_document");
```

```
// Register the input parameter: valid column name
cs.setString(4, "valid_document");

// Register the input parameter: override type
cs.setInt(5, 0);

// Register the input parameter: override
cs.setString(6, "NO_OVERRIDE");

// Register the input parameter: max # of rows
cs.setInt(7, 500);

// Register the output parameter: actual # of rows
cs.registerOutParameter(8, Types.INTEGER);

// Register the output parameter: return code
cs.registerOutParameter(9, Types.INTEGER);

// Register the output parameter: error message text
cs.registerOutParameter(10, Types.VARCHAR);

// Run the stored procedure
cs.execute();
System.out.println("# of Rows: " + cs.getInt(8));
System.out.println("return code: " + cs.getInt(9));
System.out.println("message text: " + cs.getString(10));

if (!(cs.getInt(9) == 0))
throw new Exception("Error creating XML document -- error code: " + cs.getInt(9));

// Create the XML document -- /javateam/shoreline/paymentRequest.xml
stmt.executeQuery("SELECT DB2XML.CONTENT(XML_DOCUMENT,
'/JAVATEAM/SHORELINE/paymentRequest.xml') FROM SHORELINE.XML_PAYMENT");

// Get XML document into string. Use Toolbox API so that the data is converted to ASCII.
IFSTextFileInputStream in = null;
AS400 as400 = new AS400(sysName, dbUser, dbPassword);
in = new IFSTextFileInputStream(as400, "/javateam/shoreline/paymentRequest.xml");
xmlString = in.read(in.available());
in.close();
...
```

6.2 Decomposing paymentRequest

This section describes the decomposition of the paymentRequest XML document.

6.2.1 Database details

This XML document is decomposed into the payments DB2 table. The payments table contains the payment request information that is received from the travel agency. The layout of the payments table is listed in Table 6-6 on page 68.

Table 6-6 Payments

Key	Field name	Data type	Length	VarLen Alloc	Digits, DecPos	Description
P	statement_ID	varchar	30	10		Statement ID generated by the cruise company
	amount	decimal			8,2	Amount paid by the travel agency
	payor	varchar	30	20		Travel agency
	payor_account	char				Travel agency's account number
	payto	varchar	30	20		Cruise company
	payto_account	char				Cruise company's account number

6.2.2 XML document

An example of the XML document that we attempted to decompose into Table 6-6 can be found in the 6.1.2, “Starting XML document” on page 57 section of this document.

6.2.3 Discoveries

This section lists the discoveries we encountered while creating the DAD file used for the paymentRequest XML document:

1. For decomposition, in the DAD, all the <column> elements require *type* and *name* attributes. The DAD was missing the type attributes in the <column... > elements. For example:

```
...
<element_node name ="amount">
  <RDB_node>
    <table name="bank.payments"/>
    <column name="amount" type="decimal(8,2)"/>
  </RDB_node>
</element_node>
...
```

2. All <RDB_node> elements, other than the top one, must be enclosed in <attribute_node> or <text_node> elements. The DAD was missing <text_node> elements. For example:

```
...
<element_node name ="amount">
    <text_node>
      <RDB_node>
        <table name="bank.payments"/>
        <column name="amount" type="decimal(8,2)"/>
      </RDB_node>
    </text_node>
  </element_node>
...
```

3. In the DAD, <attribute_node> elements must be listed as the first elements of a table mapping. Within the DAD, we had to change <attribute_node> to <element_node>. At first, the DAD looked like this:

```
...
<element_node name="payment-request" multi_occurrence="YES">
  <element_node name="payment-request-id">
    <text_node>
      <RDB_node>
        <table name="bank.payments"/>
        <column name="statement_id" type="varchar(30)"/>
    ...
  <element_node name="bank-customer-info">
    <element_node name="account">
      <attribute_node name="type">
        <text_node>
          <RDB_node>
            <table name="bank.payments"/>
            <column name="payor_account" type="char(6)"/>
...
```

If the attribute, such as <type>, is not the first element of a table mapping, it must be changed to an <element_node>. In other words, all the <attribute_nodes> that refer to the same table must be listed as the eldest children of the first <element_node> that refers to the table. This requires changes in the DTD. Any attributes that become elements in the DAD also must become element nodes in the DTD. For example, in the DAD:

```
...
<element_node name="bank-customer-info" >
        <element_node name="account"  >
          <element_node name="type" >
            <text_node>
              <RDB_node>
                <table name="bank.payments"/>
                <column name="payor_account" type="char(20)"/>
...
```

4. The DAD must have tags that are unique. Within the XML document, any tag with the same name must map to the same column in the DAD. Any tag changes in the DAD also require tag changes in the DTD. For example, the second <type> tag must be changed to <type2> so that it can map to a separate column in the DB2 table:

```
...
<element_node name="bank-customer-info" >
    <element_node name="account"  >
      <element_node name="type" >
        <text_node>
          <RDB_node>
            <table name="bank.payments"/>
            <column name="payor_account" type="char(20)"/>
  <element_node name="bank-customer-info">
    <element_node name="account">
      <element_node name="type2">
        <text_node>
          <RDB_node>
            <table name="bank.payments"/>
            <column name="payto_account" type="char(20)"/>
...
```

5. In the DAD, multi_occurrence="YES" must be specified after an element_node when there are multiple child elements under the element_node that are mapped to tables. For example, both statement_id and amount map to a DB2 table and both are child elements of <payment-request>:

```
...
<element_node name="payment-request" multi_occurrence="YES">
    <element_node name="payment-request-id">
      <text_node>
        <RDB_node>
          <table name="bank.payments"/>
          <column name="statement_id" type="varchar(30)"/>
        </RDB_node>
      </text_node>
    </element_node>
    <element_node name ="amount">
       <text_node>
        <RDB_node>
          <table name="bank.payments"/>
          <column name="amount" type="decimal(8,2)"/>
...
```

6. A trigger (Payments_Processed) was added to the payments table to allow the bank to determine which statement IDs have been decomposed from the paymentRequest.xml document. This allowed the bank to send the paymentResponse.xml document to the travel agency.

> **Note:** The SQL source for this trigger can be found in Section 6.2.9, “Trigger source” on page 73.

6.2.4 Final DTD

An example of the final DTD can be found in the 6.1.4, “Final DTD” on page 59 section of this document.

6.2.5 Final XML document

An example of the final XML document can be found in the 6.1.5, “Final XML document” on page 60 section of this document.

6.2.6 DTD mapping scheme

Figure 6-2 shows how the structure of the XML document relates to the DB2 table that is used for the decomposition. This was helpful in the creation of the DAD file because it shows how the two structures compare.

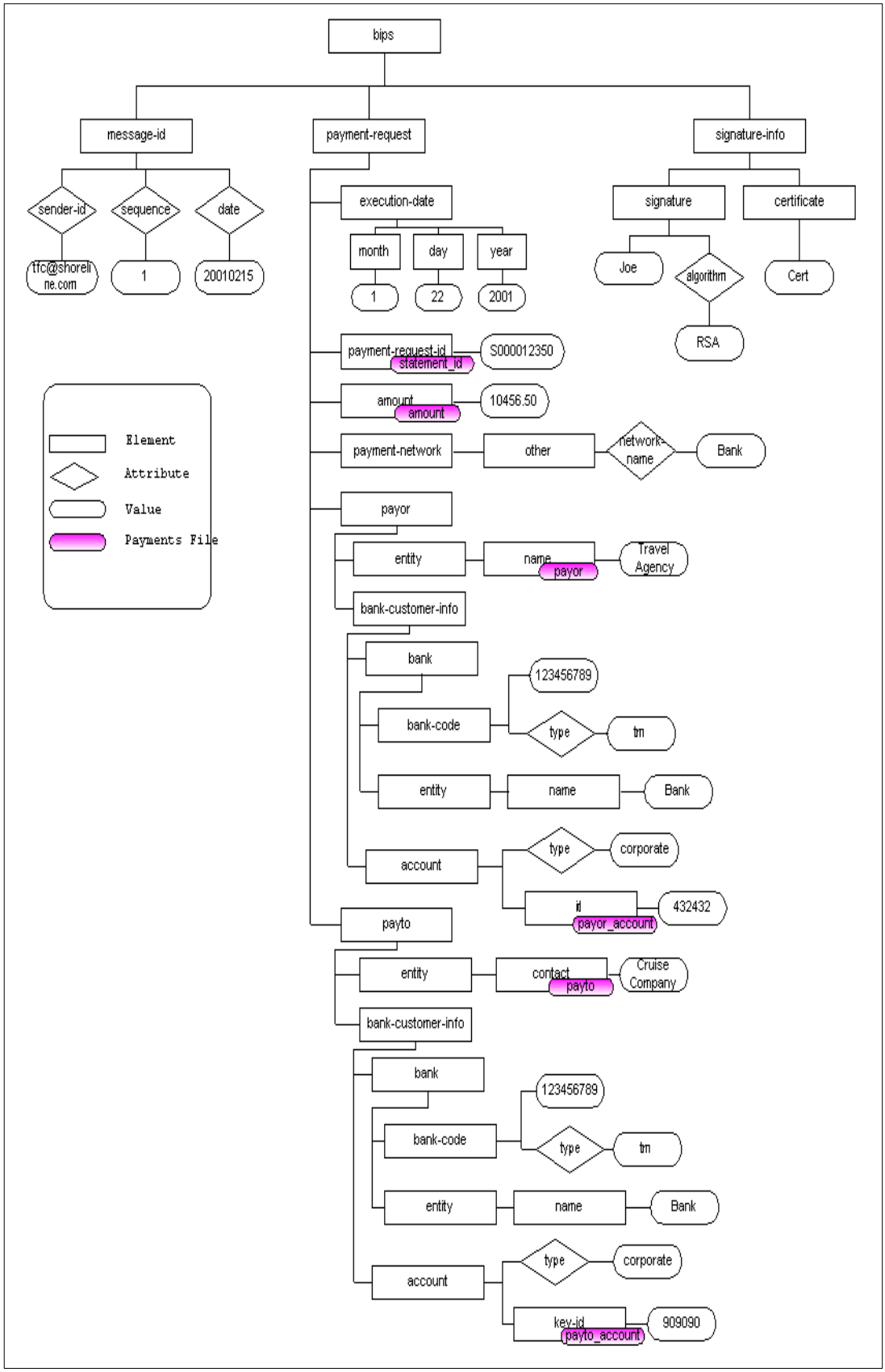


Figure 6-2 Decomposing payment mapping scheme

6.2.7 Final DAD

