Oracle Berkeley DB XML

Introduction to
Berkeley DB XML

Release 2.5

ORACLE

BERKELEY DB

Legal Notice

This documentation is distributed under an open source license. You may review the terms of this license at:
http://www.oracle.com/technology/software/products/berkeley-db/htdocs/xmloslicense.html

Oracle, Berkeley DB, Berkeley DB XML and Sleepycat are trademarks or registered trademarks of Oracle. All rights to these marks are reserved. No third-party use is permitted without the express prior written consent of Oracle.

Java™ and all Java-based marks are a trademark or registered trademark of Sun Microsystems, Inc, in the United States and other countries.

To obtain a copy of this document's original source code, please submit a request to the Oracle Technology Network forum at:
http://forums.oracle.com/forums/forum.jspa?forumID=274

Published 8/25/2009

Table of Contents

Chapter 1. Overview

Welcome to Berkeley DB XML (BDB XML), an embeddable XML database engine that provides support for XQuery access. This document will introduce you to BDB XML's feature set. After reading this document you should have a good understanding of what BDB XML can do for you and how it might be used to manage XML data within your systems and applications. Follow along with the examples and try out BDB XML on your system.

BDB XML is an embedded database specifically designed for the storage and retrieval of XML-formatted documents. Built on the award-winning Berkeley DB, BDB XML provides for efficient queries against millions of XML documents using XQuery. XQuery is a query language designed for the examination and retrieval of portions of XML documents.

Basic Concepts

Typically, BDB XML is used as a library that is linked directly into your application. In addition, BDB XML has a command line shell that allows you to work with XML documents outside of the programming languages that you normally use to interact with BDB XML. You can use the command line shell as part of your application, as a management tool, or simply as a means to explore the features of the product as we do here.

☞ Remember that BDB XML is an *embedded* database engine that supports XML data and queries against that data. This means that in contrast to other database systems, Berkeley DB (and BDB XML) is not a relational database, is not a database server, and it does not support SQL queries. Instead, it is a libarary that is meant to be used directly with your code and which provides XQuery queries against the XML data that you store within the engine.

In BDB XML, all XML data is stored within files called containers. The BDB XML shell provides a simple and convenient way to work with these containers and exposes most of the BDB XML functionality in a friendly, interactive environment, without requiring the use of a programming language.

Containers are really a collection of XML documents and information about those documents. For example, containers include any indexes that are being maintained for the documents.

Containers also store XML documents as either whole documents or as nodes. When containers store whole documents, the XML document is stored all as one unit in the container exactly as it was presented to the system. When documents are stored as nodes, the XML document is deconstructed into smaller pieces – nodes – and those small chunks are what is stored in the container.

For the node storage case, retrieval of the document still returns the document in *mostly* the same formatting state (assuming you didn't modify it) as it was in when it was stored in the container. The only difference is how the document is physically held within the container. Note that node storage typically offers better performance than does whole document storage, and for this reason node storage is the default container type.

☞ In some cases, node storage may change your document slightly. For example, an empty node in your document like this:

```
<node1></node1>
```

Will be stored in the container and returned as this:

```
<node1/>
```

Running the Shell

The shell command is located in the `<path where BDB XML is installed>/bin` directory and is named `dbxml`.

To run the shell, simply type dbxml at the command prompt for your operating system. Assuming that you have the dbxml shell in your operating system's command line path, you'll then be greeted by the `dbxml>` prompt.

```
user> dbxml
dbxml>
```

In the examples that follow, you'll see the `dbxml>` prompt followed by the command that should be entered. Most commands are simple one line commands. However, some are more complicated XQuery examples that will span multiple lines. Each example will show both the command to enter and the resulting output. When the output is too long, ellipsis (...) will be used to abbreviate the intermediate results.

When using BDB XML you will find that document content is stored in a `container`. This is the first basic concept in BDB XML: containers hold collections of XML documents. The documents within a container may or may not share the same schema.

To begin exploring BDB XML, create a container. Our first example models a simple phonebook database. The container's name will be `phone.dbxml`.

```
dbxml> createContainer phone.dbxml

Creating node storage container
```

The command and output in this case was very simple. It was meant to merely confirm command execution. Note that a file named `phone.dbxml` was created in your working directory. This is the new document storage container. Containers hold the XML data, indexes, document metadata, and any other useful information and are managed by BDB XML. Never edit a container directly, always allow the BDB XML library to manage it for you. The '`.dbxml`' extension helps to identify the BDB XML database on disk, but is simply a naming convention that is not strictly required.

☞ In addition to creating the container, the BDB XML shell also automatically opened it and made it ready for us to use.

This phonebook example's data model uses XML entries of the following format:

```
<phonebook>
    <name>
        <first>Tom</first>
        <last>Jones</last>
```

```
        </name>
        <phone type="home">420-203-2032</phone>
</phonebook>
```

Now add a few phone book entries to the container in the following manner:

```
dbxml> putDocument phone1 '<phonebook>
    <name>
        <first>Tom</first>
        <last>Jones</last>
    </name>
    <phone type="home">420-203-2032</phone>
</phonebook>' s

Document added, name = phone1

dbxml> putDocument phone2 '<phonebook>
    <name>
        <first>Lisa</first>
        <last>Smith</last>
    </name>
    <phone type="home">420-992-4801</phone>
    <phone type="cell">390-812-4292</phone>
</phonebook>' s

Document added, name = phone2
```

☞   The XML document content is wrapped in single quote characters and the command is
     terminated by an `s` character. This indicates that we are adding a new document using a
     string. The single quote characters are used for any command parameter that either
     contains spaces or needs to span multiple lines.

Now the container has a few phonebook entries. The following few examples demonstrate
some basic XQuery queries based solely on XPath statements. Subsequent sections will
demonstrate more complex XQuery statements.

☞   XPath is a central part of the XQuery specification. It serves much the same function as
     the SELECT statement does in SQL. It is essentially used to identify a subset of data within
     the data set.

To retrieve all the last names stored in the container:

```
dbxml> query '
collection("phone.dbxml")/phonebook/name/last/string()'

2 objects returned for eager expression '
collection("phone.dbxml")/phonebook/name/last/string()'

dbxml> print
Jones
Smith
```

☞ The `string()` function does not return the targeted node, but instead returns the string value of the targeted node.

To find Lisa's home phone number:

```
dbxml> query '
collection("phone.dbxml")/phonebook[name/first = "Lisa"]/phone[@type =
"home"]/string()'

1 objects returned for eager expression '
collection("phone.dbxml")/phonebook[name/first = "Lisa"]/phone[@type =
"home"]/string()'

dbxml> print
420-992-4801
```

To find all phone numbers in the 420 area code:

```
dbxml> query '
collection("phone.dbxml")/phonebook/phone[starts-with(., "420")]/string()'

2 objects returned for eager expression '
collection("phone.dbxml")/phonebook/phone[starts-with(., "420")]/string()'

dbxml> print
420-203-2032
420-992-4801
```

These queries simply retrieve subsets of data, just like a basic SELECT statement would in a relational database. Each query consists of two parts. The first part of the query identifies the set of documents to be examined (equivalent to a projection). This is done with an XQuery navigation function such as `collection()`. In this example, `collection("phone.dbxml")` specifies the container against which we want to apply our query. The second part is an XPath statement (equivalent to a selection). The first example's XPath statement was `/phonebook/name/last/string()` which, based on our document structure, will retrieve all last names and present them as a string.

Understanding XPath is the first step toward understanding XQuery.

