XWDB Developer Guide

***Draft**: Software version 1.2.0, December 2002*

XWDB Developer Guide contains extracts from Designing XML Databases *by Mark Graves, Copyrighted by Prentice Hall 2002. All references to Chapters, Sections, Figures, Tables, or Examples named DXD refer to that book. It also contains additional information about using more recent versions of the software than the version published in the book, so this Developer Guide supersedes what is described in the book. (Most of the changes to the documentation describe extensions to the software.) For more recent versions of this document, check the XWDB web site,* http://www.xwdb.org/.

This is a DRAFT version of the documentation intended to update the technical information in Designing XML Databases.

Table of Contents

1 Relational Data Server (RServe)

1.1 Background

A *relational database adaptor* makes data from a relational DBMS available in XML. It works by querying the database and formatting the report from the relational DBMS as XML. A *relational data server* is a database adaptor combined with a web server. RServe is a relational data server consisting of web application for generating XML reports from relational queries and loading relational data using XML.

1.1.1 Rendering Relational Data

The basic process for a user requesting an XML report from a relational DBMS using a web browser (see DXD Figure 5-2) is as follows:

1. The user specifies a relational query to the web browser as a URL.
2. The web browser sends the URL to the data server.
3. The data server parses the URL request and creates a SQL query.
4. The data server passes the SQL query to the database server.
5. The database server executes the query.
6. The database server returns the relational report to the data server.
7. The data server formats the report as XML.
8. The data server returns the XML report to the web browser.
9. The web browser parses the XML report and displays it to the user.

When a stylesheet is used, it must also be retrieved and parsed by the web browser.

1.1.1.1 Formatting a Report as XML

One way to format relational data as XML is to use the table names and column names to drive the creation of element type names (and/or attribute names). Information on primary keys and foreign keys can be used to create a document hierarchy. One issue in this approach is that some special characters may be used in table and column names that are not valid as XML element type names. A second issue is that following the foreign keys may cycle (lead to an infinite loop). A cycle forms when a foreign key refers to a table with a foreign key that refers back to the original table, either directly or through intermediate foreign

key constraints. This is addressed by keeping track of the tables followed or by limiting the number of foreign key constraints that may be followed.

The process for formatting a report as XML is:

1. Write the header of the document.
2. Write the root element start tag.
3. Retrieve the column names from the meta-data, if they are to be used as element type names.
4. Iterate through each record in the relational report.
   4.1. Write a start tag for the relation (such as the name of the table, something generic like “record”, or a user-specified string).
   4.2. Iterate through each column in the record.
      4.2.1. If the record value for that column is a string, write it out surrounded by start and end tags.
      4.2.2. If that column has a foreign key constraint, consider recursively writing out an element for the record to which this column value refers.
   4.3. Write an end tag for the relation.
5. Write an end tag for the root element.

This algorithm may give poor performance on large databases because of the number of queries that are performed. In addition, if the foreign keys are followed for many steps (in effect, the resulting XML tree is deep), then duplicate queries will occur when duplicate foreign key values are retrieved. The algorithm does not take advantage of structural information contained in the relational database, such as foreign key constraints. This additional information may be utilized to create queries that retrieve information from multiple rows (or across multiple tables) in a single query. This allows the data for multiple elements to be retrieved by a single query.

1.1.1.2 Extracting Dictionary Data

A commercial relational DBMS provides information in its system tables that may be useful to extract in developing a data server. For example, the primary keys of a relation provide a unique index of all the records in the relation and may be used to refer to the records. Foreign keys are references to the primary key of another relation and may be used to link an element with its embedded elements. For example, if a foreign key in a "purchase order" table refers to the primary key of a "customer" table, then the XML that presents the data in a purchase order

table may include an XML presentation of the customer record as a subelement. Most relational DBMSs store the information about primary keys and foreign keys in system tables. SQL queries can be used to extract that information from the system tables for use by the data server.

Foreign key constraints may be exploited while *emitting* (i.e., writing out) the relational data as an XML tree. The value of a column (or columns) may consist of values from the primary key of another relation. The primary key uniquely identifies a record in that relation (by definition), and the foreign key value may be substituted with the contents of the record to which it refers.

The data in a relational database can be mapped to XML as a tree by following parent/child relations in the database to create parent/child relationships in the XML tree. The parent/child relations in the relational database are typically encoded as foreign key constraints.

Warning for database developers: A potential point of confusion exists in the parent/child terminology. In relational design, a parent/child functional dependency relationship is being discussed; thus, the child depends upon the parent for information. In trees, the parent is composed of children; thus, the parent depends upon the child for information. Thus, a parent/child relation in relational terms is the *inverse* relation to the parent/child relation in terms of mapping relational data to an XML tree. For example, in a relational database where employees are employed within a department, the department would be the parent relation and the employee relation would be the child. In formatting the data as XML directly from the employee relation, the employee element is the parent and would have a department element as a child. (See DXD Figure 5-4).

The micro-array relational schema in DXD Figure 5-5 has foreign keys described in DXD Table 5-1. Using those foreign keys, XML may be generated as shown in DXD Example 5-10. The foreign keys in DXD Table 5-1 are followed to create the embedded elements.

In Oracle, the SQL code in DXD Example 5-11 may be used to extract the foreign key information from the system tables. For example the foreign keys of the fine-grained relational storage system from DXD Chapter 4 are shown in DXD Table 5-2. The SQL code in DXD Example 5-12 performs a similar function for DB2.

1.1.2 Relational Data Loader

XML documents to be loaded may come from another application, an external resource (such as database or flat file), or a data entry form. One way to create an XML document from a data entry form is to create a script or program that formats the HTML form data as XML. A Java applet with a graphical user

interface may also be used to create an entry form. This is discussed more in DXD Chapter 7.

We have implemented a loader for relational data, called rLoad, in Java that demonstrates the issues in loading XML data into a relational database. The source code for rLoad is given in DXD Example 5-6. The application loads XML data in two steps:

1. Translate the XML data using XSL stylesheets to an XML document consisting of “record” and “field” tags with attributes for the table being loaded and for other load-specific meta-data. For example, the XML document in DXD Example 5-7 can be translated using a stylesheet to a rLoad XML document, such as the one shown in DXD Example 5-8. Stylesheets are discussed more fully in DXD Chapter 7, but a simple stylesheet to translate genes for the micro-array example is given in DXD Example 5-9.
2. Load the "record/field" data into the database using a SAX based parser.

The record/field XML document, called a rloadXML document, consists of “record” and “field” elements and some additional meta-data to perform the following tasks:

- Specify into which relation table the record should be loaded. In the document, records to be loaded into different tables may be interspersed. Additionally, a record for one table may be embedded within a record to be loaded into another table. This simplifies the generation of the rloadXML document.
- Specify whether an embedded record should be loaded before or after the record in which it is embedded. Allowing this option increases the flexibility in which records may be defined by the XSL stylesheet translation without concern over foreign key dependencies. For example in an XML document, employees may specify the department to which they belong or a department may list the employees that belong to it. When loading, the independent record can be created first, then the dependent one, regardless of which one occurs as an embedded record in the XML data source.
- Specify that a field value should be obtained by a DBMS sequence generator to create a unique identifier.
- Specify that a field value should be the most recently generated value for a given sequence. Thus, a dependent record can refer to the unique identifier generated for the record upon which it depends.
- Specify that the whitespace should be trimmed or removed from the character data in the field element.

- Specify that a value should be the identifier of a record in another table whose specified column equals the value in the field element. This allows for creation of foreign key constraints on the unique identifiers generated by the DBMS while using alternate keys or columns with unique constraints for date entry and loading.

The Java implementation consists of three classes: the main LoadXML class, a JDBCLoadHandler class for use by the SAX parser, and a Record container class that contains the information for each record to be loaded. The “main” method of LoadXML takes the rloadXML file as input and calls a “parse” method that creates a SAX parser with JDBCLoadHandler as the handler. JDBCLoadHandler creates a Record object and fills in the fields as they are parsed. If more than one relational record is needed for an XML “record” element, then multiple Record objects are created and stored on a stack.

1.2 Implementation

Java code that implements the RServe relational data server is given in DXD Example 5-13. A UML class diagram is given in DXD Figure 5-6.

The main class in RServe, called Demo, translates relational data from the source specified by the Input class into XML and writes the data on the stream specified by the Output class. The query, account, and report formatting information are captured by the container class AccessSpec and its helper classes QuerySpec and ReportSpec. The QuerySpec class contains the constraints to be used in the “where” clause of the SQL query, making use of a ConstraintSpec helper object to hold each constraint. The Input class formats the SQL query, using the QuerySpec class for formatting the “where” clause, and the Input class interacts with the relational database (using helper classes from org.xwdb.xmldb.util.rdb described in Appendix A).

The query is executed by creating an AccessSpec object and passing it to the "writeDoc" method of the FormatXML class. The “writeDoc” method writes the XML document header and root element start and end tags and calls the “writeTable” method. The “writeTable” method implements the algorithm described in DXD Section 5.3.3. It uses the Input object (stored in inputSource field) to retrieve data from the database and the Output object (stored in outputSource field) to write the formatted XML. The “writeTable” method uses the “writeValue” method to write each data item, which may, recursively, call the “writeTable” method again if the data item refers to another table’s record as part of a foreign key constraint.

The primary key constraints are contained in an instance of the MetaDataRepositoryTab class. The Meta-Data Repository for Table data contains

a collection of MetaDataRecordTab(s). An instance of MetaDataRecordTab contains the information for one primary key constraint. The repository is initialized when it is first accessed. The MetaDataRepositoryTab is a hashtable of MetaDataRecordTab that is keyed off the table name and column name of the primary key. Each MetaDataRecordTab contains the table name and column of the primary key referred to by the hashtable entry. There is also a method in JDBC that could be used to access the primary key system table to initialize the table, but it was not used in this example, to make the approach more applicable to other protocols.

The foreign key constraints are contained in an instance of the MetaDataRepositoryCol class. The Meta-Data Repository for Column data contains a collection of MetaDataRecordCol(s). An instance of MetaDataRecordCol contains the information for one foreign key constraint. The repository is initialized when it is first accessed. The MetaDataRepositoryCol is a hashtable of MetaDataRecordCol, which is keyed off the table name and column name of the foreign key that would be followed to drill down in formatting the report. Each MetaDataRecordCol contains the table name and column of the primary key referred to by the hashtable entry.

1.3 Usage

1.3.1 Specifying URL Requests

The API to a relational data server may be specified using URL requests. These are described in this section before delving into the internals of RServe.

RServe allows for read-only access of relational data via URLs. A servlet is called with parameters such as the table name, constraints on the columns, and the relational database account to be accessed. An alternative implementation would allow for specifying a SQL statement as part of the URL.

Some example URLs for RServe are

- *http://127.0.0.1/servlet/org.xwdb.xmldb.rserve.XMLServlet?tablename=company* -- which retrieves all the records in the "company" table.
- *http://127.0.0.1/servlet/org.xwdb.xmldb.rserve.XMLServlet?tablename=company&stylesheet=/ss/generic_table.xsl* -- which retrieves all the records in the "company" table, then formats the data using the stylesheet http://127.0.0.1/ss/generic.xsl.

- *http://127.0.0.1/servlet/org.xwdb.xmldb.rserve.XMLServlet?tablename=company&id=12* -- which retrieves the record in the "company" table with id equal to "12".
- *http://127.0.0.1/servlet/org.xwdb.xmldb.rserve.XMLServlet?tablename=company&stylesheet=/ss/generic_table.xsl&name=Acme* -- which retrieves the record from the "company" table with name equal to "Acme", then formats the data using the stylesheet http://127.0.0.1/ss/generic.xsl.

There are three main components of the URL: the base, the stylesheet, and the query. The base refers to the servlet (or cgi-script) that provides the data service, in this case the RServe servlet installed on whatever machine the URL requests (as specified by the reserved IP address for localhost 127.0.0.1). The optional stylesheet specifies the XSL stylesheet that should be used to translate the XML to HTML for the browser. The stylesheet and query can occur in any order.

The base is something like

- *http://127.0.0.1/servlet/org.xwdb.xmldb.rserve.XMLServlet?*
- *http://mymachine/servlet/org.xwdb.xmldb.rserve.XMLServlet?*

with the trailing "?".

The stylesheet is "stylesheet=/ss/*generic_table*.xsl". The stylesheet may also be omitted in a browser, and the default stylesheet for the browser will be used. This will usually show the text and tags of the XML document.

The query is built up from the tablename and column names and the individual name-value pairs are separated by “&”.

To retrieve all the records in a table, the query “tablename=<table>” (where <table> is the name of the table). To retrieve a specific entry in the table with a specified id “tablename=<table>&id=<id>” is used. Any column name can be used as part of the query.

In addition, the data server has a facility for specifying the DBMS account to use. The name-value pair “acct=<user>/<password>@<instance>”. Any database in the DBMS may be accessed through the URL. For example:

- *http://127.0.0.1/servlet/org.xwdb.xmldb.rserve.XMLServlet?tablename=dept&stylesheet=/ss/generic_table.xsl&acct=scott/tiger@ORCL*

A couple of other things happen when creating the URL. When using HTML forms, a parameter name such as “Submit” is filled in automatically. The "Submit" parameter is ignored by the data server. URLs are encoded for reserved characters using their hexadecimal ASCII representation or another key, so for

example, “/” is encoded “%2F” and spaces are encoded with “+”. An actual URL generated by a HTML form looks like

- *http://127.0.0.1/servlet/org.xwdb.xmldb.rserve.XMLServlet?tablename=company&stylesheet=%2Fss%2Fgeneric_table.xsl&name=Acme&Submit=Find+company*

This is equivalent to

- *http://127.0.0.1/servlet/org.xwdb.xmldb.rserve.XMLServlet?tablename=company&stylesheet=/ss/generic_table.xsl&name=Acme*

1.3.2 Creating a SQL Query

The relational data server uses the parameters of the URL to build a SQL statement. For example, a request for the contents of a table may be built by adding the value of the "tablename" parameter to the following template:

```
select * from <tablename>
```

More complex queries can be built incrementally by adding column constraints as a “where” clause, sorting constraints as an “ORDER BY” clause, and the columns to be displayed as part of the select statement as:

```
select <report columns> from <tablename>
  where <conditions>
  order by <sort constraint>
```

Much of the functionality of a SQL statement can be encoded as a URL, though the "where" conditions require a little more creativity. Because the name/value pair must be separated by an “=” sign, the other operations may be encoded by pre-pending the operation to the value. For example:

```
...&partno=1234&quantity=20&... (quantity equal 20)
...&partno=1234&quantity=<20&... (quantity less than 20)
...&partno=1234&quantity=<=20&... (quantity less than or equal
      20)
```

1.3.3 XSL-Based User Interfaces

XSL-based user interfaces can be used to generate HTML. This section introduces XSL stylesheets, describes how to render an XML document as a table or record, and explains how to expand those depictions with drill-downs, proxies, and customizations.

XML with XSL is useful particularly when the presentation of the data needs to vary based on the client. For example, you might use different style sheets to take advantage of features of different web browsers, or you can use XSL to generate another markup language, such as WML for cell phones or other wireless devices.

1.3.3.1 Rendering XML as a Table

XML may be rendered as a table using an XSL stylesheet as shown in DXD Figure 7-1. The example stylesheet document in DXD Example 7-2 renders as a table an XML document of elements with a table-like subelement structure.