This section shows the final DAD document used to decompose the XML document after incorporating all of the discoveries. This DAD file for an XML collection uses RDB_node mapping:

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "/javateam/shoreline/dtd/dad.dtd">
<DAD>
<dtdid>/javateam/shoreline/dtd/bips1.dtd</dtdid>
<validation>YES</validation>
<Xcollection>
<prolog>?xml version="1.0"?</prolog>
<doctype>!DOCTYPE bips SYSTEM "/javateam/shoreline/dtd/bips1.dtd"</doctype>
<root_node>
  <element_node name="bips">
    <RDB_node>
      <table name="bank.payments" key="statement_id"/>
    </RDB_node>
    <element_node name="payment-request" multi_occurrence="YES">
      <element_node name="payment-request-id">
        <text_node>
          <RDB_node>
            <table name="bank.payments"/>
            <column name="statement_id" type="varchar(30)"/>
          </RDB_node>
        </text_node>
      </element_node>
      <element_node name ="amount">
        <text_node>
          <RDB_node>
            <table name="bank.payments"/>
            <column name="amount" type="decimal(8,2)"/>
          </RDB_node>
        </text_node>
      </element_node>
      <element_node name ="payor">
        <element_node name="entity">
          <element_node name ="name">
            <text_node>
              <RDB_node>
                <table name="bank.payments"/>
                <column name="payor" type="varchar(30)"/>
              </RDB_node>
             </text_node>
          </element_node>
        </element_node>
        <element_node name="bank-customer-info">
          <element_node name="account">
            <element_node name="id">
            <text_node>
              <RDB_node>
                <table name="bank.payments"/>
                <column name="payor_account" type="char(6)"/>
              </RDB_node>
            </text_node>
          </element_node>
          </element_node>
        </element_node>
      </element_node>
      <element_node name="payto">
```

```
        <element_node name="entity">
          <element_node name="contact">
            <text_node>
              <RDB_node>
                <table name="bank.payments"/>
                <column name="payto" type="varchar(30)"/>
              </RDB_node>
             </text_node>
          </element_node>
        </element_node>
        <element_node name="bank-customer-info">
          <element_node name="account">
           <element_node name="key-id">
            <text_node>
              <RDB_node>
                <table name="bank.payments"/>
                <column name="payto_account" type="char(6)"/>
              </RDB_node>
             </text_node>
            </element_node>  <!-- key-id -->
          </element_node><!-- account -->
        </element_node>   <!-- bank-customer-info -->
      </element_node>   <!-- payto -->
    </element_node>    <!-- payment-request -->
  </element_node>      <!-- bips -->
</root_node>
</Xcollection>
</DAD>
```