☞ You can perform a query against multiple containers using the union operator ("|") with the `collection()` function. For example, to query against containers `c1.dbxml` and `c2.dbxml`, you would use the following expression:

```
(collection("c1.dbxml") | collection("c2.dbxml"))/name/string()
```

Getting Help

The BDB XML shell has a built in help facility, simply type `help` at the command line:

☞ The text displayed by the `help` command changes frequently, and so what you see in your version of the BDB XML shell may differ from what is shown below.

```
dbxml> help

Command Summary
---------------

#                 - Comment. Does nothing
abort             - Aborts the current transaction
addAlias          - Add an alias to the default container
addIndex          - Add an index to the default container
commit            - Commits the current transaction, and starts a new one
compactContainer  - Compact a container to shrink it's size
contextQuery      - Execute query expression using the last results as
                    the context item
cquery            - Execute an expression in the context of the default
                    container
createContainer   - Creates a new container, which becomes the default
                    container
debug             - Debug the given query expression, or the default
                    pre-parsed query
debugOptimization - Debug optimization command -- internal use only
delIndex          - Delete an index from the default container
echo              - Echo to output
getDocuments      - Gets document(s) by name from default container
getMetaData       - Get a metadata item from the named document
help              - Print help information.  Use 'help commandName' for
                    extended help
info              - Get info on default container
listIndexes       - List all indexes in the default container
lookupEdgeIndex   - Performs an edge index lookup in the default container
lookupIndex       - Performs an index lookup in the default container
lookupStats       - Look up index statistics on the default container
openContainer     - Opens a container, and uses it as the default
                    container
preload           - Pre-loads (opens) a container
prepare           - Prepare the given query expression as the default
                    pre-parsed query
print             - Prints most recent results, optionally to a file
putDocument       - Insert a document into the default container
query             - Execute the given query expression, or the default
                    pre-parsed query
queryPlan         - Prints the query plan for the specified query
                    expression
quit              - Exit the program
reindexContainer  - Reindex a container, optionally changing index type
removeAlias       - Remove an alias from the default container
removeContainer   - Removes a container
removeDocument    - Remove a document from the default container
run               - Runs the given file as a script
setAutoIndexing   - Set auto-indexing state of the default container
```

```
setBaseUri        - Set/get the base uri in the default context
setIgnore         - Tell the shell to ignore script errors
setLazy           - Sets lazy evaluation on or off in the default context
setMetaData       - Set a metadata item on the named document
setNamespace      - Create a prefix->namespace binding in the default
                    context
setProjection     - Enables or disables the use of the document
                    projection optimization
setQueryTimeout   - Set a query timeout in seconds in the default context
setReturnType     - Sets the return type on the default context
setTypedVariable  - Set a variable to the specified type in the default
                    context
setVariable       - Set a variable in the default context
setVerbose        - Set the verbosity of this shell
sync              - Sync current container to disk
time              - Wrap a command in a wall-clock timer
transaction       - Create a transaction for all subsequent operations
                    to use
upgradeContainer - Upgrade a container to the current container format
```

Any given command has additional detailed help. For example:

```
dbxml> help createContainer


createContainer -- Creates a new container, which becomes the default
container

Usage: createContainer <containerName> [n|in|d|id] [[no]validate]
Creates a new default container; the old default is closed.
The default is to create a node storage container, with node indexes.
A second argument of "d" creates a Wholedoc storage container, and
"id" creates a document storage container with node indexes.
A second argument of "n" creates a node storage container, and
"in" creates a node storage container with node indexes.
The optional third argument indicates whether or not to validate
documents on insertion
A containerName of "" creates an in-memory container.
This command uses the XmlManager::createContainer() method.
```

The help text has valuable information about the command and the API calls that are used to implement a particular command. This helps you to find the relevant section of the API documentation where more detail is available and also serves as a way to explore a commonly used subset of the API calls in an interactive fashion.

☞   The debug command provides a debugging facility for XQueries. However, it is currently only useful for developers with a deep understanding of query plans. That said, you can try out the debugger by typing:

```
debug xquery
```

where *xquery* is the query that you want to debug. (XQueries are described in the next chapter of this manual).

Once in the debugger, you can get debugger-specific help by using the `help` command. Use the `step` command to step forward one instruction in the query plan. Use `run` to execute the remainder of the query plan.

Chapter 2. XQuery and Berkeley DB XML

This section steps through some of the XQuery functionality provided by BDB XML and then introduces a few of the facilities BDB XML provides that make working with XML highly efficient. Those unfamiliar with XQuery should first review one of the many excellent XQuery tutorials listed at the end of this document before proceeding.

Adding Data

In this example, the container will manage a few thousand documents modeling an imaginary parts database. Begin by using the following command to create a container called `parts.dbxml`:

```
dbxml> createContainer parts.dbxml

Creating node storage container with nodes indexed
```

A successful response indicates that the container was created on disk, opened, and made the default container within the current context of the shell.

Before we continue, we need to turn off a default behavior of BDB XML. We do this here so that we can make some points later about XQuery performance. We'll explain this later, but for now, simply enter the command:

```
dbxml> setAutoIndexing off

Set auto-indexing state to off, was on
```

Next populate the container with 100000 XML documents that have the following basic structure:

```
<part number="999">
    <description>Description of 999</description>
    <category>9</category>
</part>
```

Some of the documents will provide additional complexity to the database and have the following structure:

```
<part number="990">
   <description>Description of 990</description>
   <category>0</category>
   <parent-part>0</parent-part>
</part>
```

Use the following `putDocument` command to insert the sample data into the new parts container.

☞ Depending on the speed of your machine, you may want to reduce the total number of documents you add to your container for performance reasons. We use a moderately sized document set here so that we are better able to observe timing results later in this chapter. If you are using slow hardware, you should be able to observe the same results using a smaller document set size.

```
dbxml> putDocument "" '
for $i in (0 to 99999)
return
  <part number="{$i}">
    <description>Description of {$i}</description>
    <category>{$i mod 10}</category>
    {
      if (($i mod 10) = 0)
      then <parent-part>{$i mod 3}</parent-part>
      else ""
    }
  </part>' q
```

As the query executes, one line will be printed for each document inserted into the database.

Queries Involving Document Structure

Notice that the parts container can contain documents with different structures. The ability to manage structured data in a flexible manner is one of the fundamental differences between XML and relational databases. In this example, a single container manages documents of two different structures sharing certain common elements. The fact that the documents partially overlap in structure allows for efficient queries and common indexes. This can be used to model a union of related data. Structural queries exploit such natural unions in XML data. Here are some example structural queries.

First select all `part` records containing `parent-part` nodes in their document structure. In english, the following XQuery would read: "from the container named `parts` select all `part` elements that also contain a `parent-part` element as a direct child of that element". As XQuery code, it is:

```
dbxml> query '
collection("parts.dbxml")/part[parent-part]'

10000 objects returned for eager expression '
collection("parts.dbxml")/part[parent-part]'
```

To examine the query results, use the 'print' command:

```
dbxml> print
<part number="0"><description>Description of 0</description>
<category>0</category><parent-part>0</parent-part></part>
<part number="10"><description>Description of 10</description>
<category>0</category><parent-part>0</parent-part></part>
...
<part number="99980"><description>Description of 99980</description>
<category>0</category><parent-part>0</parent-part></part>
<part number="99990"><description>Description of 99990</description>
<category>0</category><parent-part>0</parent-part></part>
```

To display only the parent-part element without displaying the rest of the document, the query changes only slightly:

```
dbxml> query '
collection("parts.dbxml")/part/parent-part'

10000 objects returned for eager expression '
collection("parts.dbxml")/part/parent-part'

dbxml> print
<parent-part>0</parent-part>
<parent-part>1</parent-part>
<parent-part>2</parent-part>
...
<parent-part>1</parent-part>
<parent-part>2</parent-part>
<parent-part>0</parent-part>
```

Alternately, to retrieve the value of the parent-part element, the query becomes:

```
dbxml> query '
collection("parts.dbxml")/part/parent-part/string()'

10000 objects returned for eager expression '
collection("parts.dbxml")/part/parent-part/string()'

dbxml> print
0
1
2
...
1
2
0
```