The stylesheet document in DXD Example 7-2 consists of the XML header and an XSL stylesheet element. The stylesheet element consists of an attribute that defines the namespace as the version used by XSLT and two templates as subelements. An output element is defined as part of the stylesheet to specify HTML output syntax. The first template is executed at the root of the source document (denoted by the slash "/" character), and the second template is executed when it matches any element with the element type name "collection". These sections are outlined here:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:output method="html"/>
   <xsl:template match="/">
    ...
   </xsl:template>
   <xsl:template match="collection">
    ...
   </xsl:template>
</xsl:stylesheet>
```

The root template creates an HTML table by interspersing HTML and XSL commands. The "xsl:" namespace prefix distinguishes the XSL commands from the HTML commands. The root template is:

```
<xsl:template match="/">
  <TABLE STYLE="border:1px solid black">
     <TR STYLE="font-weight:bold; text-decoration:underline">
        <xsl:for-each select="collection/*[1]/*">
          <TD>
            <xsl:value-of select="name()"/>
          </TD>
        </xsl:for-each>
     </TR>
     <xsl:apply-templates />
  </TABLE>
</xsl:template>
```

To understand the template, examine it by looking at the HTML code and the XSL code separately. The HTML code is

```
<TABLE STYLE="border:1px solid black">
     <TR STYLE="font-weight:bold; text-decoration:underline">
        For each column name,
          <TD>
            The name of the column goes here.
          </TD>
```

```
      </TR>
    </TABLE>
```

The HTML commands create a “TABLE” with a particular style and create the first row in the table (again with a particular style). The row consists of a collection of element type names that become the names of the columns.

The XSL code is

```
<xsl:template match="/">
    <xsl:for-each select="collection/*[1]/*">
        <xsl:node-name />
    </xsl:for-each>
    <xsl:apply-templates />
</xsl:template>
```

The XSL code is a valid template that without the HTML commands would present the table header as plain text. The template body consists of a “for-each” statement that iterates the names of the columns (intuitively referred to as “collection/*[1]/*”) and an “apply-templates” command that applies any matching templates in the stylesheet at that point (in this case, the “collection” element is executed, emitting the rest of the rows).

The names of the columns are extracted from an XML document such as:

```
<?xml version="1.0"?>
<collection>
<GENE>
  <NAME>PDE1</NAME>
  <DESCRIPTION>PHOSPHODIESTERASE</DESCRIPTION>
  <PATHWAY>PURINE METABOLISM</PATHWAY>
</GENE>
<GENE>
  <NAME>PTR2</NAME>
  <DESCRIPTION>SMALL PEPTIDE PERMEASE</DESCRIPTION>
  <PATHWAY>TRANSPORT</PATHWAY>
</GENE>
</collection>
```

The “collection/*[1]/*” pattern consists of three parts. The entire path refers to an element at the third level of depth in the element tree. Each part refers to an element (or elements) and the slash "/" means to match the next part against the children of the already selected element (or elements). The three parts are:

1. Match the element type “collection”.
2. Match the first child of that element, denoted by “/*[1]”. Literally, of all the children of that element, choose the one that is first in sibling order.
3. Match all children of that element, denoted by “/*”.

Thus, the first step matches:

```
<collection>
 ...
</collection>
```

The second step matches:

```
<GENE>
  <NAME>PDE1</NAME>
  <DESCRIPTION>PHOSPHODIESTERASE</DESCRIPTION>
  <PATHWAY>PURINE METABOLISM</PATHWAY>
</GENE>
```

The third step matches the three elements:

```
  <NAME>...</NAME>
  <DESCRIPTION>...</DESCRIPTION>
  <PATHWAY>...</PATHWAY>
```

Those three elements are iterated over by the “for-each” command and each element type name is extracted using the “node-name” command resulting in the column headings: “Name”, “Description”, and “Pathway”.

The second template in the stylesheet is actually a little simpler. It is:

```
<xsl:template match="collection">
  <xsl:for-each select="*">
    <TR>
      <xsl:for-each select="*">
        <TD>
          <xsl:value-of select="." />
        </TD>
      </xsl:for-each>
    </TR>
  </xsl:for-each>
</xsl:template>
```

The HTML code emits a single row for each element in the collection:

```
For each element in the collection,
   <TR>
      For each subelement (i.e., column element),
        <TD>
          Value of the character data in the column goes here.
        </TD>
    </TR>
```

The XSL code is two nested “for” loops:

```
<xsl:template match="collection">
  <xsl:for-each select="*">
     <xsl:for-each select="*">
        <xsl:value-of select="." />
     </xsl:for-each>
  </xsl:for-each>
</xsl:template>
```

The outer “for-each” loop matches all of the elements in the collection, denoted by “*” in the select attribute. For the gene collection example, the loop matches each of the “gene” elements, including the one already used by the root template to extract the column names:

```
<GENE>
  ...
</GENE>
<GENE>
  ...
</GENE>
<GENE>
  ...
</GENE>
```

The inner “for-each” loop matches all of the elements within the “gene” element, for example:

```
<NAME>PDE1</NAME>
<DESCRIPTION>PHOSPHODIESTERASE</DESCRIPTION>
<PATHWAY>PURINE METABOLISM</PATHWAY>
```

The “<xsl:value-of select="." />” command emits the text value of the current node, in effect, the character data between the start and end tags. The current node is denoted by the ".". In this example, the values that are matched in the loop are: “PDE1”, “PHOSPHODIESTERASE”, and “PURINE METABOLISM”.

The entire stylesheet in DXD Example 7-2, when processed with the example data in DXD Example 7-3, results in the HTML given in DXD Example 7-4.

1.3.3.2 Rendering XML Fragment as a Record

In addition to the table-oriented view of the data, it can be useful to present a record-oriented view when the collection of data contains only one record, or when the user wishes to examine a row of the table in more depth. A record-oriented presentation is especially useful if the table presentation is significantly wider than what the browser can display.

An XSL fragment to create the record presented in DXD Figure 7-2 is given in DXD Example 7-5. To determine whether a collection has more than one element in the IE5 version of XSL, a test is done to see if a second child exists “<xsl:when test="*[1]">”. The children are numbered beginning with 0, so the second child is at position 1. The XSLT version has a function, called count, to count the number of children, and XSLT begins number children with "1". The XSLT version of the fragment in DXD Example 7-5 is given in DXD Example 7-6.

1.3.3.3 Rendering Identifiers and Proxies as Hypertext Links

When records in a database have a unique identifier, that identifier may be presented to the user as part of the table to allow for selection and display of only the data in that record. In the stylesheet of DXD Example 7-7, the identifier is assumed to be an "ID" attribute of the corresponding element and is presented as the first column in the table. The stylesheet can be modified to create a hypertext link for the identifier that retrieves the complete record from the database, such as is shown in DXD Figure 7-3. The XSLT version of the stylesheet is in DXD Example 7-8.

The hypertext link to extract a record for a unique identifier is a special case of using drill-down. *Drill-down* is a mechanism to examine some part of a collection of data in more detail. Drill-down is useful when more data is available than can be presented in a two-dimensional table.

When a document has many levels of embedded elements, it may be wasteful to send all the levels to the user interface, especially if the display presents only the topmost element. A *proxy* element may be used, which substitutes for an XML fragment and contains enough information to retrieve that fragment if needed. For a relational database, the name of the table and a primary key value are sufficient. For an XML database, a unique fragment identifier is used.

A proxy may be expanded by an application while rendering the data in the user interface by retrieving the appropriate fragment as needed. An HTML browser provides a easy mechanism through hypertext links where the user can drill-down into the information contained by the proxies. For example, the XML element

```
<proxy tablename="exper_result" id="1024"/>
```

may be transformed using the following XSL stylesheet template into a hypertext link:

```
<xsl:template match="proxy">
     <xsl:element name="a">
          <xsl:attribute
name="href">http://127.0.0.1/org.xwdb.xmldb.rserve.XMLServlet?t
ablename=<xsl:value-of select
="@tablename"/>&amp;id=<xsl:value-of
select="@id"/>&amp;stylesheet=http://127.0.0.1/ss/generic.xsl
          </xsl:attribute>
          <xsl:value-of select="@id"/>
     </xsl:element>
  </xsl:template>
```

The template creates an HTML element with element type name "a" and one attribute "href" to create a hypertext link with text value being the value of the "ID" attribute of the proxy. The HTML has the sections for the base URL, tablename, ID, and stylesheet arguments. The "&" encoding is translated to the "&" character and the resulting HTML (with line breaks added to be more readable) looks like this:

```
<a href="http://127.0.0.1/org.xwdb.xmldb.rserve.XMLServlet?
tablename=_________&id=________________&
stylesheet=http://127.0.0.1/ss/generic.xsl">
id goes here as text
</a>
```

The HTML results in hypertext links being displayed in the web browser.

The XSL stylesheet in DXD Example 7-9 emits an HTML table with proxies and drill-down.

1.3.3.4 Varying Presentation Based on Content

The stylesheet in DXD Example 7-10 provides a record presentation when there is exactly one record in the collection and a tabular presentation when more than one record is in the collection. It also supports the use of proxies.

One of the advantages of using XSL stylesheets with the XML is that customized presentations may be used in addition to the generic rendering. The information in an element may be sufficient to render the data in a graphical form, to plot the data, or to augment the data with additional queries or functionality.

One way to augment the XML is to create an alternative rendering for elements with a specific tag. For example, the presentation of the individual record in the generic stylesheet in DXD Figure 7-2 may be appended with choices based on the element type name of the element to create HTML forms that provide additional functionality via CGI programs. For example, the following XSL fragment can be used to add a comment to an experimental result:

```
<xsl:choose>
  <xsl:when test="EXPER_RESULT">
  <form action="/cgi-bin/add_comment" method="POST">
    <hidden name="tablename" value="EXPER_RESULT" />
    <xsl:element name="hidden">
      <xsl:attribute name="name">id</xsl:attribute>
      <xsl:attribute name="value">
         <xsl:value-of select="EXPER_RESULT/@id"/>
       </xsl:attribute>
    </xsl:element>
    <textarea name="comment" rows="5" cols="60">
    </textarea>
    <p>
    <input type="submit" name="Submit" value="Add comment to
      Experimental Result"/>
    </p>
  </form>
  </xsl:when>
</xsl:choose>
```

The resulting display is shown in DXD Figure 7-4.

Additional information can be added to the display based on information in the database. In addition to CGI scripts, additional information from the schema may be included. For example, the drill-downs based on a foreign key constraint in a relational database provide a link to a detail table, which provides additional information on the value in the current table. Foreign key constraint can be followed in the opposite direction to provide information on tables that refer to the current entry. For example, an employee database in a company may have many tables supporting a variety of applications that refer to the employee ID as a foreign key constraint. The "drill-up" information can be used to examine the applications in which a employee is participating, even if the participating application was developed after the current one was implemented. In a biology database, the gene table may be referred to by several other components of the database and browsing the "drill-up" information can illuminate previously unknown relationships.

1.3.4 Commands

The following arguments can be passed to RServe. They may be passed as servlet arguments, taglib fields, or calls to Java method dispatchArg.

name	*value*
tablename	name of relational table to query
id	id value
acct	database connect string to use
depth	maximum depth in the tree to render (each level requires additional db access)
elementname	name of record element (default is name of table)
xmlfragement	returns fragment instead of document if set to any value, such as "1"
stylesheet	embed in document as stylesheet
reportargs	list of columns that use used to generate the report (comma-delimited)
orderby	order of columns (subelements) in document
constraintstr	use as where clause of SQL String
sqlstring	use the SQL String
all	other args are used to build where clause of SQL string

submit	ignored (makes using HTML forms easier)

1.3.5 Constraints

To make use of the SQL operators while being compatible with the HTTP GET/POST syntax, the constraints passed to RServe may have operations embedded in them.

The following operations are supported at the beginning of the constraint string.

- Comparison operators: <,>,<=,>=,!=
- LIKE operator: Either "%" or "*" may be used as a wildcard. In addition, "%" or "*" may appear at the end of the constraint string.

Because the HTTP and taglib APIs take the parameters as Strings and the DBMS distinguishes Character data from Numeric data, it is sometimes necessary to convert between Strings and Numeric data. Oracle DBMS automatically converts Character data and Numeric data as needed, but IBM DB2 does not do any conversion and does not allow for a number to be quoted, so the following operations may be necessary:

- Quote operator: "'" (single quote) ensures that the constraint will be quoted when passed to the DBMS as part of the SQL string.
- Evaluate operator: "@" that the constraint will not be quoted when passed to the DBMS as part of the SQL string.

Rather than require the developer to distinguish every conversion the following default type conversion rules are utilized by RServe (unless overridden by the previous operators):

- If the value being passed is a number, then do not quote it.
- If the value being passed is a string, then quote it.

In practice, the only time the quote and evaluate operations are needed is:

- If passing in a function, such as sysdate() or substr, use the evaluation operator.
- If a DB2 field is Character data, but a number is a valid entry, then use the Quote operator.

1.4 Examples

1.4.1 Gene Query

The following query lists all the genes in a “GENE” table with species “Homo sapiens” and orders the result by “name”.

```
<xmldb:simplecmd name="RDB">
  <xmldb:field name="tablename">GENE</xmldb:field>
  <xmldb:field name="species">Homo sapiens</xmldb:field>
  <xmldb:field name="orderby">NAME</xmldb:field>
</xmldb:simplecmd>
```

1.4.2 Oracle Test Query

The following query works with the default database supplied by Oracle.

```
<xmldb:simplecmd name="RDB">
   <xmldb:field name="acct"
    >jdbc:oracle:thin:scott/tiger@127.0.0.1:1521:ORCL</xmldb:fi
eld>
  <xmldb:field name="tablename">EMP</xmldb:field>
  <xmldb:field name="ename">ALLEN</xmldb:field>
</xmldb:simplecmd>
```

The following query retrieves a list of all tables in the database from the Oracle system table.

```
<xmldb:simplecmd name="RDB">
  <xmldb:field name="tablename">syscat.columns</xmldb:field>
  <xmldb:field
name="reportargs">tabschema,tabname,colname,colno,typename,length
</xmldb:field>
  <xmldb:field name="tabschema"><%=schemaName%></xmldb:field>
  <xmldb:field name="tabname"><%=tabName%></xmldb:field>
  <xmldb:field name="orderby">colno</xmldb:field>
</xmldb:simplecmd>
```

1.4.3 DB2 Test Query

The following query retrieves a list of all tables in the database from the DB2 system table.