6.2.8 XML enablement commands

This section lists the commands that must be run before decomposing the PaymentRequest.xml document into the database tables:

1. We used the DTD to validate XML data in the XML collection (VALIDATION set to YES in the DAD file), so the following SQL statement had to be issued to store the Bips1.dtd in the DTD repository table:

   ```
   INSERT INTO DB2XML.DTD_REF VALUES('/javateam/shoreline/dtd/bips1.dtd',
   DB2XML.XMLCLOBFROMFILE('/javateam/shoreline/dtd/bips1.dtd'), 0, 'author_name',
   'creator_name', NULL)
   ```

2. Since we used the dxxInsertXML() stored procedure, we enabled the XML collection using the following commands:

   ```
   CALL PGM(QDBXM/QZXMADM) PARM(ENABLE_COLLECTION db_name BANK.PAYMENTS
   '/javateam/shoreline/dad/paymentRequest.dad')
   ```

6.2.9 Trigger source

The SQL statement used to add the trigger program to the specified table is listed in Table 6-7.

Table 6-7 Trigger source

Trigger name	Table	Description
BANK.PAYMENTS_PROCESSED	Bank/Payments	CREATE TRIGGER BANK.PAYMENTS_PROCESSED AFTER INSERT ON BANK.PAYMENTS FOR EACH ROW MODE DB2ROW BEGIN UPDATE BANK.PAYMENTS SET PROCESSED = 'N' WHERE BANK.PAYMENTS. PROCESSED IS NULL ; END ;

6.2.10 Java source

This section shows an example of the code snippet that decomposes the PaymentRequest XML document into the database table with the defined DAD:

```
...
try   {
  Connection con = ds.getConnection(DbUser, DbPassword);

  // Call the stored procedure
  CallableStatement cs = con.prepareCall("CALL db2xml.dxxInsertXML(?, ?, ?, ?)");

// Register the input parameter: collection name
  cs.setString(1, "BANK.PAYMENTS");

  // Register the input parameter: xml file
  cs.setObject(2, xmlString);

  // Register the output parameter: return code
  cs.registerOutParameter(3, Types.INTEGER);

  // Register the output parameter: error message text
  cs.registerOutParameter(4, Types.VARCHAR);

  // Run the stored procedure
  System.out.println("prior to execute");
  cs.execute();
  System.out.println("after execute");
  System.out.println("return code: " + cs.getInt(3));
  System.out.println("message text: " + cs.getString(4));
...
```

6.3 Saving PaymentResponse

This section describes the storing of the paymentResponse XML document.