Invert the earlier example to select all documents that do not have parent-part elements:

```
dbxml> query '
collection("parts.dbxml")/part[not(parent-part)]'

90000 objects returned for eager expression '
collection("parts.dbxml")/part[not(parent-part)]'

dbxml> print
...
<part number="99989"><description>Description of 99989</description>
<category>9</category></part>
<part number="99991"><description>Description of 99991</description>
<category>1</category></part>
<part number="99992"><description>Description of 99992</description>
```

```
<category>2</category></part>
<part number="99993"><description>Description of 99993</description>
<category>3</category></part>
<part number="99994"><description>Description of 99994</description>
<category>4</category></part>
<part number="99995"><description>Description of 99995</description>
<category>5</category></part>
<part number="99996"><description>Description of 99996</description>
<category>6</category></part>
<part number="99997"><description>Description of 99997</description>
<category>7</category></part>
<part number="99998"><description>Description of 99998</description>
<category>8</category></part>
<part number="99999"><description>Description of 99999</description>
<category>9</category></part>
```

Structural queries are somewhat like relational joins, except that they are easier to express and manage over time. Some structural queries are even impossible or impractical to model with more traditional relational databases. This is in part due to the nature of XML as a self describing, yet flexible, data representation. Collections of XML documents attain commonality based on the similarity in their structures just as much as the similarity in their content. Essentially, relationships are implicitly expressed within the XML structure itself. The utility of this feature becomes more apparent when you start combining structural queries with value based queries.

Value Queries

XQuery is equally adept at finding data based on value. The following examples combine structural queries with restrictions on the values returned in the result.

To select all `part`s that have a `parent-part` as a child and also have a `parent-part` value of 1:

```
dbxml> query '
collection("parts.dbxml")/part[parent-part = 1]'

3333 objects returned for eager expression '
collection("parts.dbxml")/part[parent-part = 1]'
```

Notice that the query is identical to the query used in the previous example, except that it uses '[parent-part = 1]'. The results follow:

```
dbxml> print
...
<part number="99820"><description>Description of 99820</description>
<category>0</category><parent-part>1</parent-part></part>
<part number="99850"><description>Description of 99850</description>
<category>0</category><parent-part>1</parent-part></part>
<part number="99880"><description>Description of 99880</description>
<category>0</category><parent-part>1</parent-part></part>
<part number="99910"><description>Description of 99910</description>
```

```
<category>0</category><parent-part>1</parent-part></part>
<part number="99940"><description>Description of 99940</description>
<category>0</category><parent-part>1</parent-part></part>
<part number="99970"><description>Description of 99970</description>
<category>0</category><parent-part>1</parent-part></part>
```

XQuery also provides a full set of expressions that you can use to select documents from the container. For instance, if we wanted to look up the parts with part numbers 1070 and 1032 we could run the following query:

☞ This query is searching on the value of an attribute rather than the value of an element. This is an equally valid way to search for documents.

```
dbxml> query '
collection("parts.dbxml")/part[@number = 1070 or @number = 1032]'

2 objects returned for eager expression '
collection("parts.dbxml")/part[@number = 1070 or @number = 1032]'

dbxml> print
<part number="1070"><description>Description of 1070</description>
<category>0</category><parent-part>2</parent-part></part>
<part number="1032"><description>Description of 1032</description>
<category>2</category></part>
```

Standard inequality operators and other expressions are also available and help to isolate the required subset of data within a container:

```
dbxml> query '
collection("parts.dbxml")/part[@number > 100 and @number < 105]'

4 objects returned for eager expression '
collection("parts.dbxml")/part[@number > 100 and @number < 105]'

dbxml> print
<part number="101"><description>Description of 101</description>
<category>1</category></part>
<part number="102"><description>Description of 102</description>
<category>2</category></part>
<part number="103"><description>Description of 103</description>
<category>3</category></part>
<part number="104"><description>Description of 104</description>
<category>4</category></part>
```

Introducing Indices

One major advantage of modern native XML databases is their ability to index the XML documents they contain. Proper use of indexes can significantly reduce the time required to execute a particular XQuery expression. The previous examples likely executed in a perceptible amount of time, because BDB XML was evaluating each and every document in the container against the query. Without indexes, BDB XML has no choice but to review each document in turn. With

indexes, BDB XML can find a subset of matching documents with a single, or significantly reduced, set of lookups. By carefully applying BDB XML indexing strategies we can improve retrieval performance considerably.

☞ By default, BDB XML turns several useful indexes on so you do not have to worry about them. However, for the purposes of this document we turned them off at the beginning of this chapter. (Using the shell `setAutoIndexing` command). We do this here so we can see relative performance differences between containers with no indexes, and containers with the indexes that we set.

To examine the usefulness of our indexes, we will use the `time` command with each of our queries. This will report how long it takes for each operation to complete.

☞ The following query execution times are relative to the computer and operating system used by the author. Your query times will differ as they depend on many qualities of your system. However, the percentage in improvement in query execution time should be relatively similar.

Recall the first structural query:

```
time query '
collection("parts.dbxml")/part[parent-part]'
10000 objects returned for eager expression '
collection("parts.dbxml")/part[parent-part]'

Time in seconds for command 'query': 0.437096
```

Notice the query execution time. This query takes almost a half a second to execute because the query is examining each document in turn as it searches for the presence of a parent-part element. To improve our performance, we want to specify an index that allows BDB XML to identify the subset of documents containing the `parent-part` element without actually examining each document.

Indices are specified in four parts: path type, node type, key type, and uniqueness. This query requires an index of the node elements to determine if something is present or not. Because the pattern is not expected to be unique, we do not want to turn on uniqueness. Therefore, the BDB XML index type that we should use is `node-element-presence-none`.

```
dbxml> addIndex "" parent-part node-element-presence-none
Adding index type: node-element-presence-none to node: {}:parent-part

dbxml> time query '
collection("parts.dbxml")/part[parent-part]'
10000 objects returned for eager expression '
collection("parts.dbxml")/part[parent-part]'
```

Our query time improved from .4 seconds to .2 seconds. As containers grow in size or complexity, indexes increase performance even more dramatically.

The previous index will also improve the performance of the value query designed to search for the value of the `parent-part` element. But for better results, we should index the node as

a double value. (You use double here instead of decimal because the XQuery specification indicates that implicit numerical casts should be cast to double).

To do this, use a `node-element-equality-double` index.

```
dbxml> time query '
collection("parts.dbxml")/part[parent-part = 1]'
3333 objects returned for eager expression '
collection("parts.dbxml")/part[parent-part = 1]'

Time in seconds for command 'query': 0.511752

dbxml> addIndex "" parent-part node-element-equality-double

Adding index type: node-element-equality-decimal to node: {}:parent-part
dbxml> time query '
collection("parts.dbxml")/part[parent-part = 1]'
3333 objects returned for eager expression '
collection("parts.dbxml")/part[parent-part = 1]'

Time in seconds for command 'query': 0.070674
```

Additional indexes will improve performance for the other value queries.

```
dbxml> time query '
collection("parts.dbxml")/part[@number > 100 and @number < 105]'
4 objects returned for eager expression
'collection("parts.dbxml")/part[@number > 100 and @number < 105]'

Time in seconds for command 'query': 5.06106
```

At over 5 seconds there is plenty of room for improvement. To improve our range query, we can provide an index for the number attribute:

```
dbxml>  addIndex "" number node-attribute-equality-double

Adding index type: node-attribute-equality-double to node: {}:number

dbxml> time query '
collection("parts.dbxml")/part[@number > 100 and @number < 105]'
4 objects returned for eager expression '
collection("parts.dbxml")/part[@number > 100 and @number < 105]'

Time in seconds for command 'query': 3.33212
```

As you can see, proper use of indexes can dramatically effect query performance.