```
  <xmldb:simplecmd name="RDB">
    <xmldb:field name="tablename">syscat.tables</xmldb:field>
    <xmldb:field
name="reportargs">tabschema,tabname</xmldb:field>
    <xmldb:field name="orderby">tabschema,tabname</xmldb:field>
  </xmldb:simplecmd>
```

2 XML Data Server (XServe)

An *XML data server* combines an XML DBMS with a web server. Users and applications can access data in the DBMS via the XML data server, using a web interface. There are three options for storage using the XML DBMS, fine-grained, coarse-grained, or medium-grained.

2.1 Data Server

XServe is an XML data server, which stores, edits, and queries an XML database with data stored in a relational database using the relational storage. This data server specifies an API via URLs similar to the relational data server of the previous section.

2.1.1 Background

In addition to the operations on documents that were defined in DXD Chapter 3, the XML data server provides similar operations on document fragments corresponding to each element in the document. Thus, it is possible to retrieve, delete, and update an element in a document.

The xServe XML data server described in this section demonstrates some of the functionality that could be provided by a web-accessible XML database system. Although a fairly simple prototype, it may be used to store and retrieve XML documents or to capture hierarchically organized data. The system also demonstrates how to develop an XML database system and some of the functionality that may be necessary in a production environment.

xServe provides the following commands:

- Store a document given its URL.
- Retrieve a document or a document fragment from the database.
- Retrieve any document as a fragment or any fragment as a document. Thus, new documents may be built from other documents and fragments.
- Update a document by replacing a document fragment with a document (or fragment) given a URL (including a URL for xServe).
- Append an XML fragment at a URL reference to a specified fragment.
- Delete a document or document fragment from the database.

Other commands that might be useful, but which are not demonstrated, are to reorder the elements in a document, to search for a document given a constraint, or to perform any other operation described in DXD Chapter 3.

A home page for xServe is shown in DXD Figure 5-7, and its HTML source is listed in DXD Example 5-14. The home page uses HTML forms to create URLs that access the XML database. Some example URLs are

- *http://127.0.0.1/servlet/org.xwdb.xmldb.xserve.XMLServlet?list=1*
- *http://127.0.0.1/servlet/org.xwdb.xmldb.xserve.XMLServlet?store=http://127.0.0.1/temp1.xml*
- *http://127.0.0.1/servlet/org.xwdb.xmldb.xserve.XMLServlet?store=D:\Temp\cancer.xml*
- *http://127.0.0.1/servlet/org.xwdb.xmldb.xserve.XMLServlet?getfragdoc=7.1*
- *http://127.0.0.1/servlet/org.xwdb.xmldb.xserve.XMLServlet?appendfrag=7.1+ http://127.0.0.1/servlet/org.xwdb.xmldb.xserve.XMLServlet?retrieve=4*
- *http://127.0.0.1/servlet/org.xwdb.xmldb.xserve.XMLServlet?deletefrag=1.2*

2.1.2 Implementation

Java code that implements the xServe relational data server is given in DXD Example 5-15. A UML class diagram is given in DXD Figure 5-8.

The main class in xServe, called Demo, is a subclass of the Demo class of the fine-grained relational storage system described in DXD Section 4.2.4. The xServe Demo class contains fields for each class that implement a command of the data server and executes the command through the “dispatchArg” method. Each command is a subclass of the Command abstract class, which contains a RDB object to connect to the database and an Output object to write the formatted data to the appropriate stream. RDB and Output are described in Appendix A. xServe commands retrieve, append, and replace fragments as well as retrieve, store, and delete documents.

A SAX parser handler is used to append and replace fragments. SAX parsers are described in Appendix B. The abstract class JDBCFragLoadHandler retrieves maximum index values for child, embedded element, and attribute records so the inserts will create appropriate unique identifiers. JDBCReplaceFragHandler uses the DeleteFrag command to delete the old element and then inserts the new XML fragment in its place. JDBCAppendFrag inserts the XML fragment after the last embedded element. Both commands insert data in a manner similar to the load

document command JDBCLoadHandler in the fine-grained relational storage system.

2.1.3 Commands

The following arguments can be passed to XServe. They may be passed as servlet arguments, taglib fields, or calls to Java method dispatchArg.

XServe has four commands:

- store – store a document in the database or replace a fragment in a document
- retrieve – retrieve a document from the database
- append – append an XML element or fragment as a child to an existing document or fragment
- delete – delete an XML document from the database or a fragment from a document in the database

The following arguments may be used with any XServe command to identify the document or fragment being used (except for store, which does not recognize the "doc" argument and delete, which does not allow the "argument" to be used):

name	*value*
doc	id of a document in the database
frag	fragment id "doc.element" in the database
name	name of a document in the database

Only one of "doc", "frag", or "name" arguments should be used.

2.1.3.1 Store Command

The following arguments may be used with the "Store" command:

name	*value*
text	text of XML to store in the database
url	url of XML to store in the database
texttag	element type name to wrap the value of "text" parameter. This is useful in passing in a string as "text" and having it stored as an element.

addslice	element type names to store using medium-grained storage

For the store command only one of "doc" or "name" arguments should be used; it is not possible to store at a specific fragment. Only one of "text" or "url" should be used. The "texttag" argument can only be used with the "text" command.

2.1.3.2 *Retrieve Command*

The following arguments may be used with the "Retrieve" command:

name	*value*
head	if value is "1", the document includes the ?xml processing header at the top of the file with version number "1.0". Having this option allows XML fragments to be retrieved as an XML document, and an XML document to be retrieved as an XML fragment.
stylesheet	href of the XSL stylesheet with which to render the document or fragment
includefragid	if included with any value, such as "1", an attribute named "FRAGID" is included for every element in the resulting document, and the element value is the unique fragment id for that element in the database. This value may be passed to the retrieve command to retrieve only this fragment. Note: in medium-grained storage, the elements in slices that are not parsed will not have fragment identifiers.

2.1.3.3 Append Command

In the "Append" command, the XML data is added as the last child of the fragment specified. If a fragment identifier is not specified, then the top-level element in the document is appended.

The following arguments may be used with the "Append" command:

name	*value*
text	text of XML to store in the database
url	url of XML to store in the database
texttag	element type name to wrap the value of "text" parameter. This is useful in passing in a string as "text" and having it stored as an

	element.

Only one of "text" or "url" should be used. The "texttag" argument can only be used with the "text" command.

2.1.3.4 Delete Command

The "Delete" command does not take any additional arguments.

2.2 Fine-Grained Relational Storage

In the fine-grained approach, every construct in the document is given a unique identity in the relational database. Every element, attribute, and character data region can be individually accessed, modified, or deleted with minimal effect on other document constructs. This provides the most flexibility and ease of access both with the XML DBMS–specific operations as well as through the traditional relational ones. However, regenerating the entire document can be time-consuming when large, and the space required for all the pointers can be significant.

2.2.1 Logical Design

The basic idea of the fine-grained approach is that each type of construct in the document has its own table: elements, attributes, and character data regions. There is also a table for documents and a table for the parent/child relationship between elements and their constituents (subelements or character data). A logical schema in the relational data model is depicted later in this section.

The logical design of the schema consists of two steps. The first step is to create a relational schema in the relational data model. The second step is to normalize that schema. Most database developers will develop a somewhat normalized schema from the beginning, but in this section we start with a schema that is not normalized to first normal form. This simplifies the presentation to those without a lot of relational database design experience and make the section more useful in developing an object-oriented implementation, as described in DXD Section 4.1.2

A logical relational schema is defined in terms of the relation names, column names, and domains. Relation names and column names are familiar to those who have used any relational DBMSs. Domain names are an integral part of the relational data model but are not implemented in a traditional relational DBMS. Instead, the simple data types—such as NUMBER, STRING, and TEXT—are used, often with some variations. Those data types are *pre-defined domains* in the

relational data model. Some relational DBMSs with an object-oriented component do implement additional *user-defined* domains to incorporate objects. The user-defined domains may be *atomic* within the schema or refer to relations in the schema (*relation domains*). The relation domain is typically implemented in a DBMS as a foreign key relationship. However, it helps to explicitly define the relational schema by using domains to make explicit the relationship between the XML data models of the previous chapter and the XML storage implementations of the following sections.

The logical schema is specified in the following format:

```
RelationName( columnName1 DOMAIN_NAME_1 , columnName2
      DOMAIN_NAME_2 )
```

A logical schema in the relational data model for the fine-grained schema is

```
Document(name DOC_NAME, root ELEMENT)
Element(doc DOCUMENT, parent ELEMENT, tag ELE_NAME)
Attribute(doc DOCUMENT, element ELEMENT, name ATTR_NAME, value
      ATTR_VALUE)
CharData(doc DOCUMENT, element ELEMENT, value CDATA)
Child(doc DOCUMENT, element ELEMENT, index NUMBER, child_class
      CHILD_CLASS, child CHILD_NODE)
```

The "CHILD_CLASS" and "CHILD_NODE" domains are used to simulate the union of the "ELEMENT" and "CHARDATA" domains.

To normalize the schema into first normal form, the relation domains “DOCUMENT” and “ELEMENT” are eliminated by creating unique identifiers for each relation "doc_id" and "ele_id" in the new atomic user-defined domains “DOC_ID” and “ELE_ID”. A normalized logical schema is:

```
Document(doc_id DOC_ID, name DOC_NAME, root ELEMENT_ID)
Element(doc_id DOC_ID, ele_id ELE_ID, parent_id ELEMENT, tag
      ELE_NAME)
Attribute(doc_id DOC_ID, ele_id ELE_ID, name ATTR_NAME, value
      ATTR_VALUE)
CharData(doc_id DOC_ID, cdata_id CDATA_ID, ele_id ELE_ID, value
      CDATA)
Child(doc_id DOC_ID, ele_id ELE_ID, index NUMBER, child_class
      CHILD_CLASS, child CHILD_NODE)
```

DXD Figure 4-4 shows a diagram of the normalized logical schema, and DXD Figure 4-5 shows the schema with atomic data types instead of domains.

In the logical schema, the document table contains columns for the name, document ID, and element root. Each entry in the table corresponds to a document in the database. The name provides a unique name for the document for reference by external applications. The document ID provides a unique number identifier that is referenced by the other tables to associate each document construct with

the appropriate document. This is necessary for rapid retrieval of the document when performing searches on the document constituents. The element root connects the document table with the element in the element table that is the root of the document element tree.

The element table is central to the design and connects the remaining document constructs. Each element in a document has a unique identifier. The element identifier together with the document identifier serve to uniquely identify the element. Having an element identifier unique to each document simplifies the process of searching and copying documents within the database. The element table also contains columns for the element type (tag) name, the document identifier, and the parent element identifier. The tag name column captures the string value of the element type, and the document identifier refers to the document of which the element is a part.

Every element must be contained in a document. If this constraint were not present, the database could more easily become corrupted should multiple applications simultaneously create elements without documents. The parent element identifier connects the subelements to the parent elements in a tree structure. It provides a method for navigating from the leaves and internal nodes of the tree to the root. Every element in the document should have a parent element except for the root element (which is denoted by a NULL parent identifier). The root element should also be referenced by the root column of the document table.

The attribute table is fairly straightforward. It contains the name and value of the attribute and a reference to the element identifier for which it is an attribute. The element identifier connects the element to its set of attributes. In addition, the attribute has its own unique identifier and has a reference to the document identifier. The attribute identifier is used as an internal reference by database operations that access the attribute directly, such as query, update, and delete.

The child table connects the elements in the element table to their constituents through the element identifier and child identifier columns. It provides a straightforward way to access all the subelements and character data regions that are direct descendants of an element as well as to navigate up the element tree. The child table also provides an ordering of the element constituents with an index column. The index orders both the subelements and character data regions in the one ordering. There is a column called child class that distinguishes between constituent types element and character data. The child table contains a reference to the document identifier, but does not need a unique identifier. The primary key of the child table is the document identifier, element identifier, and index number—this is sufficient to uniquely identify each child relationship in the database.

The child class may also be used to extend the current schema to incorporate other XML constituents, such as comments. This approach may also be used to incorporate data type information from the XML document if that is specified as part of a document type definition (DTD) or XSDL schema. For example, another table may be created for the data type NUMBER, DATE, or user-defined data types.

In some cases more efficient queries may be performed by duplicating the index column of the child table in the Element and CharData tables. For example, this would allow a query of all subelements for a specified element to retrieve only from the Element table without needing to perform a join with the Child table. However, duplicating the index column would require that the application ensure that the index values are kept synchronized or require the use of additional code in the database to keep the values synchronized, such as triggers, SQLJ methods, or PL/SQL procedures. It is also possible to drop the Child table and use only the index column in the Element and CharData tables, but then queries that retrieve based on index number would have to attempt querying from both Element and CharData tables as the data type of the child at a specified index is not explicitly stored. This would probably be okay with only two types of children (Element and CharData), but would not scale-up well if other child types are added. A reasonable compromise for some applications would be to duplicate the index column for the Element table only, which would allow queries of element type names, such as simple XML Path queries, to be executed more efficiently.

2.2.2 Physical Design

The first step in creating a physical schema from the logical schema consists of determining reasonable sizes for the columns. This is best done knowing the particulars of the applications, but reasonable assumptions are made here for a moderate-sized application. Values are chosen to allow for 100 million documents with 100 million elements each, and each element can contain up to 1 million children. Tag and attribute names are limited to 32 characters, and attribute values are limited to 255 characters. The size of the document name was chosen to be 128 characters to allow for a flat collection of named documents, to include path information, or to store URLs.

A physical schema for Oracle is presented in DXD Figure 4-6 and the code that creates it is listed in DXD Example 4-3. When developing a physical schema for Oracle, be aware of the limitation of string sizes to 2000 characters (Oracle 7) or 4000 characters (Oracle 8). It is reasonably likely in many applications that character data regions may be longer than several hundred characters and thus must be stored separately. Oracle provides the LONG data type that provides relatively unlimited string length (2 GB) but very limited access through

relational operations. To address this, it is necessary to modify the schema to provide for both storage mechanisms.

Two approaches to allowing for character and long data types are: (1) to create an additional column; or (2) to create an additional table. To create an additional column, the value column in the character data table is replaced with three columns: a VARCHAR2 value column, a LONG value column, and a boolean flag to distinguish which one is filled (the other is null). The Oracle DECODE function can be used to retrieve the data somewhat transparently. To create an additional table, the character data table is replaced with two tables: one where the value is of type VARCHAR2 and one where the value is of type LONG. A flag to distinguish between the two is added to the table(s) that access the character data. In general, adding a column is the best approach; however, because the element child table already distinguishes between constituents of type subelement and character data, the schema becomes cleaner by splitting the character data table and replacing the child class “Character Data” with “String” and “Long”. The character data length to split between the two tables becomes a tunable schema parameter. A somewhat arbitrary length of 255 was chosen, which does allow for the creation of an index on the String value column.

A physical schema for other DBMSs can be created similarly. A physical schema for DB2 is shown in DXD Figure 4-7. The code to generate the schema is given in DXD Example 4-4. The differences between the Oracle schema and DB2 schema are as follows:

- For strings of varying length, Oracle uses the data type name VARCHAR2 and DB2 uses the name VARCHAR.
- The Oracle schema uses NUMBER(8) for identifiers, and the DB2 schema uses INTEGER, which allows for 2^{32} (about 2 billion) identifiers (in DB2 v7.0).
- DB2 only allows for 18 characters in the length of a constraint name, so the foreign key constraint names "fk_xdb_child_doc_id" and "fk_xdb_child_ele_id" were shortened to"fk_xdb_child_doc" and "fk_xdb_child_ele", respectively.
- DB2 does not like the comment in the xdb_child table creation statement, and it is omitted.
- Oracle provides a LONG type for very long character strings up to 2GB. DB2 provides a CLOB (Character Large OBject) type that provides similar functionality, also up to 2GB. However, the DB2 CLOB allows specification of a lower maximum size that is more efficient.

For DB2, a size of 100KB is chosen for the maximum character data region—the maximum length of text without a tag. This is probably enough for most documents, however, some applications may quote a large text region in a CDATA section, which could require an increase in the maximum character data region size.

Additional physical design includes creating indices and tuning the DBMS. Indexes are useful on most of the columns, but indices are best developed within the requirements of the particular application. There are books available on creating appropriate indexes and tuning specific DBMSs, so the remainder of physical design is outside the scope of this book.

2.2.3 Examples

To demonstrate the functionality of the fine-grained approach, a simple system is developed that supports the operations of the Generic Data Model described in DXD Section 3.5.7. To demonstrate the system, some of the tasks from DXD Section 3.4.2 are implemented. Several of the tasks may be performed through SQL statements that are executed against the Oracle database, but other tasks—such as storing and retrieving an entire document—must be developed in a traditional programming language.

A Java system is developed (and described in the next section) that provides storage and retrieval for XML documents captured in the fine-grained relational schema. Although missing much functionality necessary for a complete DBMS, the system allows XML documents to be stored and retrieved in a manner that allows for searching and updating. It also demonstrates the basic operations of an XML DBMS in a form that is comprehendible in a single example and can serve as the basis for more complex systems.

To use the system, a command line application is presented here. The command line interface provides options to store a document, to retrieve by identifier, to list all documents, or to delete by identifier; the retrieved XML documents are written to standard out. Java GUI applications or applets are also possible, and an applet is shown in DXD Figure 4-8—where a user may store an XML document by pasting a URL and selecting the “Store” button, and documents may be retrieved by their internal identifier or by selecting from a list of all documents.

After a document is stored in the database, SQL queries may be asked to search for elements or attributes that meet certain constraints, as shown in DXD Example 4-5. Some examples of use are shown in DXD Example 4-6. These examples demonstrate some of the functionality of the system.

2.2.4 Implementation

The Java code that implements the store and retrieve is listed in DXD Example 4-7. Note that the CLASSPATH must contain a JDBC driver and an XML parser (IBM's XML parser is used in the example). A UML diagram for the Java implementation is given in DXD Figure 4-9.

The remainder of the section describes the Java code for the readers who wish to better understand the implementation.

The main class for the storage system is "Demo". Each command is implemented by a class, and the "Demo" class keeps an instance of the command classes. "Demo" also keeps an "RDB" class that refers to a connection to the relational database used for storage. The "RDB" class allows for connection and querying of a relational database through a JDBC connection and is described more fully in Appendix A. "Demo" also provides a mechanism for writing the SQL onto standard out for testing or use within a script to access a relational database that does not have a JDBC connection. The "main" method in "Demo" reads the commands from the command line and dispatches them to the appropriate command instance.

The command classes are "Format", "Delete", and "Load". They each access the database through the "RDB" interface and write status messages to the standard error stream.

The primary method for a "Format" instance is the "writeDoc" method that takes a document identifier as its argument and begins to retrieve the data for it from the relational database. It calls "writeElement" on the document element, which then writes out the element, retrieving information as necessary from the relational database. The method "writeElement" writes the element type name and the attributes, then accesses the "xdb_child" table to retrieve a list of children. The method then calls "writeElement" recursively or "writeCdata" as appropriate. The "writeElement" method retrieves all children from the database before writing any child and uses the "Child" class as a helper object to store the vector of children. All output uses the "writeln" method.

The "Delete" command class has a primary method called "deleteDoc" that takes a document identifier as an argument and deletes all items in all tables with that identifier as the "document_id". The method ensures that foreign key constraints are not broken by deleting from tables in the proper order and by explicitly breaking the circular constraint between the document element entry in the "xdb_element" table and the document entry in the "xdb_document" table. A SQL delete cascade could have been used to simplify the implementation of the delete command.

The “Load” command uses a SAX parser. The parser is described in more detail in Appendix B. The SAX parser uses an instance of the “LoadHandler” class to load each document. The “startElement” method in class “LoadHandler” creates a database entry for the element and each attribute. The counters for the unique identifiers of the element, attributes, and character regions are tracked within the instance as they depend upon the document. An entry in the “xdb_child” table is created between the current element and its parent (as tracked by an instance of the “Element” helper class in the “currentElement” instance variable). The “characters” method of “LoadHandler” creates an entry for the character date region in the “xdb_str” or “xdb_text” table as determined by the “CDATA_SPLIT_LENGTH” class variable. The method also creates an entry in the “xdb_child” table using information in the “currentElement” instance variable. The recursive calls of “startElement” by the SAX parser are handled by keeping a stack of “Element” helper objects with the pertinent information. All output goes through the “out” method of “LoadHandler” that defaults to standard out. For JDBC connections, a subclass of “LoadHandler” is created called “JDBCLoadHandler” that redefines the “out” method to use a “RDB” method and defines that the document ID is obtained from a “RDB” query to add one to the current maximum document identifier.

2.3 Coarse-Grained Relational Storage

Another approach to storing the documents in the database is to store them in their entirety. Although similar to storing documents in flat files, this approach has the advantage of allowing the documents to be referred to within other structures in the database. It also provides the security, recovery, and other features of the DBMS in which it is stored. Depending upon the DBMS, there may be built-in operations designed to work on large text documents, which may be useful.

Although some early vendors propose this approach, its primary usefulness is part of a hybrid representation as discussed in the next section. However, it is worth examining independently because it demonstrates a very simple mechanism to capture XML in an existing DBMS. This can be a important step in a phased deployment plan where a flat file database is created, then moved with little modification into a relational database using this coarse-grained approach, then refined making use of the capabilities of the relational DBMS in a fine-grained or medium-grained approach.

A very simple conceptual schema and logical schema is developed here. A document is defined to have a name and a body. The optional “name” provides a unique identifier by which the user may refer to the document. The logical schema consists of one table called document with three columns: a name, a unique numeric identifier, and a body. A unique identifier is added to the schema

at this stage to facilitate its integration into other parts of the system and is not strictly needed.

A logical schema for the coarse-grained storage approach is very simple:

```
Document (name STRING, body TEXT)
```

A physical schema for Oracle is presented in DXD Figure 4-10 and SQL code to implement it is given in DXD Example 4-8. The code to implement the database in DB2 is given in DXD Example 4-9.

2.4 Medium-Grained Relational Storage

The disadvantage of the fine-granularity approach is that storing and reconstructing a document is very expensive; its advantage is that some queries and modifications are very simple. Evaluating the approaches against the tasks from DXD Section 3.4.2, the fine-grained approach works well to perform tasks that access elements (Tasks c–i), whereas the tasks to store and retrieve an entire document are difficult (Tasks a, b). The coarse-grained approach works well in manipulating entire documents (Tasks a, b) and has difficulty with the element manipulations (Tasks c–i). Task j was to manipulate one element in the context of another, and Task k was to join two elements, which are difficult in either solution.

Another storage approach is to create a medium-granularity approach—a compromise between the fine-granularity and coarse-granularity approaches. The document tree can be sliced into sections where the sub-sections are stored with a coarse-grained approach. This is particularly useful if the sections are accessed individually: for example in reference books such as dictionaries or encyclopedias, a medium-grained approach would be to store each entry separately. Consider the document in DXD Example 4-10. It might be useful to break up the document into 1201 sections: 1 section for the top-level document and 1200 sections for each of the dictionary entries. The medium-grained approach performs better than the fine-grained approach on storing and retrieving documents, and still performs okay on the element manipulating tasks. Task j—to manipulate an element only within the context of another—is still difficult.

2.4.1 Slice Points

Determining appropriate slice points is a complex issue.

- How many slice points are created?
- How many levels of slicing are created?

- Does the slicing depend upon the element type name or the depth in the tree or the size of the document section?
- Are some sections of the document sliced more finely than other sections?

The fine-grained approach and the coarse-grained approach are actually opposite poles of the medium-grained approach. The fine-grained approach can be described as creating a slice at every tag, and the coarse-grained approach can be described as creating zero slices per document.

One way to approach the slicing granularity is to view slicing as a method to index the database. An index speeds up access for a particular database request by creating an index table that provides quick navigation to the indexed information. Slices can be created on a element type name (or names) for which frequent access is anticipated. A combination of element type names and attribute values can also be used to drive the index slice method. Choosing slice points based on desired indexing will reduce the data access time over the coarse-grained approach for queries or other accesses that involve the indexed tags. Indexes can be created on a few highly requested element type names or on the majority of element type names for which access is anticipated. If most of the directly accessed element type names are indexed, then the access time for those queries approaches the access time under the fine-grained approach while also reducing the document regeneration time because other elements do not need to be regenerated unnecessarily.

Another way to approach the slicing granularity is to view the slicing method as a buffering mechanism. Slices may be determined by physical characteristics, such as size. The slice size can be chosen to efficiently use network communication protocols to reduce the time needed to retrieve a portion of the document when the network response is a critical factor. For example, if the query process or application is on a separate machine or processor from the storage device, and the communication takes place in 8K buffers, then overall efficiency may be improved by sending over portions of the document in slightly less than 8K sections.

Combinations of approaches may also be used for particular applications or documents. For example, a reference book such as the one presented in DXD Example 4-10 essentially combines an index on the dictionary entry with slices of approximately the same size.

One issue that needs to be addressed in this approach is how to represent the slice points in the document. One mechanism is to create a specific element type to represent the necessary information, ensuring that the element type name is unique in the document, possibly by creating it in a new namespace using XML Namespaces. For example, an element type called “slice” or “proxy” could be

created with attributes that contain sufficient information to reconstruct the document, namely “document_id” and “element_id”. The dictionary document of DXD Example 4-10 is shown in DXD Table 4-1, which illustrates that mechanism.

2.4.2 Database Design

The medium-grained schema is created by adding two relations to the fine-grained schema. The first relation is somewhat similar to the document relation of the coarse-grained schema: It contains identifiers and a body that consists of marked-up text. The second relation serves as an index of what elements are included in which fragment. Although not strictly necessary, having this XML index greatly simplifies performance on some operations. (Note: this relation defines one index in the XML DBMS that is implemented as a relation in a relational DBMS. It is a part of the design of an XML DBMS unlike the relational indexes that may be added later.)

The logical schema of a medium-grained implementation is as follows:

```
Document(name DOC_NAME, root ELEMENT)
Element(doc DOCUMENT, parent ELEMENT, tag ELE_NAME)
Attribute(doc DOCUMENT, element ELEMENT, name ATTR_NAME, value
      ATTR_VALUE)
CharData(doc DOCUMENT, element ELEMENT, value CDATA)
Child(doc DOCUMENT, element ELEMENT, index NUMBER, child_class
      CHILD_CLASS, child CHILD_NODE)

Fragment(doc DOCUMENT, element ELEMENT, body TEXT)
FragmentReference(doc DOCUMENT, fragment FRAGMENT,
      element_reference ELEMENT)
```

To normalize the schema into first normal form, the relation domains are eliminated by creating unique identifiers for each relation. A normalized logical schema is as follows:

```
Document(doc_id DOCUMENT, name DOC_NAME, root ELEMENT)
Element(doc_id DOCUMENT, ele_id ELEMENT, parent_id ELEMENT, tag
      ELE_NAME)
Attribute(doc_id DOCUMENT, ele_id ELEMENT, name ATTR_NAME,
      value ATTR_VALUE)
CharData(doc_id DOCUMENT, ele_id ELEMENT, value CDATA)
Child(doc_id DOCUMENT, ele_id ELEMENT, index NUMBER,
      child_class CHILD_CLASS, child CHILD_CHILD)