6.3.1 Database details

This XML document is stored into the accounts_payable DB2 table. The accounts_payable table contains the payment response XML document that is received from the bank. The layout of the accounts_payable table is listed in Table 6-8.

Table 6-8 Accounts_payable

Key	Field name	Alias name	Data type	Length	Allow Nulls	Description
	PDSTATUS	PAID_STATUS	CHAR	1		Paid Status ► O=unprocessed ► P=paid ► R=rejected ► S=send
	XMLSTMT	XML_STATEMENT	XMLCLOB			XML Statement Document
	XMLPAYRES	XML_PAYMENT_RESPONSE	XMLVARCHAR		Y	XML Payment Response Document

6.3.2 Discoveries

No new discoveries were found.

6.3.3 Final DTD

An example of the final DTD can be found in 6.1.4, “Final DTD” on page 59.

6.3.4 Final XML document

This section shows an example of the XML document that is stored in Table 6-8:

```
<?xml version="1.0"?>
<!DOCTYPE bips SYSTEM "/JavaTeam/Shoreline/dtd/bips1.dtd">
<bips>
  <payment-response>
    <payment-response-id>s000000006</payment-response-id>
    <request-message-id sender-id="tfc@shoreline.com" date="20010516" sequence="1"/>
    <payment-request-id>s000000006</payment-request-id>
    <statement>
      <code>200</code>
      <description>200</description>
      <contextual-info></contextual-info>
    </statement>
  </payment-response>
</bips>
```

6.3.5 XML enablement commands

Since we used the database table to store the XML document, no special enablement was required.

6.3.6 Java source

The following code snippet stores the payment response XML document into the accounts_payable database table:

```
...
// creating Byte array from String
byte buf[] = new byte[15000];
buf = aXMLString.getBytes();

// Save into /Javateam/Shoreline/xml/PaymentResponse.xml on the system
AS400 as400 = new AS400(sysName, dbUuser, dbPassword);
IFSFileOutputStream file = new IFSFileOutputStream(as400,
"/javateam/shoreline/xml/PaymentResponse.xml", 437);

// Write a String to the file (don't convert characters).
File.write(buf);

// Close the file
file.close();

// inserting into DB
stmt = con.createStatement();

stmt.executeUpdate("UPDATE SHORELINE.ACCOUNTS_PAYABLE SET XML_PAYMENT_RESPONSE =
DB2XML.XMLVARCHARFROMFILE('/JAVATEAM/SHORELINE/XML/PAYMENTRESPONSE.XML') WHERE
DB2XML.EXTRACTVARCHAR(XML_STATEMENT, '/Invoice/InvoiceHeader/InvoiceNumber') = '" +
aStatementID + "'");
...
```

7

Miscellaneous discoveries

This chapter contains discoveries that we found while performing DB2 XML Extender testing. These discoveries may be helpful if you are going to implement XML Extenders into your environment:

1. Except for the iSeries server, XML Extender is shipped with the parser on all platforms on which the product runs. In the case of the iSeries server, the parser is part of the operating system, so the customer must install a new operating system level to get a new parser level. On the iSeries server (V5R1), the parser level is 3.1. On other platforms, the parser level is 3.3.
2. You cannot restore database tables with RSTLIB when they contain XML-enabled columns because:
   - Important meta data that is stored in the XML Extender is not restored to the new system when you restore your library and database tables. This meta data can only be created on the target system by running the enable_column command.
   - When you restore your library with RSTLIB, SQL triggers in your library are unusable because the prerequisite meta data is missing from the XML Extender. The presence of these triggers prevent you from running the enable_column command.

   Note: See discovery number 6 on page 79 for additional information.