☞ We mentioned at the beginning of this section that we had turned auto indexing off. If we had left it on, the container would have automatically had the following indexes:

```
node-element-string-equality
node-attribute-string-equality
```

```
                 node-element-double-equality
                 node-attribute-double-equality
```

These indexes would have been added for all attribute and leaf nodes. For this example, the indexes would have been added for the `<description>`, `<category>` and `<parent-part>` nodes. They would have also been added for the `number` attribute on the `<part>` node.

Reshaping the Result

XQuery is also useful when reshaping XML content. A common use for this feature is to restructure data into a display oriented dialect of XML, such as XHTML for presentation in a web browser.

Again, begin with the same value query seen earlier, modify it using XQuery and generate an XHTML version of the result suitable for display in a web browser:

```
dbxml> query '<html><body>
    <ul>
        {
        for $part in
            (collection("parts.dbxml")/part[@number > 100 and @number < 105])
        return
            <li>{$part/description/string()}</li>
        }
    </ul></body></html>'
1 objects returned for eager expression '<html><body>
    <ul>
        {
        for $part in
            (collection("parts.dbxml")/part[@number > 100 and @number < 105])
        return
            <li>{$part/description/string()}</li>
        }
    </ul></body></html>'

dbxml> print
<html><body><ul>
<li>Description of 101</li>
<li>Description of 102</li>
<li>Description of 103</li>
<li>Description of 104</li>
</ul></body></html>
```

The following shows the previous HTML as displayed in a web browser:

This XQuery introduces the XQuery FLWOR expression (For, Let, While, Order by, Return — sometimes written as FLWR or FLOWR). Note that XPath is still used in the query. Now, however, it is part of the overall FLWOR structure.

☞ Processing XML data in containers for display in dynamic web sites is best done using the language APIs most suitable to your web development rather than the command line tool we're using for examples.

Sorting the Result

The 'O' in FLWOR stands for 'order by'. The previous XQuery expression did not contain explicit ordering instructions, and so the results were presented based on its order in the container. Over time, as the document set changes, the data will not maintain a constant order. Adding an explicit order by clause to the XQuery statement allows us to implement strict ordering:

```
dbxml> query '<html><body>
    <ul>
        {
        for $part in
            (collection("parts.dbxml")/part[@number > 100 and @number < 105])
        order by xs:decimal($part/@number) descending
        return
            <li>{$part/description/string()}</li>
        }
    </ul></body></html>'
1 objects returned for eager expression '<html><body>
    <ul>
        {
        for $part in
            (collection("parts.dbxml")/part[@number > 100 and @number < 105])
        order by xs:decimal($part/@number) descending
        return
            <li>{$part/description/string()}</li>
        }
    </ul></body></html>'

dbxml> print
<html><body><ul>
<li>Description of 104</li>
<li>Description of 103</li>
<li>Description of 102</li>
```

```
<li>Description of 101</li>
</ul></body></html>
```

The following shows the previous HTML as displayed in a web browser:

The parts are now ordered in descending order, as expected.

Working with Data from Multiple Containers

An application may use one or more containers. BDB XML and XQuery provides excellent support for this situation. First, create a second container and add some additional data. A few simple documents will be enough to demonstrate this feature. To begin, we add them the new container:

```
dbxml> createContainer components.dbxml

Creating node storage container with nodes indexed

dbxml> putDocument component1 '<component number="1">
<uses-part>89</uses-part>
<uses-part>150</uses-part>
<uses-part>899</uses-part>
</component>'

Document added, name = component1

dbxml> putDocument component2 '<component number="2">
<uses-part>901</uses-part>
<uses-part>87</uses-part>
<uses-part>189</uses-part>
</component>'

Document added, name = component2

dbxml> preload parts.dbxml

dbxml> preload components.dbxml
```

These new documents are intended to represent a larger component consisting of several of the parts defined earlier. To output an XHTML view of all the components and their associated parts across containers, use:

```
dbxml> query '<html><body>
  <ul>
    {
      for $component in collection("components.dbxml")/component
      return
        <li>
          <b>Component number: {$component/@number/string()}</b><br/>
          {
            for $part-ref in $component/uses-part
            return
              for $part in collection("parts.dbxml")/part[@number =
                $part-ref cast as xs:decimal]
              return
                <p>{$part/description/string()}</p>
          }
        </li>
    }
  </ul>
</body></html>'
1 objects returned for eager expression '<html><body>
  <ul>
    {
      for $component in collection("components.dbxml")/component
      return
        <li>
          <b>Component number: {$component/@number/string()}</b><br/>
          {
            for $part-ref in $component/uses-part
            return
              for $part in collection("parts.dbxml")/part[@number =
                $part-ref cast as xs:decimal]
              return
                <p>{$part/description/string()}</p>
          }
        </li>
    }
  </ul>
</body></html>'
```

☞ This query will take advantage of one of the indexes we created earlier. XQuery assigns the variable `$part-ref` the very general XPath `number` type. The index we defined earlier applies only to `decimal` values which is a more specific numeric type than `number`. To get the query to use that index we need to provide some help to the query optimizer by using the `cast as xs:decimal` clause. This provides more specific type information about the data we are comparing. If we do not use this, the query optimizer cannot use the decimal index because the type XQuery is using and the type of the index is using do not match.

The output of the query, reformatted for readability, is:

```
dbxml> print
<html><body>
    <ul>
        <li>
            <b>Component number: 1</b><br/>
            <p>Description of 89</p>
            <p>Description of 150</p>
            <p>Description of 899</p>
        </li>
        <li>
            <b>Component number: 2</b><br/>
            <p>Description of 901</p>
            <p>Description of 87</p>
            <p>Description of 189</p>
        </li>
    </ul>
</body></html>
```

The following shows the previous HTML as displayed in a web browser:
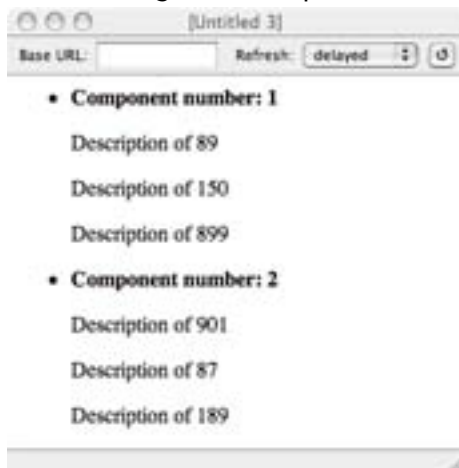
The BDB XML container model provides a great deal of flexibility because there is no specific XML schema associated with a container. XML documents of varying structures can coexist in a single container . Alternatively, separate containers can contain XML documents that are identical along conceptual lines, or for other purposes. Container and document organization should be tailored to the needs of your application.

Working with Data from a Specific Document

Previous queries have executed against all the documents in a container, but there are cases where access to data in a single document is the goal. It is possible to isolate a single document component based on the name we assigned to it, and then perform XQuery expressions against it alone.

For example, to select the number attribute from a document named `component1` in the `components.dbxml` container:

```
dbxml> query '
doc("components.dbxml/component1")/component/@number'
1 objects returned for eager expression '
doc("components.dbxml/component1")/component/@number'

dbxml> print
{}number="1"
```

☞ The doc function shown here can be used to access XML data external to any BDB XML managed container. For instance, to integrate with a web service that returns XML over HTTP use the doc function to execute that web service and then use the resulting data as part of an XQuery query.

A web service that is able to look up the price of a particular part could be knit into a HTML page as it's built in a single XQuery FLWOR expression, and this page can then be hosted under a normal webserver. It is then possible to access that pricing data using the `doc` function in an XQuery expression.