Fragment(doc_id DOCUMENT, frag_id FRAGMENT, ele_id ELEMENT,
      body TEXT)
FragmentReference(doc_id DOCUMENT, frag_id FRAGMENT,
      element_reference ELEMENT)
```

The first five relations in each schema are identical to the fine-grained schema. A physical schema is given in DXD Figure 4-11, and SQL code to implement the design of the two additional tables in Oracle is given in DXD Example 4-11. The SQL for DB2 is given in DXD Example 4-12, and the physical schema is given in DXD Figure 4-12.

2.4.3 Implementation

Java code that implements storage and retrieval operations for the medium-grained approach is listed in DXD Example 4-13. Note that the CLASSPATH must contain a JDBC driver and an XML parser (IBM's XML parser is used in the example). An UML diagram for the Java implementation is given in DXD Figure 4-13.

The Java code is based on the implementation of the fine-grained approach, and the main class "Demo" is a subclass of the fine-grained "Demo" class, as are the command classes "Load", "Delete", and "Format". The "LoadHandler" class for the medium-grained approach is also a subclass of the "LoadHandler" class for the fine-grained approach, though it adds tracking of the slice points. The slice points are captured in the "Slice" class, which is basically a hashtable of element type names upon which slices are to be performed. The "Slice" class may be extended by modifying the method "sliceElement" to allow for more complex slice points, such as slice points based on attribute values or depth in the tree. The "sliceElement" method determines whether a slice point should be created in the storage of a document and is only called by the "startElement" method of "LoadHandler". The class "JDBCLoadHandler" adds the same JDBC database access functionality to "LoadHandler" as the "JDBCLoadHandler" adds to the "LoadHandler" class in the fine-grained approach.

3 Hybrid Relational/XML Server

Although the relational and XML data servers described in the previous two sections are useful on their own, a hybrid system that provides query access to either would also be useful. A hybrid relational/XML data server can transparently provide access to relational and XML data. Data can be stored in the data model (and DBMS) most appropriate for it; the hybrid data server can provide limited data integration across the data models through connections between the underlying data sources. For example, an XML document may have an embedded "access" element representing a relational query. As the XML document is retrieved, the relational query may be performed and merged into the XML data stream. Similarly, the values in a column in the relational DBMS may refer to documents or fragments in the XML DBMS, and the XML data may be merged with the relational data as it is retrieved and formatted as XML. Thus, the efficiency of relational storage may be combined with the flexibility of an XML DBMS.

3.1 Background

One way to build a hybrid relational/XML data server is with an n-tier architecture that calls two backend servers: a traditional relational server and the XML data server. The hybrid server passes URL requests (or uses other protocols) to each of the data servers and merges the documents as they are retrieved.

For demonstration, however, it is probably simpler to create a new data server in a three-tier architecture that access the relational DBMS and the XML DBMS. This simplifies the architecture, but does require that the XML and relational data servers reside within the same application. Because the xServe and RServe applications can coexist in the same application, the hybrid system—xrServe—can pass commands directly to the appropriate class. The n-tier architecture would be similar except for intervening steps on the hybrid server to format, post, and retrieve the data from the separate servers.

Additional functionality may be added within the same framework. For example, it may be useful to provide direct access to external URLs within the XML document or relational tables. Another "access" element type may provide information on accessing a resource from files or URLs (as XML).

3.2 Implementation

The Java servlet for the Gene Notebook reimplements the hybrid relational/XML data server of DXD Chapter 5 to allow more flexibility in the commands that can be defined for the server. DXD Figure 10-7 shows a UML diagram for the system, and DXD Figure 10-8 shows a UML diagram for the classes that implement database commands. Commands are defined as objects to allow them to take arguments and perform more complex processing. The inheritance hierarchy for commands is shown in DXD Figure 10-9. All commands inherit from the abstract class org.xwdb.xmldb.demo.cmd.Command. Primarily, two abstract commands are used to define the functionality of subsequent commands: "dispatchArg" and "execute". The method "dispatchArg" takes two arguments: the name of a parameter to the command and its value. The method stores the values in the appropriate instance variables of the command. For example, the “store” command dispatches the parameters "text" and "url" that contain either the text of an XML document to be stored or a URL referring to an XML document, respectively. The "execute" method is called after the parameters have been processed and performs the functionality of the command. The servlet also has a method “XMLServlet#writeEntryScreenHtml” that generates a generic home page to display a list of documents in the database and to allow simple commands to be performed, such as to retrieve, store, or delete a document as shown in DXD Figure 10-10. The code for the Java servlet is given in DXD Example 10-5.

The Java servlet also provides dynamic HTML pages to prompt the user when some required parameters are not specified in the URL. This feature is used in the Gene Notebook to allow modification of data in the Notebook without specifying a complete data entry form as part of the user interface. For example, to change the name of a gene, a link is followed that specifies the identifier associated with the element for the gene as part of a URL, but not the new name. The user is prompted by generic code in the servlet to fill in the missing information, as shown in DXD Figure 10-11.

Each fragment in the database is identified by a unique identifier consisting of the numeric document identifier and a numeric fragment identifier. The fragment identifier initially is the position of the element in the document in text order (i.e., *document order)*, though inserted elements are assigned the next higher ID number in the document, regardless of the location in document order. An identifier does not change until that element is deleted. The command to retrieve a document takes as a flag whether to include the unique fragment identifier as an additional attribute on each element. That "includefragid" option is set for the Gene Notebook retrievals, and the XSL stylesheet extracts that identifier and uses it in creating URL commands to the database to modify the XML document.

4 Application Development

4.1 Instant Applications

One goal of examining the data stream is to work toward supporting instant applications. *Instant applications* are applications that can examine and edit the data in an XML database based solely on the structure of the data. For example, if data is presented on employees in the database, an application may be generated that allows for the addition, deletion, and editing of employee information in the database, as well as querying based on characteristics such as name, years employed, or salary. The application would have no intrinsic information about the employee domain, only the structure of the data in the database and constraints on the fields. Although the development of instant applications are outside the scope of this book, two ways to use XML in these applications are in the transfer of data between database and application and in the specification of data entry and query forms.

Instant applications are particularly important because of the varying structure of XML elements in the database. Unlike the relational database where all the rows of a relation have the same structure or an object-oriented database where all the instances of a class have the same structure, all the elements in an XML document are not required to have the same structure.

For example, consider a Rolodex database of one document that contains contact information for people. All the entries will have a name; most will have a phone number, though some might have only an email address. The phone number could be a work number, home number, cell phone number, and/or pager. There might also be a fax number. Personal contacts might have a birthday, whereas business contacts would have corporate information. There might be notes about the person, and if a business-related contact, the nature of the previous business. There could be multiple physical mail addresses and email addresses, some people might have a web page, whereas others might have alternate ways of contact, such as an occasionally used phone number. If sales-related, or if the person is a client, there could be information about favorite restaurants or previously given gifts.

If the database were modeled as a single concept, it could easily have thirty characteristics. This would lead to a really ugly, non-normalized relational table with 30–50 columns or a normalized database with half a dozen tables. However, an XML document could capture the information using about thirty element type names. To create a traditional data entry editing and query application that would work with the 2^{30} possible layouts could be time-consuming if done manually.

However, an instant application would automatically create the appropriate form for each person in the database, regardless of structure.

The application could be implemented to display the form as an HTML form or a Java applet. If a Java applet is used, the applet can dynamically change the user interface based on the characteristics of the XML element. If HTML is used, then a program would need to receive the XML and generate the HTML form. The program can be implemented in Java (as a servlet or JSP) or in XSL, where the data item would be transformed using an XSL stylesheet into the HTML form. In either case, the form would post the new data to the database (as a servlet or CGI script) to execute the update statement (with appropriate validation).

The data used for these applications may be identical to the data presented in browsing the data. Consider the data in DXD Example 7-13. The data may be presented as HTML using an XSL stylesheet, such as the one described in DXD Section 7.2. In addition, an editing application could present the user with a list of names from which to select. A selected name would bring up an HTML form, such as the one shown in DXD Figure 7-7. The HTML for the form is given in DXD Example 7-14. Other data items would have similar forms with data entry fields that depend upon the element type names of the subelements.

The pseudo-code to create the form is:

1. Emit the header of the form HTML document.
2. Emit the form start tag and any hidden form elements.
3. Iterate over the subelements, and emit a text box for entry with the subelement's element type name as a text label.
4. Emit the end tag of the form.
5. Emit the end tags for the HTML document.

The same form can be used to create a new data item to be added to the document based on the existing data item.

Element hierarchies can also be supported using this scheme. Each subelement is associated with a part of the form to preserve the hierarchy. For example, the XML in DXD Example 7-15 can be mapped to the HTML form in DXD Example 7-16. Each subelement is grouped in the presentation, but the entire form is updated and the previous element is modified. A possible presentation is shown in DXD Figure 7-8. Other presentations are possible, for example, to group the subelements using HTML tables. When the form is sent to the database, the entire element could be replaced, or a more complex mapping can be used to only update the subelements when they have changed.

Specialized data entry widgets can be used for particular elements (or subelements), by mapping the element type names to the widgets. This works whether the user interface is in HTML or Java. For example, the phone number data entry widget could be a entry box with a specific number of digits, the birth date data entry box could have restrictions on the values allowed. More complex user interfaces are possible, such as slider bars or a custom user interface. The data entry and query forms may be also specified using the second-generation W3C form language Xforms. If the data restrictions provided by Xforms are used, it may be possible to update data items directly without validating the changes by a separate program.

The power of instant applications occur when a custom user interface can be automatically embedded within another custom interface. For example, if the Rolodex interface can be embedded in other applications, then anytime contact information is needed on a person, a small change to the database schema automatically results in a new user interface being available to all applications that access the personal contact database.

4.2 Java-Based Visualizations

Java-based user interfaces provide another way of interacting with XML databases. Java, or another programming language, can be used to implement editors, data entry applications, visualization tools, report generators, process management software, or other clients that interact with an XML database. Java applets or applications can be used to implement the client depending upon whether the client is to be web-delivered. Clients interact with the database using commands and XML and with the user via the user interface. The interaction process between a DBMS and client is described in DXD Chapter 5. This chapter focuses on the internal architecture and functionality of the client. This section describes applications that display XML from the database, and the next section describes the creation of applications that interact with the database.

4.2.1 Client Architecture

The simplest client architecture displays XML to the user with no editing or further DBMS interactions. The Java client must first parse the XML before displaying a user interface. There are two major types of XML parsers. The first is a tree-based parser, such as a DOM parser, which would create a tree of the XML document. After parsing is completed, the client creates a display. The second kind of parser is a SAX parser, which is an event-based parser (it creates events while parsing the document). At key locations in the document—such as start and end tags or character data regions—events are created by the parser and passed to

an application-specific document handler. The application-specific handler uses the data associated with the event, such as element type name or character data, to create a custom data structure or perform other actions. In the visualization display, the handler would directly create the display without creating a data structure to contain the entire document.

The SAX parser is used in several applications in the book and is explained more fully in Appendix B. The example in Appendix B shows how to immediately print the element type names of the document, which demonstrates a rudimentary display. More realistic displays will need to create a temporary data structure to capture some data from the document to be combined with data occurring later in the document before creating the display. Those data structures can be incorporated into the Document Handler, which the SAX parser uses to handle application-specific code. The SAX parser uses generic code to parse the document, then calls methods in the Document Handler based on the content of the document, as described in Appendix B. For example, a Document Handler that keeps a stack of element type names is presented in DXD Example 7-11. The stack is used to verify the validity of the element type name nesting and to track the depth of the tree. The stack of element type names is defined by the class, pushed in the start element method, popped in the end element method, and accessed in the start element and characters method.

4.2.2 Tree Example

A more realistic application is given in DXD Example 7-12, which creates a tree representation of an XML document using JTree in Swing. A UML diagram for the Java implementation is given in DXD Figure 7-5. A visualization of the tree is shown in DXD Figure 7-6.

The TreeHandler is a subclass of org.xml.sax.HandlerBase and imports a SAX parser (from IBM) and the Swing JTree class from the package "javax.swing.tree.*" for Java 2 or the package "com.sun.java.swing.tree.*" for Java 1.1.x with Swing 1.0.x. It implements the org.xml.sax.DocumentHandler interface and creates a variable to contain an instance of java.util.Stack. It also creates an instance variable for a Swing DefaultTreeModel of which the JTree is a view. Swing uses the model-view-controller paradigm where the data structure capturing the information (model) is isolated from the visualization of the data (view). Thus, to present information using Swing both a model class (DefaultTreeModel) and a view class (JTree) must be created.

In addition to access methods for the instance variables, the methods for the handler are defined: start element, end element, and characters. The start element method creates a node in the tree for the current element as a child of the parent

node. The parent node is the top element of the stack or a root node created as the top of the document tree. The current element is then pushed onto the stack. When a character data region occurs in the parsing of the document, the characters method is called that creates a (leaf) node for the character data region. The end element method pops the stack of elements.

There are three parts to the application or applet: User Interface, Parser, and Document Handler. The process consists of

1. Creating an instance of the Parser. The code from Runner#parse (i.e., the parse method of Runner) is

```
String parserClass = "com.ibm.xml.parsers.SAXParser";
Parser parser = ParserFactory.makeParser(parserClass);
```

2. Creating and initializing an instance of the Document Handler. The code from Runner#parse is

```
HandlerBase handler = new TreeHandler(getTreeModel());
```

3. Initializing the Parser to use the Document Handler. The code from Runner#parse is

```
parser.setDocumentHandler(handler);
parser.setErrorHandler(handler);
```

4. Calling the Parser with the XML URL. The code from Runner#parse is

```
public void parse(String xmlFile) {
    ...
    parser.parse(xmlFile);
    ...
}
```

5. The Parser retrieves and parses the document, calling the Handler at

   - The beginning of each element. The code to do that is in TreeHandler#startElement:

```
public void startElement(String tag, AttributeList attrList)
        {
        MutableTreeNode currentParent = getCurrentParent();
        MutableTreeNode node = new
        DefaultMutableTreeNode(tag);
        treeModel.insertNodeInto(node, currentParent,
        currentParent.getChildCount());
        getParentStack().push(node);
}
```

   The method gets the current parent node from the getCurrentParent accessor, creates a node in the tree model, then pushes itself on the stack.

- The end of each character data region. The code to do that is in TreeHandler#characters

```
public void characters(char[] chars, int start, int length)
      {
      String string = new String(chars, start, length);
      string = string.trim();
      //skip whitespace
      if (string.length() == 0) return;
      MutableTreeNode currentParent = getCurrentParent();
      MutableTreeNode node = new
      DefaultMutableTreeNode(string);
      treeModel.insertNodeInto(node, currentParent,
      currentParent.getChildCount());
}
```

  The characters method trims whitespace and creates a node in the tree for the character data region.

- The end of each element. The code to do that is in TreeHandler#endElement:

```
public void endElement(String tag) {
      MutableTreeNode currentParent = getCurrentParent();
      if (tag.equalsIgnoreCase(currentParent.toString())) {
            getParentStack().pop();
      } else {
            System.err.println("Error at end tag: expected
      "+currentParent.toString()+" found "+tag);
      }
}
```

  The end element pops the stack and verifies that the stack and document are consistent.

6. After the visualization is created, the user interface is displayed to the user. The code to do that is in Runner#display, which is called from Runner#main.

Similar code can be used to generate a table, such as the table provided by JTable in Swing or a commercial widget (possibly implemented as a Java Bean). The table can be augmented to use active buttons to allow for drill-downs, much as hypertext links in HTML do. In addition, some of the default cells may be replaced with custom widgets to provide interactive visualizations of the data, for example, a widget to display a 3D rotating chemical structure that could be used in a chemistry database application.

Widgets can be connected to the type names of the elements either directly in the Java code or through a mapping stored in a repository or database. Although that mapping could be hard-coded, if the applications are undergoing rapid development, it may be useful to have the mappings stored in the database. Thus, new widgets may be added to the user interface without modifying the Java code.

In addition, when the XML is schema-driven, a completely new custom interface may be deployed by augmenting the schema and adding an entry to the widget mapping database.

To modify the display to handle widgets, the “startElement” method of the parser is modified to handle specific tags through alternative code. The widget display could be a hard-coded switch statement or a lookup table that was initialized from a database of widget mappings. The widget mapping can occur in one table that contains the element type name and the Java class to use. With a widget mapping database, the widgets may be selected from a predetermined set of Java classes or dynamically loaded using “classForName”. For simple widgets, the element type name and attribute list may suffice as parameters. In more complex examples, a temporary data structure may need to be created, in which case the widget display may need to associated with the “endElement” method.

5 Querying

Understanding the type of queries used and how to represent queries and data for efficient querying is important in determining the type of query processing to be included in an XML DBMS or to be used to query XML documents. DXD Section 8.1 describes a classification of queries and DXD Section 8.2 describes how to represent XML data to support more efficient querying. Sections 8.3 and 8.4 describe possible query engines, and DXD Section 8.5 describes query reporting tools.

5.1 Background

Querying is the process of selecting and synthesizing data from a database by asking questions based on characteristics of the data. The language in which the questions are asked facilitates or limits the answers that can be generated. Because slight variations in a query can yield dramatic differences in the report, it is essential that the query language mimic the structure of the data, capture the features of the domain, and be well-understood by the person forming the query.

5.1.1 Query Classifications

A query can be classified based on the source of the data, the data type of report generated, and the topology of the query. These classifications are not independent and may influence each other. These classifications and some of their interactions are discussed briefly here.

The source of the data to be accessed from an XML database during queries includes an XML document, a fragment of an XML document, character data, or a collection of elements. The data could be an XML element, a string, or a set of elements. This chapter focuses on querying the elements of one or more XML documents (or fragments). Querying from character data is not particular to XML databases and is addressed by techniques for text searching; because it is not specific to XML databases, it is not covered here.

In addition to retrieving data in the representation in which it was stored, data may be combined with other data throughout the database to create novel reports. These reports may take the data type of a string, set of strings, document node, set of document nodes, table, document, or collection of documents.

The data type of the report may depend upon the source of the data. For example:

- An attribute value may be returned as a string.
- A collection of elements with identical structure might be returned as a table.

- One or more elements might be collected together as one document or kept separately as individual elements.
- Character data regions may also be embedded in new elements to construct an XML report.

There are four topologies for the query discussed in this chapter: singleton, path, tree, and graph. Queries can retrieve a single element node from a unique identifier; retrieve one or more nodes from a path as defined in an XML Path Specification; or retrieve a table of nodes from a tree or graph pattern. The distinction between a singleton, path, tree, and graph query is shown in DXD Figure 8-1, where a singleton is shown as a rectangle at a single node, a path is shown from the root node to one of the leaves, a graph is all the edges in the diagram, and a tree is all the edges in the diagram except those labeled as being part of a graph.

The data type of the query results may depend upon the query topology. A query of an individual node by its unique identifier may result in a string or an element, depending upon whether the identifier refers to a character data region or an element, respectively. The path query algorithm described in this chapter formats the result as a set of nodes. The tree and graph query algorithms render the result as a table. Technically, all four query topologies can be thought of as returning a table of results is needed: in the path query, the table has one column (mathematically, a set of 1-tuples), which is equivalent to a set, and in the singleton query, the one column has one row, which is equivalent to a single value.

For example, given the schema tree in DXD Figure 8-2, any individual entry in the underlying database may be accessed by a unique identifier. Path queries can be formed that retrieve the entries in the database that satisfy certain logical constraints (or predicates), such as:

- Find all entries by William Shakespeare.
- Find all entries titled Hamlet.
- Find all entries published by Prentice Hall.
- Find all entries containing the character data “Romeo” (regardless of the tag, i.e., either the author or title could contain Romeo).

Tree queries are used when conjunctions of paths are queried or when multiple results are required, such as:

- Find all entries for Hamlet by William Shakespeare.
- Find all entries that reference an entry for Hamlet by William Shakespeare.

- Find all titles and their publishers for entries by William Shakespeare.

Graph patterns extend the hierarchical queries supported by XML paths, increasing the complexity and flexibility of querying allowed. A tree structure of an XML document becomes a graph when nodes are connected by intradocument links (via IDREF) or interdocument links (via XML Link). Querying with a graph pattern allows an entire (indexed) XML document with links to be examined, and all fragments in the document that match the pattern are returned. Graph pattern queries are also used when two or more paths of a tree must be joined for a query. These queries are especially important for data mining.

An example of graph pattern query is

- Find all entries that refer to an entry by the same author. (As shown in DXD Figure 8-3.)

5.1.2 Node-centric versus Edge-centric Representations

Graph data models support the representation of organizations and relationships as the nodes and edges of a graph. The organization of documents and the semantic relationships of data may be modeled as either nodes or edges. The different combinations of those have been utilized in different systems. Node-centric representations of physical organization is fairly common in document processing systems and object-oriented database that support CAD/CAM tools. Node-centric representations of semantic relationships are common in relational databases. Edge-based representations of the organization of inter-linked documents are modeled in Grove, a model that influenced the development of GML, a predecessor of SGML and XML. Edge-centric representations of semantic relationships are modeled in semantic and graph data models.

A graph representation of data clarifies the choices. Consider the text:

"The book entitled *Romeo and Juliet* is authored by William Shakespeare."

This may be modeled as a graph as shown in DXD Figure 8-4.

As an edge-centric model, the graph may be captured by the relationships: title and author.

Title

book	title
1	Romeo and Juliet

Author

book	author
1	William Shakespeare

In a relational database, these two edge relationships could also be combined into a "book" relationship as long as each book has only one author.

In a node-centric model, the graph may be captured by the "book," "Romeo and Juliet" and "William Shakespeare" nodes, for example, as the following object:

```
Book
   title: Romeo and Juliet
   author: William Shakespeare
```

Data in the book example could be presented in a node-centric XML representation, such as follows:

```
<book id="1">
      <title>Romeo and Juliet</title>
      <author>William Shakespeare</author>
</book>
```

Or, in an edge-centric form:

```
<title book="1">Romeo and Juliet</title>
<author book="1">William Shakespeare</author>
```

Typically, the node-centric form is a more common representation for adding data about books to a database. However, the edge-centric form is more amenable to querying based on the characteristics "title" and "author" of a book.

To understand why, consider a collection of 1,000 books stored in either form. To find the book titled "Romeo and Juliet" in the node-centered collection, up to 1,000 books must be retrieved, then for each book, the title must be retrieved, then the title must be compared to "Romeo and Juliet" to see if it matches. In the edge-centric collection, up to 1,000 titles still must be retrieved and compared, but not the 1,000 "book" elements. When the elements have greater subelement depth, then the efficiency of the edge-centric approach increases compared to the node-centric approach.

The node-centric approach could be made more efficient by indexing the "title" element type names within the document. In fact, indexing all the element type names is as efficient as accessing the document in edge-centric form, and this is the approach taken to translate a node-centric document into an edge-centric form amenable to querying.

Edge-centric forms are superior for querying XML because:

- They support queries where an element type name is undefined.
- They cleanly capture in the same framework: element/subelement relationships, intra-document links using IDREFs, and inter-document links using XML Links.

- They support querying documents that have a mutable and frequently changing structure.
- They efficiently support queries with many links between documents.

Efficient querying of XML, especially across XML Links, requires translating a node-centric form to an edge-centric form. Either the data must be translated from a node-centric form to an edge-centric form or the query must be translated from an edge-centric form to a node-centric form.

XML has focused on the node-centric form of element, which made sense coming from a document organization perspective. Unfortunately, that means to effectively query an XML databases, the underlying gulf between a node-centric data model and an edge-centric data model must be crossed.

Because tools supporting XML technologies are typically node-centric, it is difficult to use XML tools to access relational databases. Instead it is simpler to modify database tools to work with the node-centric operations of XML. This chapter describes how it is possible to view XML elements in an edge-centric form.

5.1.3 Representing Links

This section describes how to create relationships between nodes, thus allowing data in a document to benefit from a more edge-centric approach. Because data in an XML database is stored in a non-linear form (i.e., it is not flat), it is easy to create edges between elements.

There are two possible ways to modify an XML data model to accept edges between existing nodes in a tree (called *links)*. The first is to allow multiple parents for a fragment as shown in DXD Figure 8-5. This would change the XML data models in DXD Chapter 3 to graph data models because the one parent per element rule would have to be relaxed. This solution would not allow XML to be used directly for all documents.

The second solution augments the XML data model to include a linking mechanism. Non-hierarchical relationships are captured as links as illustrated in DXD Figure 8-6 by a dashed line, where the ToyotasForSale document root element is linked to each appropriate automobile element. The second solution has been chosen because it is superior for working within the XML framework.

Querying a document with links should integrate link traversal and subelement relationships, supporting the following query against the schema in DXD Figure 8-7:

- Find all URLs for ToyotasForSale possibilities where status in blank.

Links also are bi-directional. For example, the following query could be asked against DXD Figure 8-8:

- Find all buyers interested in one-year-old automobiles.

Links are useful for capturing non-hierarchical relationships, such as those discovered during data mining. Because they are bi-directional, queries can be asked from different perspectives, allowing flexible and complex querying.

The biggest advantage of links is that they may span across documents, even at different sites. Because there are storage and security issues associated with links spanning sites, this chapter works with links within a single site. It is easy to have a million URLs to point to *www.cnn.org* (well, theoretically easy), but it would be much more difficult if CNN also had to keep track of all links in which anyone referred to their site. DXD Chapter 9 addresses storage of links that refer to documents at another site.

5.1.4 XML Links Presentations of Edges

XML Link Specification provides a mechanism for creating links between XML elements (as a special case of XML documents and other web resources). Our approach allows most (if not all) likely uses of XML Link and allows the links to be stored and manipulated cleanly with database operations. Database operations can also be expanded to allow for necessary node-centric operations, such as XPath references.

Links can be created in the non-linear database, but must be presented in a flat XML document to be useful. Generically, there are three ways to present any bi-directional link. Bi-directional links may be treated as a separate relationship or as if either element contained the other as a special kind of subelement.

In DXD Figure 8-9, an automobile and a buyer are separate XML documents. The link connecting the representation of buyer with the representation of auto could be presented as a separate link; as a buyer as a subelement of auto; or as an auto as a subelement of buyer. These three possibilities are outlined here.

The separate link presentation could look like this:

```
<auto ID="auto_1">
   <make>Chevy</make>
   <model>Cavalier</model>
   <year>2004</year>
</auto>
<buyer ID="buyer_1">
   <name>Fred Flintstone</name>
</buyer>
<link>
      <auto xlink:href="#auto_1"/>
```

```
      <buyer xlink:href="#buyer_1"/>
</link>
```

where the (slightly simplified) link element refers to the "auto" element by its ID and the "buyer" element by its ID. The "#" hypertext reference refers to the element with the name or ID of "#name" in the specified document, which, in this case, is the default current document.

In the second case, the link relationship between "auto" and "buyer" could also be presented as a subelement of "auto":

```
<auto>
   <make>Chevy</make>
   <model>Cavalier</model>
   <year>2004</year>
   <buyer xlink:href="#buyer_1"/>
</auto>
<buyer ID="buyer_1">
      <name>Fred Flintstone</name>
</buyer>
```

Or, in the third case, the relationship could be presented as a subelement of "buyer" as:

```
<buyer>
   <name>Fred Flintstone</name>
   <auto xlink:href="#auto_1"/>
</buyer>
<auto ID="auto_1">
   <make>Chevy</make>
   <model>Cavalier</model>
   <year>2004</year>
</auto>
```

To keep the use of links compatible with the edge-centered approach needed for querying, the first of the three options is chosen.

5.1.5 Storing Links

The edge-centric data model in DXD Section 3.5.6 describes a mechanism to store XML data in edge-centric form. If a node-centric data model is used, then the queries and relational views of DXD Section 8.4.4.1 may be used to translate the node-centric data model to an edge-centric form. However, in storing a document of XML links, those may be stored more efficiently in an edge-centric form. Because the storage mechanisms of DXD Chapter 4 do not directly address the storage of edge-centric data models, that issue is addressed here briefly in the context of storing the links.

An edge consists of a source, destination, and relation name. Thus, an edge may be stored in a relational database as a single binary relation:

```
Edge(Node source, String relationName, Node destination)
```

To store that relation using the relational DBMS storage mechanisms of DXD Chapter 4 across documents, the nodes will refer to fragment identifiers and will require a specification of document identifiers to support interdocument links. That relation is

```
Link(Document source_doc, Node source,
     String relationName,
     Document destination_doc, Node destination)
```

If the fine-grained or medium-grained storage relational solutions are used, then a table for "link" can be added to the schema as a new type of "child" table. The code to implement this is shown in DXD Example 8-1 for Oracle and DXD Example 8-2 for DB2.

5.2 Query Engines

Three query algorithms are described in this chapter. The first is a path querying algorithm (DXD Section 8.3.1); the second is a tree querying algorithm (DXD Section 8.3.2); the third is a graph querying algorithm for documents stored in a relational DBMS (DXD Section 8.4.4).

5.2.1 Path Querying

Path querying provides a mechanism to retrieve the nodes occurring at some position relative to a starting node. The query engine typically follows a *path* from the starting node to the ending nodes of the path. For example, in DXD Figure 8-10, following the path query

- Find all makes in the classifieds

would start at the “classified” node, pass through the “auto” node, and end at the “make” nodes. Thus, a query against DXD Example 8-3, would return: “Chevrolet”, “Toyota”, and “Volkswagen”. Starting from the root of the document, the path of the query would be “classifieds/auto/make”. The basic algorithm for the path query is as follows:

1. *Begin with the start node.* In this example, the start node would be the root node of DXD Example 8-3.
2. *Retrieve the subelements of the start node with the same element type name as the first type name in the path.* In this case, “classifieds” would be the first type name. There may be more than one subelement node with that element type name.

3. *For each subelement, repeat the previous steps with the remainder of the path, in effect, call the procedure recursively for each subelement.* Thus, the first time through the procedure would match the (only) “classifieds” node and call the procedure recursively. The second time through, the start node would be the classified node(s) and the path being searched would begin with “auto”. This step of the procedure would match each of the five “auto” elements in the example. The procedure would be called (recursively) five times with path “make”, once with each of the “auto” elements. The recursive parameter calls are given in DXD Table 8.1.
4. *If the path is complete, collect the subelements in a list to be returned when all branches have been completed.* The third time through the procedure—with path “make”—the path remainder is empty; so instead of calling the procedure recursively, the elements that matched “make” are added to a result list. Because there are five “auto” nodes and the procedure was called recursively five times in recursion 1, this step will occur five times for the example document.

In theoretical terms, the algorithm is a depth-first search of the part of the document tree corresponding to the path.

When executing the path query, it is possible to optimize the query based on additional information, if indices are available and a complete search is not required. For example, consider a database of one document that contains basic geographic, political, and economic information about the countries in the world. A query to find the population of each country might be "country/demographics/population" and proceed efficiently in the top-down manner described in the algorithm. However, a query to obtain the production of yak milk by a country might be "country/agriculture/yak_milk" and be more efficiently executed by first searching all "agriculture" elements for those that have "yak_milk" subelement, assuming that the "agriculture" elements were indexed. These optimizations are similar to optimizing join paths in queries of relational databases.

In addition to following a completely specified path, any node in an XML Path may be a wildcard or constrained with a node constraint. For example, if the “auto” node were split between types for “auto” or “truck,” the name of those nodes in the path could be replaced with a wildcard, such as “classifieds/*/make”. A node in the path could also be constrained, such as “classifieds/auto[price<10000]/make”, which would refer to makes of all autos with a price subelement with value less than 10,000. The algorithm is modified in these cases to use a more general retrieval of the subelement nodes in Step 2 and/or to filter those nodes based on the constraint.

In addition to following paths down the subelement tree, it is useful to follow paths up the tree or to follow paths up or down without specifying all the intermediate steps. For example, the path “classifieds//make” would find the make subelements without regard for the number of intermediate nodes (called the descendent nodes).

The XML Path Specification provides for thirteen dimensions in which a path may follow from a given context. Eight of the dimensions are geared toward navigating the subelements of the document tree.

- Child—contains the children of the context node.
- Parent—contains the single parent node, if there is one.
- Descendent—contains all the descendent nodes. These are all the subelement nodes, their subelement nodes, their subelement nodes, and so on.
- Ancestor—contains the parent node, its parent node, and so on, up to and including the root node.
- Preceding sibling—contains the previous siblings of the context node.
- Following sibling—contains the following siblings of the context node.
- Preceding—contains all the nodes preceding the context node, excluding any ancestor nodes.
- Following—contains all the nodes following the context node, excluding any descendent nodes.

Three of the dimensions include the context node within the document tree navigation. These dimensions may simplify some queries.

- Self—contains only the context node itself.
- Descendent-or-self—contains the context node and descendents.
- Ancestor-or-self—contains the context node and ancestors.

The remaining two dimensions of the XML Path Specification are to handle attributes and namespaces in the same framework as subelements.

- Attribute—contains the attributes of the context node.
- Namespace—contains the namespace node of the context node.

5.2.2 Tree Querying

Tree querying depends upon the data model chosen to represent the tree. Primarily tree data models may be either node-centric (as described in DXD Section 3.5.5) or edge-centric (as described in DXD Section 3.5.6).

Much of the XML activity has taken a node-centric approach to data modeling, which has historically been appropriate for a document-processing-oriented view of XML. However, an edge-centric approach is probably superior for querying.

In the short term, the use of node-centric querying facilitates the rapid development of a query data model and tools and ensures compatibility with other XML standards. However, it ultimately limits the integration and interoperability of XML and relational data. Relational data is already compatible with edge-centric queries because the edges of a tree are easily modeled as binary relations in a relational database.

Tree querying provides a mechanism to retrieve the nodes occurring at multiple positions relative to a starting root node. The query engine typically follows a *tree* from the starting node to the ending nodes of the path. For example, in DXD Figure 8-10, following the tree query

- Find all makes and models in the classifieds.

would start at the “classified” node, pass through the “auto” node, and end at the “make” and "model" nodes. Thus, a query against DXD Example 8-3, would return: “Chevrolet Cavalier”, “Toyota Camry”, Toyota Celica”, and “Volkswagen Bug”.

Tree querying is a special case of graph querying, which is described in the next section. A simple tree querying algorithm is presented here:

1. *Begin with the start node*. In this example, the start node would be the root node of DXD Example 8-3.
2. *For each subelement of the tree pattern, retrieve the subelements of the start node with the same element type name as the type name in the tree pattern.* In this case, “classifieds” would be the first type name. There may be more than one subelement node with that element type name.
3. *For each subelement, repeat the previous steps, in effect, call the procedure recursively for each subelement.* Thus, the first time through the procedure would match the (only) “classifieds” node and call the procedure recursively. The second time through, the start node would be the classified node(s) and the tree being searched would begin with “auto”. This step of the procedure would match each of the five “auto” elements in the example. The procedure would be called (recursively) for each matching subelement of the “auto”

elements, in effect, five times with tree “make” interspersed with five calls with tree "model".

4. *If the subtree is complete, collect the subelements in a table to be returned when all branches have been completed.* The table is associated with the root node of each tree in the pattern and these tables are joined (Cartesian product) after the completion of each subtree in the pattern.

In theoretical terms, the algorithm is a depth-first search of the part of the document tree corresponding to the tree pattern.

When executing the tree query, it is possible to optimize the query based on additional information, such as indices, as was mentioned for the path query. Indexing is discussed in DXD Chapter 9.

5.2.3 Graph Querying

Graph querying consists of two steps. The first step is to compare the graph pattern with the documents in the database. The second step is to generate a report from the result of the query.

Graph querying matches a graph pattern that describes a query against the documents in the database. Graph patterns are described in DXD Section 8.4.2, and DXD Section 8.4.4 and DXD Section 8.4.5 describe in detail two graph query algorithms.

Graph querying may be used across the subelement and link relationships that occur within a document as well as across links that span documents and databases. The subelement relationships and links can be viewed as a graph, where each subelement relationship or link is an edge in the graph. Although some cases exist where a complex link may be mapped to multiple edges in the graph, those may be addressed by creating a new node in the graph to connect the link components as individual edges.

The graph query algorithm is related to and more general than the path following algorithm or the tree querying algorithm. The path following algorithm follows a collection of linearly-connected edges from one start node to one end node. A tree querying algorithm follows a hierarchical collection of edges from one start node (the root) to several destination nodes (the leaves). The query algorithm "follows" an arbitrary collection of edges without concern for start or end nodes. For example, the graph in DXD Figure 8-11 may be followed in any order. In terms of result, it does not matter whether the make, model, or name edges are retrieved first. In general, the six edges may be matched in any of the 6! (=720) orders, but hopefully with the one leading to the most efficient query.

The graph pattern is compared with the documents in the database via the graph query algorithm described in DXD Section 8.4. The query algorithm has two parameters: the first is the graph pattern, as described in DXD Section 8.4.2; the second is the document (or documents) against which the query pattern is to be compared. The document (or documents) may be specified directly or using some of the ways described in DXD Section 8.5.1 to select documents to be queried.

The result of the graph query is a collection of graphs. The graphs consist of the graph pattern and the variable bindings in the query result. The variable bindings are presented as a table. The table may be rendered as a tabular report, as would occur with a relational database query, or it may be rendered as an XML document. In either case, only some of the variables would be desired for the report. A simple XML document can be created from the query result table by creating a single element for each record in the table. For example, the report of the query in DXD Figure 8-11 might result in a document such as this:

```
<buyers>
   <auto>
         <model>Camry</model>
         <year>2000</year>
         <price>7995</price>
         <name>Fred Flintstone</name>
   </auto>
   <auto>
         <model>Celica</model>
         <year>2003</year>
         <price>14995</price>
         <name>George Jetson</name>
   </auto>
<buyers>
```

The document could be returned to the user as is or could be processed further. For example, the graph pattern could also be used in some cases to generate a document that has a similar structure to the source document, though that would be difficult if the graph pattern was not a strict tree. A more useful report may be generated by using an XSL stylesheet on the query result document to further transform the data into a custom report.

If a web architecture for the XML database is used—such as one described in DXD Chapter 5—the query tool may be embedded within a web server, which would result in an XML report being returned for the query.

5.3 Path Query Tools

This section is not complete.

5.3.1 Path Querying with XSL

The templates in XSL match paths in a document before executing the body of the template. The same form can be used to combine path querying in an XML database with report generation using XSL. One approach to supporting path querying in an XML database is to modify an XSL query engine to perform path queries against the documents in the database.

Although it is possible to retrieve the entire document before performing the match, it is more efficient to retrieve only the nodes that are specified in the path patterns. The nodes may be accessed directly from the database when needed. Caching and database indexing may improve performance of the database request.

A useful extension to the XSL query engine would support querying across documents in a database. XSL supports pattern matching against the elements of a document but not across the collection of documents in a database. There are three possible solutions:

a. To match against all the elements in all the documents.

b. To restrict querying to only one document.

c. To store additional information about the documents, which could then be filtered to limit the documents, against which the querying is performed.

Solutions (a) and (b) are problematic. Solution (a) is time-consuming and potentially uninformative, and Solution (b) is overly restrictive. Solution (c) addresses those issues by allowing filtering appropriate to the query. For Solution (c), information you can store about a document includes some of the following:

- Name of the document.
- Descriptive features of the document that are defined by the user.
- Any element type names used in the document.
- The person who created the document.
- The owner of the document (if different from the creator).
- When the document was created and last accessed.
- Summary information defined by the user. For example, in the automobile buyer examples, the summary information might include the source of the classifieds and in the Wal-Mart example, the summary information might include the store.

Any of this information can be used to filter the documents to be included in the query. A query would be performed only on documents that met those specific restrictions.

The information can be stored in a specialized form in the database. The storage structures in DXD Chapter 4 support storage of the name of the document. The indexing strategies in DXD Chapter 9 address ways in which the element type names might be associated with a document. A specialized data structure can also be created for the descriptive features of the document defined by the user, or that data may be stored as a summary or meta-data document in XML. The summary or meta-data document can also be searched using the XSL path querying. It may also be possible to store some of that information as a fragment within the document.

XPath, XSLT, or other path-centric querying tools are most useful with a document-processing-oriented view of the data. Graph querying is more useful when taking a data-processing-oriented approach.

5.4 RGQuery

An XML DBMS must provide access to data as edges to support path or graph querying. Path querying simplifies the access because each edge is traversed linearly; thus, the edge retrieval always specifies one of the nodes and the edge label and requests the other node. Graph querying requires more general access as the edges may be requested in any order; thus, the edge retrieval may have any combination of source node, edge label, or destination node unspecified.

DXD Chapter 9 describes how to efficiently provide the edge access of the DBMS to support querying. The remainder of this chapter describes how to support querying when those edges are available and how to perform querying when using the storage approach of DXD Chapter 3. The indexing strategies in DXD Chapter 9 are particularly useful when storing XML in custom storage facilities (such as a new XML DBMS) or on top of an object-oriented DBMS. When a relational DBMS is used to store the XML documents, then completely indexing the relational tables may provide reasonably efficient query support, which this section describes. When larger queries are performed or a storage facility other than a relational DBMS is used, the algorithm in DXD Section 8.4.5 is necessary. In either case, information from the indexed database, such as data distribution, can be used to optimize the queries.

5.4.1 Graph Data Model

A graph is a set of vertices and labeled edges. Labeled edges connect two vertices and have a type name. The vertices are the nodes in a document and the edges are the link names or element type names. A graph may be represented as a list of edges. The following document fragments can be represented as the graph shown in DXD Figure 8-12 where the vertices are rectangles and the edges are lines.

```
<auto id="auto_1">
      <make>Toyota</make>
      <model>Celica</model>
      <year>2003</year>
      <price>14995</price>
      <description>Runs great</description>
</auto>

<BuyerNotebook id="BuyerNotebook_1">
      <name>Fred Flintstone</name>
      <possibility xlink:type="simple" xlink:href="#auto_1"/>
</BuyerNotebook>
```

The edges are also listed in DXD Table 8.2.

A data model was given in DXD Section 3.5.6 that supports accessing subelement (child) relationships and attributes of a document. To support links, the edge-centric data model is extended to contain a new dimension called "link" and to allow edges to refer to nodes in different documents. The simple binary links can be captured directly as edges.

5.4.2 Graph Patterns

Just as the subelement or link relationships may be represented as a graph with vertices and edges, the patterns that would match them may be represented as a graph with vertices, edges, and variables. For example, the pattern corresponding to an automobile with its make and model can be diagrammed using the graph pattern in DXD Figure 8-13, where auto, make, and model are variables, denoted by ovals in the diagram.

A graph pattern is described in terms of its edges. The edges in DXD Figure 8-13 are listed in DXD Table 8.3. The graph pattern consists of two edge patterns that can be denoted as (?auto make ?make) and (?auto model ?model) where “?name” denotes a variable and “make” and “model” are the edge label names. A graph pattern consists of a collection of edge patterns.

In practice, it may be useful for a graph pattern to contain report information, such as a name of the pattern for the title of a report, and an ordered list of variables in the pattern whose order serves as the order for the columns of a report.

To contain the edge patterns and report information, a graph pattern can be rendered as an XML document, such as the one in DXD Example 8-4, which is an XML representation of the graph in DXD Figure 8-11.

5.4.3 Retrieving Edges

To compare graph patterns against edges retrieved from the XML database, the database must support viewing the nodes and relationships of the XML document as edges. The mechanism to do that depends upon the storage system used. If a relational database is used to store the XML documents, then a relational view can be created on the XML storage tables. If an object-oriented database is used for storage or a custom implementation, then the mechanism would need to be implemented within that system.

For the fine-grained relational storage system described in DXD Chapter 4, edges may be viewed by combining information stored in the tables. The Oracle SQL in DXD Example 8-5 shows the relationships between each element and its subelements, and DXD Example 8-6 has the equivalent SQL for DB2. The result of those queries on the DXD Example 8-3 document is given in DXD Table 8.4. DXD Example 8-7 is the (Oracle and DB2) query that shows the relationships between elements and their character data regions stored as strings. A similar query could be formed to show the relationships between elements and their character data regions stored as Oracle Longs or DB2 CLOBs. The DXD Example 8-7 query shows the character data sections that occur as the first child of an element and thus assumes that a data-processing-oriented approach has been taken for the documents where each character data region is individually embedded in a element. DXD Table 8.5 is the result for the query against DXD Example 8-3.

A relational view can be created that combines the two queries; however, there would be some overlap as the leaf-most subelement relationship is redundant. This can be filtered from the view by eliminating those relationships from the subelement part of the query and including only subelement relationships that also contain a subelement relationship. DXD Example 8-8 for Oracle is such a relational view and the (sorted) view of the document in DXD Example 8-3 is given in DXD Table 8.6. DXD Example 8-9 shows the equivalent DB2 view.

Such a filtered view provides a way to retrieve binary relationships for a specific document. For example, the SQL in DXD Example 8-10 for Oracle (and DXD Example 8-11 for DB2) retrieves the binary relationships for the “make” relationships in document “7”. To retrieve relationships across documents, the view in DXD Example 8-12 for Oracle (and DXD Example 8-13 for DB2) combines the document and element identifiers as a fragment identifier. When

retrieving relationships across documents, it may be useful to sort the result by document and element identifier. The SQL to perform that query is given in DXD Example 8-14 for Oracle (and DXD Example 8-15 for DB2), and the result of that query for DXD Example 8-3 is given in DXD Table 8.7

If the link table of DXD Section 8.2.5 is added to the database, then the query algorithm would need to include a query of the link table as well as the relational views of the element/subelement relationships. This could be done by creating a new view that combines element queries with link queries or by modifying the algorithm to allow either or both to be specified as part of the query.

5.4.4 RGQuery Algorithm

For small queries, a simple algorithm can be used where the edge patterns are joined together. This algorithm is especially amenable to implementation when the documents are stored in a relational database. The algorithm generates a SQL statement that will result in executing a graph pattern query.

For example, consider the graph pattern in DXD Figure 8-16, which retrieves the make, model, and year of an automobile. The SQL that performs the query is in DXD Example 8-16 for Oracle (and DXD Example 8-17 for DB2), and the result of the query on DXD Example 8-3 is given in DXD Table 8-8.

The pseudo-code to create an SQL query from a graph pattern is as follows:

1. For each edge in the graph pattern
    1.1. Create a *table alias* of the database edge table to be included in the "from" clause, such as the table alias "xdb_ele_edges_help_v t1".
    1.2. Create a "where" constraint for the *source variable*. The name of the variable in the SQL is "<table_alias>.source". The name should be stored in a local repository of variable names, such as an associative array or hashtable. If the variable has occurred before (and thus it is already in the repository), a "where" constraint is created where the current variable is constrained to be equal to the variable name from the repository. For example, if the variable "auto" occurs in the first three edge patterns, its name would be "t1.source" in the repository, and constraints would be added for the second and third edge patterns that "t1.source = t2.source" and "t1.source = t3.source". If a variable has not occurred in the graph pattern, no "where" constraint is needed.
    1.3. Create a "where" constraint for the *edge label.* The relation name column in the table is constrained to be the relation mentioned in the edge pattern. For example, if the label of the first edge is "make", a constraint is added where "t1.rel = 'make'".

1.4. Create a “where” constraint for the *destination*. If the destination of the edge is a variable, it is treated the same as the source variable, and the same repository is used for the source and destination variables. If the destination is a constant, it is handled in the same way as the constant edge label, and a constraint is created to specify that the destination column have that value. For example, if the destination of the first edge is the variable “make”, the name “t1.dest” would be associated with the name “make” in the repository. If the destination of the first edge is the constant “Toyota”, a constraint is added where “t1.dest = ‘Toyota’”.

2. Emit the “SELECT”.
3. For each variable in the parameter list of the graph pattern, retrieve its SQL name from the repository. Emit it as part of the select list, with the name of the variable as a column alias. For example, if the parameters were “make, model, year”, the select list would be “t1.dest make, t2.dest model, t3.source dest”.
4. Emit the collection of table aliases created in Step 1.1 as the “from” clause.
5. Emit the collection of “where” constraints created in Steps 1.2–1.4.

The SQL code in DXD Example 8-16 is slightly more complicated than the algorithm described in the pseudo-code because the Oracle and DB2 views contain multiple documents and splits destination into two columns: the numeric fragment reference and the string character data value. The variable constraints added in Step 1.2 and 1.4 can be augmented to constrain the document columns in addition to the source or destination columns. The view described in DXD Example 8-12 for Oracle (and DXD Example 8-13 for DB2) eliminates the need in user queries for the “decode” statement in Oracle SQL (or "case" statement in DB2 SQL).

5.4.5 RGQuery Implementation

DXD Example 8-18 is the Java code to read an XML graph pattern and create a SQL statement. An UML diagram for the Java implementation is given in DXD Figure 8-17. The program takes a graph pattern, such as the one in DXD Example 8-19, creates SQL (as given in DXD Example 8-20 for Oracle and DXD Example 8-21 for DB2), executes the SQL against the database, and returns the result as an XML report, shown in DXD Example 8-22.

The Java implementation consists of a Test class whose main method calls the XML parser, builds the SQL, and executes it against the database. The building and execution of the SQL uses the RServe package from DXD Chapter 5.

The parser uses a PatternHandler instance, which is a SAX handler that creates a GraphPattern object for the graph pattern in the XML file. A GraphPattern consists of a name and an EdgePattern for each edge. The EdgePattern has a source, label, or destination that can be either a PatternConstant or PatternVar (variable). The PatternConstant and PatternVar are both subclasses of the PatternNode abstract class.

After the GraphPattern object is created from the XML graph pattern file by the XML parser, a StatementBuilder object is created to build a SQL statement from the GraphPattern. The SQL statement is built using the AccessSpec class of the RServe package. Query and constraint objects are created as part of the AccessSpec that are used to generate the SQL string after all parts of the query are specified. The SQL string is generated and sent to the database using the Input class of RServe, and the resulting data is rendered using RServe's FormatXML object.

5.4.6 Commands

The following arguments can be passed to RGQuery. They may be passed as servlet arguments, taglib fields, or calls to Java method dispatchArg.

name	*value*
doc	id for document in the database to query
text	text of graph query in XML form to use
url	url for graph query in XML form to use
stylesheet	stylesheet to be used in formatting the result
templatedoc	id for a document in the XML database to use as graph query

Only one of "text", "url", or "templatedoc" should be specified.

The result of the query is an XML document with a root element type name of "collection". This is identical to the XML document created by a RDB query using RServe.

6 Java Utilities

Most of the examples in the book require a connection to a relational database as well as output to a servlet. Rather than duplicate the description of those utilities where used, they are presented in this Appendix. Also presented in the appendix is a "Default" class for the systems in the book. This class collects together the system parameters that can be set to make the software examples work together in a system.

The access utilities consist of three parts: a relational database connection, classes that provide output streams, and an interface that defines interaction with both a relational database and output source. The Java examples in the book use only these classes for interacting with the database and output source, which provides a simple way to change the RDBMS used or the output source.

The only other Java packages used are a JDBC client and a SAX parser.

6.1 System Defaults

The code for the class is in the class org.xwdb.xmldb.Default. The class contains values for four (static) variables. The variables are

- DBProduct—defines the relational DBMS used for storage
- xmldbAcct—defines the account within the DBMS that contains XML documents in a form defined in DXD Chapter 4
- reldbAcct—defines the account that contains any relational tables for access by the system describe in DXD Chapter 5
- parserClass—describes the SAX parser used (as described in Appendix B)

The variables are accessed by the system via getter/setter methods. The default values are

- DBProduct = ORACLE

The format for the access string for the variables xmldbacct and reldbacct is: "jdbc:oracle:thin:acct/password@machine:port:instance" for Oracle JDBC or

"jdbc:db2:instance" for IBM DB2 via JDBC. Their defaults are

- xmldbAcct = "jdbc:oracle:thin:xmldb/xmldb@127.0.0.1:1521:ORCL";
- xmldbAcct = "jdbc:db2:XMLDB";

The default value for "reldbAcct" is null, which means that the same value as the "xmldbAcct" is used.

The default value for the SAX parser is the IBM xml4j parser:

- parserClass = "com.ibm.xml.parsers.SAXParser";

The entire code for the class is:

```
package org.xwdb.xmldb;

/**
 * Contains defaults for system
 */
public class Default {
      //access string is acct/password@machine:port:instance
      for Oracle JDBC
      public static String xmldbAcct =
      "jdbc:oracle:thin:xmldb/xmldb@127.0.0.1:1521:ORCL";
      //public static String xmldbAcct = "jdbc:db2:XMLDB";
      public static String reldbAcct = null;
      //public static String reldbAcct =
      "jdbc:oracle:thin:scott/tiger@127.0.0.1:1521:ORCL";
      public static String parserClass =
      "com.ibm.xml.parsers.SAXParser";
public Default() {
      // do not instantiate
}
public static String getParserClass() {
      return parserClass;
}
public static String getReldbAcct() {
      if (reldbAcct == null) {
         return getXmldbAcct();
      }
      return reldbAcct;
}
public static String getXmldbAcct() {
      return xmldbAcct;
}
public static void setParserClass(String newValue) {
      Default.parserClass = newValue;
}
public static void setReldbAcct(String newValue) {
      Default.reldbAcct = newValue;
}
public static void setXmldbAcct(String newValue) {
      Default.xmldbAcct = newValue;
}
}
```

6.2 Relational Database Connection

Several pieces of software in the book require a connection to a relational database. They all use a simple utility written in Java to connect to a relational DBMS using JDBC. There are many commercial and free systems that provide

the same functionality and often provide much more. Some of these are integrated with web servers as an application server. The utility in this Appendix is used because it provides only the functionality required for the examples in the book and is simple to understand.

The RDB utility consists of two interfaces and two classes. The interface RDB provides access to a relational database with methods to open, close, query, and modify the database. The interface RDBAcct encapsulates the account information needed to access the relational DBMS. The class DBConnector implements the RDB interface and provides access to a DBMS through JDBC. The class JDBCAcct implements the RDBAcct interface for JDBC access. The classes OracleDB and DB2DB are subclasses of DBConnector.

The code for RDB interface is