3. When running XML Extender User Defined Functions (UDFs), be aware of the following commitment control guidelines:
   - XML Extender assumes that the application handles COMMIT and ROLLBACK, but never performs these actions.
   - We recommend that an application design include any XML Extender UDF, XML Extender stored procedures, and any INSERT, UPDATE, or DELETE that uses XML Extender triggers, within one commitment control definition. This commit definition should be completed by your application with the appropriate COMMIT or ROLLBACK.
   - XML Extender UDFs require *CS, and the application should perform an explicit COMMIT after running any UDFs, stored procedures, or SQL statements that INSERT, UPDATE, or DELETE documents in an XML column.

© Copyright IBM Corp. 2001

– The application must explicitly perform a COMMIT before performing another SQL action on the same row that has already been updated by an XML Extender UDF or another SQL query.

These guidelines are provided to address the following error conditions:

– **Problem**: You received the message "Record 1 member MYTABLE already locked to this job." after inserting or updating data using XML Extender UDFs.

Explanation: The application has inserted or updated data using XML Extender UDFs and then attempted to change or delete a new row of data before performing a COMMIT.

The XML extender assumes that it is run within a commitment control definition initiated by the application. It also assumes the application performs a COMMIT or ROLLBACK once a unit of work is completed. The XML Extender never executes an SQL COMMIT.

Corrective action: Perform an explicit COMMIT or ROLLBACK at the completion of the unit of work. We recommend that you include the XML Extender UDF, XML Extender stored procedures, and any INSERT, UPDATE, or DELETE within a commitment control definition.

– **Problem**: You received the message "Changes waiting for COMMIT or ROLLBACK." when using STRSQL (Interactive SQL) to call an XML Extender UDF on a column that has been XML-enabled.

Explanation: This message can occur when a STRSQL session running COMMIT(*NONE) has ended under the following circumstances:

- An SQL commitment control definition was started within the body of a UDF or a trigger that requires COMMIT(*CS).
- The application has not issued an explicit COMMIT before terminating.

This message is for diagnostic purposes and can be ignored.

4. When re-using prestart jobs, such as QSQSRVR, the following errors may be found in the job log for the prestarted job:

```
Tried to refer to all or part of an object that no longer exists.
Application error. MCH3402 unmonitored by QZXMDB2XML at statement 0000000030,
instruction X'0000'.
Trigger program or external routine detected an error.
```

The workaround for this problem is to change the number of times the pre-started job is used by issuing the following command:

```
CHGPJE SBSD(QSYSWRK) PGM(QSQSRVR) MAXUSE(1)
```

5. While using the DB2 XML Extender product, we struggled with the lack of appropriate error messages. The end user should search for messages that help them determine the cause of the problems they encounter.

Listed below are some helpful tips on diagnosing errors:

– There is valuable information put into the job logs by the system when an error occurs. The Toolbox JDBC uses the QZDASOINIT prestart jobs and the Native JDBC uses the QSQSRVR jobs. If you are using QShell, you can find the appropriate job log using WRKOUTQ QEZJOBLOG.

– There is also a trace tool available that can help with diagnosing problems. The tool is turned on for a particular user ID. Once turned on, verbose detail is gathered in a trace file for all jobs running under that user ID. It is not separated by thread, so it can become very confusing if it is not used carefully. Also, the size of the trace file is not

limited. Therefore, if it is turned on and forgotten, the file can become large. The name of the trace file is:

```
'/path/dxxNNNNN.trc'
```

Here, '/path' is wherever you want the detail saved (you must have R/W permission to the directory path). *NNNNN* is the numeric uid assigned for user ID.

To turn trace on and off using native OS command line, use the following commands:

```
CALL QDBXM/QZXMTRC PARM(ON UserID '/path')
CALL QDBXM/QZXMTRC PARM(OFF UserID)
```

To turn trace on and off using QShell, use the following commands (these require /usr/bin in your environment variable 'PATH'):

```
dxxtrc on UserID /path
dxxtrc off UserID
```

Once you turn trace on, you can delete or edit the dxxNNNNN.trc file to add markers, delete detail, clear it, etc. When you turn trace off, the trace file is not deleted. When you turn trace on again, the trace information will be appended to the trace file.