For example, suppose you had this page, which provide the prices of the parts of our components:

```
<prices>
    <part number="87">29.95</part>
    <part number="89">19.95</part>
    <part number="150">24.95</part>
    <part number="189">5.00</part>
    <part number="899">9.95</part>
    <part number="901">15.00</part>
</prices>
```

And suppose this page was available from at the following location:

```
http://www.oracle.com/fakefile.html
```

In this case, we can enhance our earlier parts query to add prices for all the parts. At the same time we'll also convert it to use an HTML table to display the data.

```
dbxml> query '<html><body>
  <ul>
    {
      for $component in collection("dbxml:components.dbxml")/component
      return
        <li>
          <b>Component number: {$component/@number/string()}</b><br/>
          <table>
          {
            for $part-ref in $component/uses-part
            return
              for $part in collection("dbxml:parts.dbxml")/part[@number =
```

```
                    $part-ref cast as xs:decimal]
                return
                <tr><td>{$part/description/string()}</td>
                <td>{
                    doc("http://www.oracle.com/fakefile.html")//part[
                        @number = $part/@number]/string()
                }</td></tr>
            }
            </table>
        </li>
    }
  </ul>
</body></html>'
1 objects returned for eager expression '<html><body>
  <ul>
    {
      for $component in collection("dbxml:components.dbxml")/component
      return
        <li>
          <b>Component number: {$component/@number/string()}</b><br/>
          <table>
          {
            for $part-ref in $component/uses-part
            return
              for $part in collection("dbxml:parts.dbxml")/part[@number =
                    $part-ref cast as xs:decimal]
                return
                <tr><td>{$part/description/string()}</td>
                <td>{
                  doc("http://www.oracle.com/fakefile.html")//part[
                        @number = $part/@number]/string()
                }</td></tr>
          }
          </table>
        </li>
    }
  </ul>
</body></html>'
```

And the result with formatting for readability:

```
dbxml> print
<html>
    <body>
        <ul>
            <li>
                <b>Component number: 1</b>
                <br/>
                <table>
```
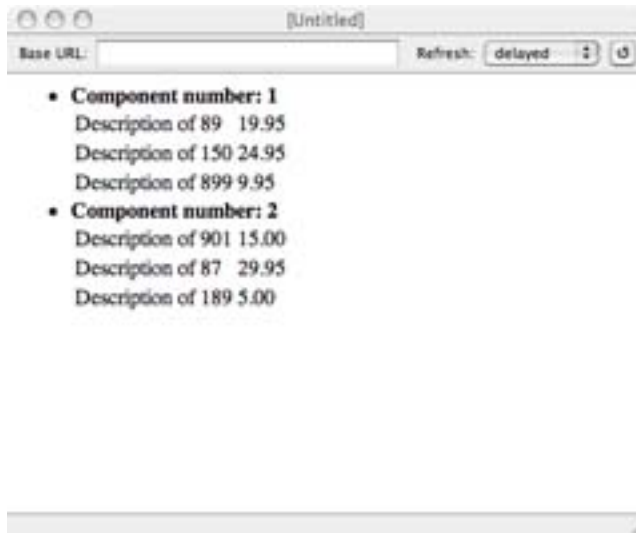
```
                        <tr>
                            <td>Description of 89</td>
                            <td>19.95</td>
                        </tr>
                        <tr>
                            <td>Description of 150</td>
                            <td>24.95</td>
                        </tr>
                        <tr>
                            <td>Description of 899</td>
                            <td>9.95</td>
                        </tr>
                    </table>
                </li>
                <li>
                    <b>Component number: 2</b>
                    <br/>
                    <table>
                        <tr>
                            <td>Description of 901</td>
                            <td>15.00</td>
                        </tr>
                        <tr>
                            <td>Description of 87</td>
                            <td>29.95</td>
                        </tr>
                        <tr>
                            <td>Description of 189</td>
                            <td>5.00</td>
                        </tr>
                    </table>
                </li>
            </ul>
        </body>
    </html>
```

The following shows the previous HTML as displayed in a web browser:

This ability to bring in data from outside BDB XML as part of any query from a web service or other source of XML data provides tremendous power and flexibility when building applications.

Using Metadata

Metadata is data about data. That is, it provides additional information about a document that isn't really part of that document. For example, documents added to the `components.dbxml` container were given a name. Each name represents metadata about each individual document. Other common metadata might include the time a document was modified or the name of the person who modified it. In addition, there are cases when modifying the actual document is not possible and additional data is required to track desired information about the document. As an example, you may be required to keep track of what user last altered a document within a container, and you may need to do this in a way that does not modify the document itself. For this reason, BDB XML stores metadata separately from the document, while still allowing you to perform indexed searches against the metadata as if it were actually part of the document.

To add custom metadata to a document, use the `setMetaData` command.

```
dbxml> openContainer components.dbxml

dbxml> setMetaData component1 '' modifyuser string john

MetaData item 'modifyuser' added to document component1

dbxml> setMetaData component2 '' modifyuser string mary

MetaData item 'modifyuser' added to document component2
```

In BDB XML metadata names are of the form:

```
URI:name
```

where the `URI`. is some unique name that can be common to multiple metadata items, and the `name` is a name associated with the metadata. The combination of the URI and the name must represent a unique identifier for your container.

In the example, above, a `URI` is not used because a `URI` is optional so long as the `name` uniquely identifies the metadata within the container. If you wanted to use `modifyuser` for more than one type of metadata, then you would be required to provide a unique `URI` for both types of metadata.

Note that all metadata defined by BDB XML uses the `dbxml` URI. So, for example, all documents have metadata which identifies the document name. This metadata is defined by BDB XML, and it has the identifier: `dbxml:name`.

To query metadata, use the built-in extension function `dbxml:metadata`. For example:

```
dbxml> query '
collection("components.dbxml")/component[dbxml:metadata("modifyuser")="john"]'

1 objects returned for eager expression '
collection("components.dbxml")/component[dbxml:metadata("modifyuser")="john"]'

dbxml> print
<component number="1">
<uses-part>89</uses-part>
<uses-part>150</uses-part>
<uses-part>899</uses-part>
</component>
```

Notice how the metadata doesn't actually appear in the result document. The metadata is not part of the document; it exists only within the container and with respect to a particular document. If you retrieve the document from BDB XML and transfer it to another system, the metadata will not be included. This is useful when you need to preserve the original state of a document, but also want to track some additional information while it's stored within BDB XML.

Modifying Documents

The best way to modify a document stored in BDB XML is to use XQuery Update statements. Update statements allow you to insert, delete, replace and rename information in an XML document. In this section, we provide a very brief overview of this aspect of the XQuery language.

Note that if you use update statements on a document stored in a whole document container, then you might lose some of your document's formatting. This is because update statements reparse the documents they operate upon and then ultimately store them back in the container in the format used for node storage containers. For this reason, if the formatting of your XML documents are very important to you, you should avoid using XQuery Update Statements on your documents.

☞ The rules surrounding the usage of update statements are somewhat complex. For a brief overview of those rules, see the Getting Started with Berkeley DB XML for C++ [http://www.oracle.com/technology/documentation/berkeley-db/xml/gsg_xml/cxx/BerkeleyDBXML-CXX-GSG.pdf] or Getting Started with Berkeley DB XML for Java [http://www.oracle.com/technology/documentation/berkeley-db/xml/gsg_xml/java/BerkeleyDBXML-JAVA-GSG.pdf] guide. For a complete description of update statements, see the *XQuery Update Facility 1.0* specification. Note that this specification is currently a "candidate specification". BDB XML implements the draft dated 1 August 2008 [http://www.w3.org/TR/xquery-update-10/] specification.