```
package org.xwdb.xmldb.util.rdb;

import java.sql.*;
/**
 * Interface to access a Relational Database
 */
public interface RDB {
void close();
public boolean connect(RDBAcct acct);
public boolean connect(Connection conn);
ResultSet executeQuery(String statement) throws SQLException;
int executeUpdate(String statement) throws SQLException;
ResultSet getData(String query);
String getDataItem(String query);
public boolean isClosed();
}
```

The code for the interface RDBAccess is

```
package org.xwdb.xmldb.util.rdb;

/**
 * Account and access for relational database.
 */
public interface RDBAcct {
String getAcct();
void setAcct(String acct);
String toString();
}
```

The code for the class JDBCAccess is

```
package org.xwdb.xmldb.util.rdb;

/**
 * Captures acct and access information for connection to RDB.
 */
public class JDBCAcct implements RDBAcct {
      protected String jdbcaccess = null;
```

```
public JDBCAcct() {
      super();
}
public JDBCAcct(String acct) {
      super();
      setAcct(acct);
}
public String getAcct() {
      return jdbcaccess;
}
public void setAcct(String acct) {
      this.jdbcaccess = acct;
}
public String toString() {
      return getAcct();
}
}
```

The code for the class DBConnector is

```
package org.xwdb.xmldb.util.rdb;

import java.sql.*;
/**
 * Implements access to a relational database
 * Requires JDBC Thin Driver in the class path.
 *
 * Drivers are registered from the array driverArray.
 * Connection string is connectStringPrefix + acct +
      connectStringSuffix
 */
public class RDBConnector implements RDB {
      protected Statement[] statement = new Statement[32];
      protected RDBAcct acct = null;
      protected Connection conn = null;
      //If you always use the same DBMS, set
      connectStringPrefix
      // to be the appropriate URL prefix. Then only the
      account
      // is needed as part of the connect string.
      //public String connectStringPrefix =
      "jdbc:oracle:thin:";
      public String connectStringPrefix = "";
      //If you always use the same machine and instance, set
      connectStringSuffix
      // to be the appropriate URL suffix. Then only the
      account
      // is needed as part of the connect string.
      public String connectStringSuffix = "";
      public String[] driverArray = {
         "oracle.jdbc.driver.OracleDriver",
         "COM.ibm.db2.jdbc.app.DB2Driver"
      };
public RDBConnector() throws SQLException {
      super();
      register();
}
public RDBConnector(RDBAcct acct) throws SQLException {
```

```
        super();
        setAcct(acct);
        register();
        connect();
    }
    public void close() {
        try {
           if (conn.isClosed()) {
                 return;
           } else {
                 conn.close();
           }
        } catch (SQLException ex) {
           ex.printStackTrace();
        }
    }
    public boolean connect() {
        try {
           if (conn != null) {
                 return true;
           }
           conn =
        DriverManager.getConnection(getConnectString());
           return true;
        } catch (SQLException ex) {
           ex.printStackTrace();
           return false;
        }
    }
    public boolean connect(RDBAcct acct) {
        setAcct(acct);
        return connect();
    }
    public boolean connect(Connection newconn) {
        conn = newconn;
        return true;
    }
    public ResultSet executeQuery(String query) throws SQLException
        {
        return executeQuery(query,0);
    }
    public ResultSet executeQuery(String query, int statementNum)
        throws SQLException {
        if (statement[statementNum] == null) {
           if (conn == null) {
                 System.err.println("Database is closed for " +
        query);
                 return null;
           }
           statement[statementNum] = conn.createStatement();
        }
        return
        statement[statementNum].executeQuery(query.trim());
    }
    /**
     * execute a sql statement that does not return a result
     * return the number of rows modified, or 0 if no result
```

```
 */
public int executeUpdate(String sql) throws SQLException {
      try {
         if (statement[0] == null) {
               if (conn == null) {
                     System.err.println("Database is closed for
      " + sql);
                     return 0;
               }
               statement[0] = conn.createStatement();
         }
         return statement[0].executeUpdate(sql.trim());
      } catch (SQLException ex) {
         ex.printStackTrace();
         return 0;
      }
}
public RDBAcct getAcct() {
      return acct;
}
public java.sql.Connection getConnection() {
      //Provide direct access to RDB Connection. This typically
      will not be needed.
      return conn;
}
public String getConnectString() {
      return getConnectStringPrefix() + getAcct() +
      getConnectStringSuffix();
}
public String getConnectStringPrefix() {
      return connectStringPrefix;
}
public String getConnectStringSuffix() {
      return connectStringSuffix;
}
public ResultSet getData(String query) {
      return getData(query,0);
}
public ResultSet getData(String query, int statementNum) {
      try {
         return executeQuery(query, statementNum);
      } catch (SQLException ex) {
         ex.printStackTrace();
         return null;
      }
}
/**
 * Get the singleton value of a query
 */
public String getDataItem(String query) {
      //returns the value of the first column of the first row
      try {
         ResultSet resultSet = getData(query);
         if (resultSet == null) {
               //error in query
               //we may have problems later, but return null
               //throw new Error("Error in query: " + query);
```

```
                return null;
            }
            resultSet.next();
            return resultSet.getString(1);
        } catch (SQLException ex) {
            ex.printStackTrace();
            return null;
        }
    }
    public Statement getStatement() {
        return statement[0];
    }
    public Statement getStatement(int statementNum) {
        return statement[statementNum];
    }
    public boolean isClosed() {
        if (conn == null) {
            return true;
        } else {
            try {
                return conn.isClosed();
            } catch (SQLException ex) {
                ex.printStackTrace();
                return true;
            }
        }
    }
    public void register() {
        //registers all the drivers in driverArray
        for (int i = 0; i < driverArray.length; i++) {
            try {
                Class.forName(driverArray[i]);
            } catch (Exception ex) {
                //ignore failed drivers
                //ex.printStackTrace();
            }
        }
    }
    public void setAcct(JDBCAcct newValue) {
        this.acct = newValue;
    }
    public void setAcct(RDBAcct newValue) {
        this.acct = newValue;
    }
    public void setConnectStringPrefix(String newValue) {
        this.connectStringPrefix = newValue;
    }
    public void setConnectStringSuffix(String newValue) {
        this.connectStringSuffix = newValue;
    }
    }
```

The code for OracleDB is

```
package org.xwdb.xmldb.util.rdb;

import java.sql.*;
```

```
/**
 * Implements access to Oracle
 * Requires JDBC Thin Driver in the class path.
 */
public class OracleDB extends RDBConnector implements RDB {
public OracleDB() throws SQLException {
      super();
      setConnectStringPrefix("jdbc:oracle:thin:");
}
public OracleDB(RDBAcct acct) throws SQLException {
      super();
      setAcct(acct);
      setConnectStringPrefix("jdbc:oracle:thin:");
      connect();
}
}
```

The code for DB2DB is

```
package org.xwdb.xmldb.util.rdb;

import java.sql.*;
/**
 * Implements access to DB2
 * Requires JDBC Thin Driver in the class path.
 */
public class DB2DB extends RDBConnector implements RDB {
public DB2DB() throws SQLException {
      super();
}
public DB2DB(RDBAcct acct) throws SQLException {
      super();
      setAcct(acct);
      connect();
}
public boolean connect() {
      try {
         if (conn != null) {
               return true;
         }
         conn = DriverManager.getConnection("jdbc:db2:xmldb",
      "", "");
         return true;
      } catch (SQLException ex) {
         ex.printStackTrace();
         return false;
      }
}
}
```

6.3 Servlet Output

Three classes provide output for the software described in this book: the Output class, which writes to standard output; ServletOutput, which inherits from the Output class and provides the ability to write to a ServletOutputStream; and

JSPOutput, which inherits from the Output class and provides the ability to write to a JSPWriter.

The code for Output is

```
package org.xwdb.xmldb.util.io;

import java.io.*;
/**
 * Handles output to an output source.
 */
public class Output {
public Output() {
      super();
}
public void write(String s) {
      System.out.print(s);
}
public void writeln(String s) {
      System.out.println(s);
}
}
```

The code for ServletOutput is

```
package org.xwdb.xmldb.util.io;

import javax.servlet.*;
import javax.servlet.http.*;
/**
 * Handles Servlet output
 */
public class ServletOutput extends Output {
      protected ServletOutputStream out = null;
public ServletOutput() {
      super();
}
public ServletOutput(ServletOutputStream newValue) {
      super();
      setOut(newValue);
}
public ServletOutputStream getOut() {
      return out;
}
public void setOut(ServletOutputStream newValue) {
      this.out = newValue;
}
public void write(String s) {
      try {
         out.print(s);
      } catch (Exception ex) {
         ex.printStackTrace();
      }
}
public void writeln(String s) {
      try {
         out.println(s);
```

```
        } catch (Exception ex) {
           ex.printStackTrace();
        }
    }
    }
```

The code for JSPOutput is

```
package org.xwdb.xmldb.util.io;

import javax.servlet.jsp.*;
public class JSPOutput extends Output {
      protected javax.servlet.jsp.JspWriter out = null;
public JSPOutput() {
      super();
}
public JSPOutput(JspWriter newValue) {
      super();
      setOut(newValue);
}
public JspWriter getOut() {
      return out;
}
public void setOut(JspWriter newOut) {
      out = newOut;
}
public void write(String s) {
      try {
         out.print(s);
      } catch (Exception ex) {
         ex.printStackTrace();
      }
}
public void writeln(String s) {
      try {
         out.println(s);
      } catch (Exception ex) {
         ex.printStackTrace();
      }
}
}
```

6.4 Interactive Access Interface

The RDBInteractor class combines functionality from the RDB interface and Output class. The code for RDBInteractor is

```
package org.xwdb.xmldb.util.rdb;

import org.xwdb.xmldb.util.rdb.*;
import org.xwdb.xmldb.util.io.*;
/**
 * Basic RDB interactor.
 */
```

```
public abstract class RDBInteractor {
      private RDB rdb = null;
      //access string is acct/password@machine:port:instance
      for JDBC, or null for stdout
      protected String rdbAccessString =
      "xmldb/xmldb@127.0.0.1:1521:ORCL";
      protected Output output = null;
public RDBInteractor() {
      super();
}
public void connect() {
      try {
         setRdb(new OracleDB(new JDBCAcct(rdbAccessString)));
      } catch (java.sql.SQLException ex) {
         ex.printStackTrace();
      }
}
public void connect(String value) {
      this.setRdbAccessString(value);
      connect();
}
public Output getOutput() {
      if (output == null) {
         setOutput(new Output());
      }
      return output;
}
public RDB getRdb() {
      if (rdb == null) {
         if (getRdbAccessString() != null) {
               //create a connection to Oracle
               connect();
         }
      }
      return rdb;
}
public String getRdbAccessString() {
      return rdbAccessString;
}
public void setOutput(Output newValue) {
      this.output = newValue;
}
protected void setRdb(RDB newValue) {
      this.rdb = newValue;
}
public void setRdbAccessString(String newValue) {
      this.rdbAccessString = newValue;
}
}
```

7 Reference

7.1 Commands

The following commands are used to access RServe, XServe, and RGQuery. They may be used via the servlet interface, the taglib interface, or the Java API method dispatchArg.

RServe has one command:

- rdb – query a traditional relational database

XServe has four commands:

- store – store a document in the XML database or replace a fragment in a document
- retrieve – retrieve a document from the XML database
- append – append an XML element or fragment as a child to an existing document or fragment
- delete – delete an XML document from the database or a fragment from a document in the database

RGQuery has one command:

- query – query against a document in the XML database using a graph query

All commands take as an argument:

submit	ignored (makes using HTML forms easier)
ignore	ignored (makes using HTML forms easier)
ss	ignored in most cases. It is used to generate a stylesheet externally instead of in the browser via stylesheet XML processing instruction (only works with custom JSP applications)

7.1.1 RDB Command

The following arguments can be passed to RServe using the "RDB" command. They may be passed as servlet arguments, taglib fields, or calls to Java method dispatchArg.

name	*value*
tablename	name of relational table to query
id	id value
acct	database connect string to use
depth	maximum depth in the tree to render (each level requires additional db access)
elementname	name of record element (default is "record")
xmlfragement	returns fragment instead of document if set to any value, such as "1"
stylesheet	embed in document as stylesheet
reportargs	list of columns that use used to generate the report (comma-delimited)
orderby	order of columns (subelements) in document
constraintstr	use as where clause of SQL String
sqlstring	use the SQL String
all	other args are used to build where clause of SQL string
submit	ignored (makes using HTML forms easier)

7.1.2 Store Command

The store command is used to store a document in the database or replace a fragment in a document. The following arguments may be used with the "Store" command to XServe:

name	*value*
frag	fragment id "doc.element" in the database to replace
name	name for the new document in the database
text	text of XML to store in the database
url	url of XML to store in the database
texttag	element type name to wrap the value of "text" parameter. This is useful in passing in a string as "text" and having it stored as an element.

addslice	element type names to store using medium-grained storage

Only one of "doc" or "name" arguments should be used. Only one of "text" or "url" should be used. The "texttag" argument can only be used with the "text" command.

7.1.3 Retrieve Command

The following arguments may be used with the "Retrieve" command:

name	*value*
doc	id of a document in the database
frag	fragment id "doc.element" in the database
name	name of a document in the database
head	if value is "1", the document includes the ?xml processing header at the top of the file with version number "1.0". Having this option allows XML fragments to be retrieved as an XML document, and an XML document to be retrieved as an XML fragment.
stylesheet	href of the XSL stylesheet with which to render the document or fragment
includefragid	if included with any value, such as "1", an attribute named "FRAGID" is included for every element in the resulting document, and the element value is the unique fragment id for that element in the database. This value may be passed to the retrieve command to retrieve only this fragment. Note: in medium-grained storage, the elements in slices that are not parsed will not have fragment identifiers.

Only one of "doc", "frag", or "name" arguments should be used.

7.1.4 Append Command

In the "Append" command, the XML data is added as the last child of the fragment specified. If a fragment identifier is not specified, then the top-level element in the document is appended.

The following arguments may be used with the "Append" to XServe command:

name	*value*
doc	id of a document in the database
frag	fragment id “doc.element” in the database
name	name of a document in the database
text	text of XML to store in the database
url	url of XML to store in the database
texttag	element type name to wrap the value of “text” parameter. This is useful in passing in a string as “text” and having it stored as an element.

Only one of “doc”, “frag”, or “name” arguments should be used. Only one of “text” or “url” should be used. The “texttag” argument can only be used with the “text” command.

7.1.5 Delete Command

The “Delete” command to XServe does not take any additional arguments other than the specification of the document or fragment. Deletion by name is not allowed.

doc	id of a document in the database
frag	fragment id “doc.element” in the database

Only one of “doc”, “frag” arguments should be used.

7.1.6 Query Command

The following arguments can be passed to RGQuery. They may be passed as servlet arguments, taglib fields, or calls to Java method dispatchArg.

name	*value*
doc	id for document in the database to query
text	text of graph query in XML form to use
url	url for graph query in XML form to use
stylesheet	stylesheet to be used in formatting the result

templatedoc	id for a document in the XML database to use as graph query

Only one of “text”, “url”, or “templatedoc” should be specified.

The result of the query is an XML document with a root element type name of “collection”. This is identical to the XML document created by a RDB query using RServe.