- If an error message does not contain a DXX or SQL code, it is probably due to the parser rejecting an invalid document. You can try to run DOMPrint (parser demo program) on your XML document first to flush out all of the XML syntax errors. It prints out the document, if it is valid, and reports errors if it is not. The DOMPrint is available in the xerces samples.

6. On OS/400, save and restore for schemas have the following restrictions:
   - Do not save, restore, or delete the DB2XML schema (library).
   - You can restore user created schemas that contain database tables used by XML Extender under the following conditions:
     - Schemas that contain XML collections, but do not contain XML-enabled columns, can be restored at the library level so long as the database on the target system has been enabled for XML Extender. If the XML collection is enabled on the original system, you must re-enable the XML collection on the target system.
     - Schemas that contain columns of XML user-defined type (XMLCLOB, XMLVarchar, etc.) can be restored at the library level so long as the column has not been enabled for XML and the database on the target system has been enabled for XML Extender.
     - Schemas that contain columns that have been enabled for XML cannot be restored at the library level. The base table and the side tables (database tables) can be restored using RSTOBJ.

The following sections provide steps for restoring schemas with database tables that are used with XML collections and XML columns.

Restoring XML collection database tables

To restore database tables to another system when they are part of an enabled XML collection, perform the following steps:

1. Enable the database on the new system for XML Extender.
2. Restore the XML collection database tables using RSTLIB.
3. If the XML collection was enabled on the original system, run the enable_collection command to enable the XML collection on the target system.

For more detailed information about completing the XML Extender enablement steps, see the *DB2 Universal Database for iSeries XML Extender Administration and Programming Guide*, which is available on the Web at:

http://publib.boulder.ibm.com/pubs/html/as400/v5r1/ic2924/books/c2711720.pdf

Restoring database tables with XML user-defined type

To restore database tables with XML user-defined type, perform the following steps:

1. Enable the database on the target system for XML Extender.
2. Restore the database tables using RSTLIB.

Restoring XML column database tables

To restore database tables that contain columns enabled for the XML column, perform the following steps:

1. Enable the database on the target system for XML Extender.
2. Restore the base table by using RSTOBJ.
3. Remove old triggers defined in the base table using RMVPFTRG.
4. Enable the XML column on the target system. You must use the -r parameter to identify the primary key of the base table if the -r parameter was used to enable the base table on the previous system.
5. Add user-defined triggers to the base table using ADDPFTRG and restore those programs on the target system.
6. Restore the data to the side tables using RSTOBJ.

For more detailed information about completing the XML Extender enablement steps, see the *DB2 Universal Database for iSeries XML Extender Administration and Programming Guide.*

8

Final thoughts

When using the DB2 XML Extender product, you should have a good understanding of both DB2 and XML. DB2 knowledge is required to create and manipulate tables. XML knowledge is required to create DADs and DTDs. Creating DAD files is complicated. This is partially due to the XML restrictions documented in the Discoveries sections of this Redpaper. In our case, the situation was further complicated due to the absence of the Web Services Wizard tool that was not available during the development of this scenario. The Web Services Wizard that will be made available for the iSeries server later this year can be found at:

http://www.alphaworks.ibm.com/aw.nsf/frame?ReadForm&/aw.nsf/techmain/751C2A483B9B2387852569920001AC96

© Copyright IBM Corp. 2001

Special notices

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter 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 the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

The following terms are trademarks of other companies:

Tivoli, Manage. Anything. Anywhere.,The Power To Manage., Anything. Anywhere.,TME, NetView, Cross-Site, Tivoli Ready, Tivoli Certified, Planet Tivoli, and Tivoli Enterprise are trademarks or registered trademarks of Tivoli Systems Inc., an IBM company, in the United States, other countries, or both. In Denmark, Tivoli is a trademark licensed from Kjøbenhavns Sommer - Tivoli A/S.

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

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

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

PC Direct is a trademark of Ziff Communications Company in the United States and/or other

© Copyright IBM Corp. 2001

countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States and/or other countries.

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

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

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