First, let's create a container and then a couple of documents that we can use for our Update queries:

```
dbxml> createContainer modify.dbxml
Creating node storage container with nodes indexed

dbxml> putDocument "mod1.xml" '<mod1>
    <nodeOne>Sample text</nodeOne>
    <nodeTwo>
        <nodeTwoOne>Two One text</nodeTwoOne>
        <nodeTwoTwo>Two Two text</nodeTwoTwo>
        <nodeTwoThree>Two Three text</nodeTwoThree>
    </nodeTwo>
    <nodeThree>Node three text</nodeThree>
</mod1>' s
Document added, name = mod1.xml

dbxml> putDocument "mod2.xml" '<mod2>
    <nodeA>A sample text</nodeA>
    <nodeB>
        <nodeBA>B A text</nodeBA>
        <nodeBB>B B text</nodeBB>
        <nodeBC>B C text</nodeBC>
    </nodeB>
</mod2>' s
Document added, name = mod2.xml
```

Now let's insert some content to `mod1.xml`. There's a few basic rules that are good to keep in mind at this point:

1. Update queries never return a result; they just modify the document and then quit.

2. The queries that you use to select a document for updating must not themselves be an update query.

3. Update queries can only work on one document at a time, although they can be used in FLOWR expressions and so operate on multiple documents and containers as the expression iterates.

Here's how you insert a node, specifying it as a simple text argument on the query:

```
dbxml> query 'insert nodes
        <newNode>Some new text</newNode>
        after
        doc("dbxml:/modify.dbxml/mod1.xml")/mod1/nodeOne'
0 objects returned for eager expression 'insert nodes
        <newNode>Some new text</newNode>
        after
        doc("dbxml:/modify.dbxml/mod1.xml")/mod1/nodeOne'
```

Notice that no documents were returned as a result of the query. This is required by the XQuery Update specification.

Also, notice the keyword `after` in the query. This causes the new data to be inserted after the selected node. Other keywords are available: `before`, `into`, `as first into` and `as last into`.

The results of this query is:

```
dbxml> query 'collection("modify.dbxml")/mod1'
1 objects returned for eager expression 'collection("modify.dbxml")/mod1'

dbxml> print
<mod1>
    <nodeOne>Sample text</nodeOne><newNode>Some new text</newNode>
    <nodeTwo>
        <nodeTwoOne>Two One text</nodeTwoOne>
        <nodeTwoTwo>Two Two text</nodeTwoTwo>
        <nodeTwoThree>Two Three text</nodeTwoThree>
    </nodeTwo>
    <nodeThree>Node three text</nodeThree>
</mod1>
```

It is also possible to identify new content using a selection query, instead of providing it as a string. For example:

```
dbxml> query 'insert nodes
        doc("dbxml:/modify.dbxml/mod2.xml")/mod2/nodeB
        before
        doc("dbxml:/modify.dbxml/mod1.xml")/mod1/nodeThree'
0 objects returned for eager expression 'insert nodes
        doc("dbxml:/modify.dbxml/mod2.xml")/mod2/nodeB
        before
        doc("dbxml:/modify.dbxml/mod1.xml")/mod1/nodeThree'
```

As a result of this query, we now have:

```
dbxml> query 'collection("modify.dbxml")/mod1'
1 objects returned for eager expression 'collection("modify.dbxml")/mod1'


dbxml> print
<mod1>
```

```
    <nodeOne>Sample text</nodeOne><newNode>Some new text</newNode>
    <nodeTwo>
        <nodeTwoOne>Two One text</nodeTwoOne>
        <nodeTwoTwo>Two Two text</nodeTwoTwo>
        <nodeTwoThree>Two Three text</nodeTwoThree>
    </nodeTwo>
    <nodeB>
        <nodeBA>B A text</nodeBA>
        <nodeBB>B B text</nodeBB>
        <nodeBC>B C text</nodeBC>
    </nodeB><nodeThree>Node three text</nodeThree>
</mod1>
```

You can delete a node, and when you do the selected node and all its children are also deleted:

```
dbxml> query 'delete nodes
        doc("dbxml:/modify.dbxml/mod1.xml")/mod1/nodeB'
0 objects returned for eager expression 'delete nodes
        doc("dbxml:/modify.dbxml/mod1.xml")/mod1/nodeB'


dbxml> query 'collection("modify.dbxml")/mod1'
1 objects returned for eager expression 'collection("modify.dbxml")/mod1'


dbxml> print
<mod1>
    <nodeOne>Sample text</nodeOne><newNode>Some new text</newNode>
    <nodeTwo>
        <nodeTwoOne>Two One text</nodeTwoOne>
        <nodeTwoTwo>Two Two text</nodeTwoTwo>
        <nodeTwoThree>Two Three text</nodeTwoThree>
    </nodeTwo>
    <nodeThree>Node three text</nodeThree>
</mod1>
```

You can also rename a node:

```
dbxml> query 'rename node
        doc("dbxml:/modify.dbxml/mod1.xml")/mod1/nodeTwo
        as "renamedTwo"'
0 objects returned for eager expression 'rename node
        doc("dbxml:/modify.dbxml/mod1.xml")/mod1/nodeTwo
        as "renamedTwo"'


dbxml> query 'collection("modify.dbxml")/mod1'
1 objects returned for eager expression 'collection("modify.dbxml")/mod1'
```

```
dbxml> print
<mod1>
    <nodeOne>Sample text</nodeOne><newNode>Some new text</newNode>
    <renamedTwo>
        <nodeTwoOne>Two One text</nodeTwoOne>
        <nodeTwoTwo>Two Two text</nodeTwoTwo>
        <nodeTwoThree>Two Three text</nodeTwoThree>
    </renamedTwo>
    <nodeThree>Node three text</nodeThree>
</mod1>
```

Finally, you can replace document content. Here you can either replace an entire node, or just the node's content. Also, just as is the case with content insertion, you can either specify the new content as a string, or use a selection query to obtain the content from some other document.

To replace a node:

```
dbxml> query 'replace node
        doc("dbxml:/modify.dbxml/mod1.xml")/mod1/nodeOne
        with
        doc("dbxml:/modify.dbxml/mod2.xml")/mod2/nodeB'
0 objects returned for eager expression 'replace node
        doc("dbxml:/modify.dbxml/mod1.xml")/mod1/nodeOne
        with
        doc("dbxml:/modify.dbxml/mod2.xml")/mod2/nodeB'


dbxml> query 'collection("modify.dbxml")/mod1'
1 objects returned for eager expression 'collection("modify.dbxml")/mod1'


dbxml> print
<mod1>
    <nodeB>
        <nodeBA>B A text</nodeBA>
        <nodeBB>B B text</nodeBB>
        <nodeBC>B C text</nodeBC>
    </nodeB><newNode>Some new text</newNode>
    <renamedTwo>
        <nodeTwoOne>Two One text</nodeTwoOne>
        <nodeTwoTwo>Two Two text</nodeTwoTwo>
        <nodeTwoThree>Two Three text</nodeTwoThree>
    </renamedTwo>
    <nodeThree>Node three text</nodeThree>
</mod1>
```

To replace a node's contents, use `replace value of node` instead of `replace node`:

```
dbxml> query 'replace value of node
        doc("dbxml:/modify.dbxml/mod1.xml")/mod1/nodeB
        with
        "replacement text"'
0 objects returned for eager expression 'replace value of node
        doc("dbxml:/modify.dbxml/mod1.xml")/mod1/nodeB
        with
        "replacement text"'


dbxml> query 'collection("modify.dbxml")/mod1'
1 objects returned for eager expression 'collection("modify.dbxml")/mod1'


dbxml> print
<mod1>
    <nodeB>replacement text</nodeB><newNode>Some new text</newNode>
    <renamedTwo>
        <nodeTwoOne>Two One text</nodeTwoOne>
        <nodeTwoTwo>Two Two text</nodeTwoTwo>
        <nodeTwoThree>Two Three text</nodeTwoThree>
    </renamedTwo>
    <nodeThree>Node three text</nodeThree>
</mod1>
```

Note that with the exception of deleting nodes, none of the previous update statements are correct if they match more than one node. That is, they all assume that they are operating on one and only one node. (In fact, BDB XML will return an error if they do match more than one node).

If you wanted to match multiple nodes with your update statements, use a FLWOR statement. For example, the previous replace statement will always return without error if you use a FLOWR statement like this:

```
dbxml> query '
for $i in doc("dbxml:/modify.dbxml/mod1.xml")/mod1/nodeB return
replace value of node $i with "replacement text"'
0 objects returned for eager expression '
 in doc("dbxml:/modify.dbxml/mod1.xml")/mod1/nodeB return
replace value of node $i with "replacement text"'
```

It is possible to create an update function that groups multiple update statements together in a single query. When this is done, there are series of rules that govern the order in which the statements are applied to the targeted document, and the conditions under which an error will automatically be raised. See the Getting Started with Berkeley DB XML for C++ [http://www.oracle.com/technology/documentation/berkeley-db/xml/gsg_xml/cxx/BerkeleyDBXML-CXX-GSG.pdf] or Getting Started with Berkeley DB XML for Java [http://www.oracle.com/technology/documentation/berkeley-db/xml/gsg_xml/java/BerkeleyDBXML-JAVA-GSG.pdf] guide for a description of those rules.

For example, notice that in the following updating function, both the `rename` and `replace` operations operate on the same target. However, the `rename` appears *before* the `replace` operation. If these operations were applied strictly in the order that they are supplied, then the `replace` should fail because the node that it operates on has been replaced. However, this is not the case. The reason why is that XQuery Update always applies `replace` operations to a document before it applies `rename` operations.

```
dbxml> query 'declare updating function
    local:myUpdate($target as element(),
                    $repVal as xs:string,
                    $rep as xs:string)
{
    rename node $target as $rep,
    replace value of node $target with $repVal
};

local:myUpdate(
    doc("dbxml:/modify.dbxml/mod1.xml")/mod1/nodeB,
    "nodeZ content",
    "nodeZ")'
0 objects returned for eager expression 'declare updating function
     local:myUpdate($target as element(),
                    $repVal as xs:string,
                    $rep as xs:string)
{
    rename node $target as $rep,
    replace value of node $target with $repVal
};

local:myUpdate(
    doc("dbxml:/modify.dbxml/mod1.xml")/mod1/nodeB,
    "nodeZ content",
    "nodeZ")'


dbxml> query 'collection("modify.dbxml")/mod1'
1 objects returned for eager expression 'collection("modify.dbxml")/mod1'


dbxml> print
<mod1>
    <nodeZ>nodeZ content</nodeZ><newNode>Some new text</newNode>
    <renamedTwo>
        <nodeTwoOne>Two One text</nodeTwoOne>
        <nodeTwoTwo>Two Two text</nodeTwoTwo>
        <nodeTwoThree>Two Three text</nodeTwoThree>
    </renamedTwo>
    <nodeThree>Node three text</nodeThree>
</mod1>
```

Finally, while it is not possible to perform an update and return a result in the same operation, it *is* possible to create a function that creates a copy of a document, modifies the copy, and then returns that copy to you. Note that this copy is not stored permanently in the container; the update operations are simply applied to the result of copy operation. All other rules applying to Update operations still apply.

These sort of functions are called *transform functions*. For example:

```
dbxml> query 'copy $c := doc("dbxml:/modify.dbxml/mod1.xml")/mod1
modify (rename node $c/nodeZ as "nodeB",
        replace value of node $c/nodeZ with "replacement text")
return $c'
1 objects returned for eager expression 'copy $c :=
doc("dbxml:/modify.dbxml/mod1.xml")/mod1
modify (rename node $c/nodeZ as "nodeB",
        replace value of node $c/nodeZ with "replacement text")
return $c'


dbxml> print
<mod1>
    <nodeB>replacement text</nodeB><newNode>Some new text</newNode>
    <renamedTwo>
        <nodeTwoOne>Two One text</nodeTwoOne>
        <nodeTwoTwo>Two Two text</nodeTwoTwo>
        <nodeTwoThree>Two Three text</nodeTwoThree>
    </renamedTwo>
    <nodeThree>Node three text</nodeThree>
</mod1>
```

However, if we query for mod1.xml, we see that it has not been modified by the copy and modify operation:

```
dbxml> query 'collection("modify.dbxml")/mod1'
1 objects returned for eager expression 'collection("modify.dbxml")/mod1'


dbxml> print
<mod1>
    <nodeZ>nodeZ content</nodeZ><newNode>Some new text</newNode>
    <renamedTwo>
        <nodeTwoOne>Two One text</nodeTwoOne>
        <nodeTwoTwo>Two Two text</nodeTwoTwo>
        <nodeTwoThree>Two Three text</nodeTwoThree>
    </renamedTwo>
    <nodeThree>Node three text</nodeThree>
</mod1>
```

Schema Constraints

XML documents can optionally be validated against a schema to enforce document similarity. Most databases support schema constraints, but BDB XML has the unique ability to store collections of data with schemas that vary from document to document if desired. This is an added level of functionality not commonly found in XML databases.

Recall our phonebook example. The documents for that example had the following structure:

```
<phonebook>
    <name>
        <first>Tom</first>
        <last>Jones</last>
    </name>
    <phone type="home">420-203-2032</phone>
</phonebook>
```

Three things are required to validate this document within BDB XML. First, a schema is required. Because the subject of XML schemas are well beyond the scope of this document, we simply provide one for you here. There are many excellent books and tutorial web sites on the subject, and we suggest you review some of that material if you are not familiar with XML schemas.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="phonebook">
      <xs:complexType>
        <xs:sequence>
            <xs:element name="name" minOccurs="1" maxOccurs="1">
              <xs:complexType>
                <xs:sequence>
                    <xs:element name="first" type="xs:string"/>
                    <xs:element name="last" type="xs:string"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
            <xs:element name="phone" minOccurs="0" maxOccurs="unbounded">
              <xs:complexType>
                <xs:simpleContent>
                  <xs:extension base="xs:string">
                    <xs:attribute name="type" type="xs:string"/>
                  </xs:extension>
                </xs:simpleContent>
              </xs:complexType>
            </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
</xs:schema>
```

Suppose this schema is available from a web-server at:

```
http://www.oracle.com/fakeschema.xsd
```

Second, we need to create a container with validation enabled.

```
dbxml> createContainer validate.dbxml d validate
Creating document storage container, with validation
```

Third, we need to attach the schema to a document and insert it into the container.

```
dbxml> putDocument phone1 '
<phonebook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation=
        "http://www.oracle.com/fakeschema.xsd">
    <name>
        <first>Tom</first>
        <last>Jones</last>
    </name>
    <phone type="home">420-203-2032</phone>
</phonebook>' s

Document added, name = phone1
```

That document was successfully added because it conforms to the schema. Now, try to add an invalid document.

```
dbxml> putDocument phone2 '
<phonebook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation=
        "http://www.oracle.com/fakeschema.xsd">
    <name>
        <first>Tom</first>
        <last>Jones</last>
    </name>
    <phone type="home">420-203-2032</phone>
    <cell-phone>430-201-2033</cell-phone>
</phonebook>' s

stdin:67: putDocument failed, Error: XML Indexer:  Parse error in document at
line, 10, char 17. Parser message: Unknown element 'cell-phone']]>
```

Since the schema doesn't define the `cell-phone` element and we have schema validation
enabled, BDB XML won't allow the document to be added to the container.

XML schemas provide a powerful tool for constraining the structure and content of XML
documents.

The Berkeley DB XML API

The Berkeley DB XML command line shell is a tool and not an end-user application, it has been
useful in exploring the features of this system. Applications will be built using the programming

language APIs. In this final example, we implement our first example, the phonebook example, in C++.

```cpp
#include <string>
#include <fstream>
#include "dbxml/DbXml.hpp"

using namespace std;
using namespace DbXml;

int
main(int argc, char **argv)
{
    try {
        XmlManager mgr;

        // Create the phonebook container
        XmlContainer cont = mgr.createContainer("phone.dbxml");

        // Add the phonebook entries to the container
        XmlUpdateContext uc = mgr.createUpdateContext();
        cont.putDocument("phone1", "<phonebook><name><first>Tom</first>
<last>Jones</last></name><phone type=\"home\">420-203-2032</phone>
</phonebook>", uc);
        cont.putDocument("phone2", "<phonebook><name><first>Lisa</first>
<last>Smith</last></name><phone type=\"home\">420-992-4801</phone>
<phone type=\"cell\">390-812-4292</phone></phonebook>",
uc);

        // Run an XQuery against the phonebook container
        XmlQueryContext qc = mgr.createQueryContext();
        XmlResults res =
        mgr.query("collection('phone.dbxml')/phonebook[name/first = 'Lisa']/
phone[@type = 'home']/string()", qc);

        // Print out the result of the query
        XmlValue value;
        while (res.next(value))
            cout << "Value: " << value.asString() << endl;
    } catch (XmlException &e) {
        std::cout << "Exception: " << e.what() << std::endl;
    }
    return 0;
}
```

While this example is in C++, the BDB XML API is similar across all supported languages. This makes it easy to transfer knowledge about the API between languages and can enable useful scenarios such as prototyping the application in Python and then implementing the final version

in Java or C++. Because of the similarity across languages porting, the BDB XML code is relatively simple.

Chapter 3. Wrapping Up

As you explore BDB XML further and begin to write applications, you should read the "Getting Started Guide for Berkeley DB XML". That guide contains much more detail about all the topics covered in this introduction and more. BDB XML has many more advanced features that are of interest when building real applications.

Benefits

When choosing a XML database for your application, consider all the qualities you've observed of BDB XML. Also consider the foundations of this technology. The Berkeley DB database engine is a proven scalable transactional system with all the mature features you'd expect and likely require in your application.

On top of this solid database foundation, BDB XML offers a solid XQuery implementation that offers the ability to retrieve documents, modify existing documents, and format the query output as needed.

In addition, BDB XML provides indexed queries and whole or node level document storage within containers. W3C XML schemas can be used to validate individual documents or all documents stored within BDB XML containers. Schema validation is enabled per container and the schema used is specified as part of the document being stored. This provides great flexibility in how you utilize schemas, including allowing you to store XML with no associated schema.

Moreover, because BDB XML is a native XML database that stores XML data in its native format, it maintains the same extensible structure that has attracted many developers to XML. It is this flexibility that makes BDB XML a better choice than relational database offerings that must translate XML data into internal tables and rows, thus locking the data into a static schema while paying a heavy penalty in processing overhead when documents are reconstituted from tables and rows.

XML Features

BDB XML is implemented to conform to the W3C standards for XML, XML Namespaces, and the latest available XQuery standards. The following additional features specifically designed to support XML data management and queries go above and beyond any existing standard, and serve to set BDB XML further ahead of similar solutions:

- Containers: a single file that contains one or more XML documents, and their metadata and indexes.

- Indexes: quickly identify subsets of documents that match specific queries, thus allowing for improved query performance against the corresponding XML data set.

- Integrity: documents are stored (and retrieved) in their native format with all whitespace preserved.

- Metadata: each document stored in BDB XML can have "data about the data" associated with the document.

Database Features

Berkeley DB XML inherits a great many features from Berkeley DB. These features put it years ahead of the competition and makes it an ideal candidate for mission-critical applications that must manage XML data.

Important features that BDB XML inherits from Berkeley DB are:

- In-process data access. BDB XML is compiled in the same way as any library. It runs in the same process space as your application. The result is database support in a small footprint without the IPC-overhead required by traditional client/server-based database implementations.

- Ability to manage databases up to 256 terabytes in size.

- Database environment support. BDB XML environments support all of the same features as Berkeley DB environments, including multiple databases, shared data cache, transactions, deadlock detection, lock and page control, and encryption. In particular, this means that BDB XML databases can share an environment with Berkeley DB databases, thus allowing an application to gracefully use both.

- Atomic operations. Complex sequences of read and write access can be grouped together into a single atomic operation using BDB XML's transaction support. Either all of the read and write operations within a transaction succeed, or none of them succeed.

- Isolated operations. Operations performed inside a transaction see all XML documents as if no other transactions are currently operating on them.

- Recoverability. BDB XML's transaction support ensures that all committed data is available no matter how the application or system might subsequently fail.

- Concurrent access. Through the combined use of isolation mechanisms built into BDB XML, plus deadlock handling supplied by the application, multiple threads and processes can concurrently access the XML data set in a safe manner.

- Replication. BDB XML provides the ability to distribute updates made to a master database to multiple replica databases. This provides the application with the ability to support fail-over for High Availability applications, as well as scalability for load balancing of queries across multiple systems.

Languages and Platforms

The official BDB XML distribution provides the library in the C++, Java, Perl, Python, PHP, and Tcl languages. Because BDB XML is available under an open source license, a growing list of third-parties are providing BDB XML support in languages other than those that are officially supported.

BDB XML is supported on a very large number of platforms. Check with the BDB XML forum for the latest news on supported platforms, as well as for information as to whether your preferred language provides BDB XML support.

Chapter 4. Where to Learn More

Berkeley DB XML Resources

- BDB XML product information page [http://www.oracle.com/technology/software/products/berkeley-db/xml/index.html]

- BDB XML documentation [http://www.oracle.com/technology/documentation/berkeley-db/xml]

- Getting Started with Berkeley DB XML for C++ [http://www.oracle.com/technology/documentation/berkeley-db/xml/gsg_xml/cxx/BerkeleyDBXML-CXX-GSG.pdf]

- Getting Started with Berkeley DB XML for Java [http://www.oracle.com/technology/documentation/berkeley-db/xml/gsg_xml/java/BerkeleyDBXML-JAVA-GSG.pdf]

- Berkeley DB XML Getting Started with Transaction Processing for C++ [http://www.oracle.com/technology/documentation/berkeley-db/xml/gsg_xml_txn/cxx/BerkeleyDBXML-Txn-CXX.pdf]

- Berkeley DB XML Getting Started with Transaction Processing for Java [http://www.oracle.com/technology/documentation/berkeley-db/xml/gsg_xml_txn/java/BerkeleyDBXML-Txn-JAVA.pdf]

- Berkeley DB XML Programer's Reference Guide [http://www.oracle.com/technology/documentation/berkeley-db/xml/ref_xml/toc.html]

- Berkeley DB XML C++ API [http://www.oracle.com/technology/documentation/berkeley-db/xml/api_cxx/frame.html]

- Berkeley DB XML Javadoc [http://www.oracle.com/technology/documentation/berkeley-db/xml/java/index.html]

- BDB XML example code, which is available in your distribution in the following directory:

  ```
  <dbxml-distribution>/dbxml/examples
  ```

  In the examples directory, there are subdirectories for various programming languages including C++, Java, Python, PHP and Perl.

XML Resources

- XML Specification [http://www.w3.org/TR/xml/]

- Namespaces in XML Specification [http://www.w3.org/TR/xml-names/]

- O'Reilly's XML.com [http://www.xml.com]

- IBM developerWorks XML [http://www.ibm.com/developerworks/xml/]

XQuery Resources

- XQuery 1.0 Specification [http://www.w3.org/TR/xquery/]

- XQuery Update Facility 1.0 Specification [http://www.w3.org/TR/xquery-update-10/]

- XPath 2.0 Specification [http://www.w3.org/TR/xpath20/]

- XML.com *What is XQuery?* [http://www.xml.com/pub/a/2002/10/16/xquery.html]

- XML.com *Practical XQuery Column* [http://www.xml.com/pub/at/28]

- IBM developerWorks *An introduction to XQuery* [http://www.ibm.com/developerworks/xml/library/x-xquery.html]