Introduction to Berkeley DB XML

SLEEPYCAT SOFTWARE

Makers of Berkeley DB

Legal Notice

To obtain a copy of this document's original source code, please write to <support@sleepycat.com>.

Published 10/21/2005

Table of Contents

1. Overview 1
    - Basic Concepts 1
    - Running the Shell 1
    - Getting Help 4
2. XQuery and Berkeley DB XML 7
    - Adding Data 7
    - Queries Involving Document Structure 8
    - Value Queries 10
    - Introducing Indices 11
    - Reshaping the Result 14
    - Sorting the Result 15
    - Working with Data from Multiple Containers 16
    - Working with Data from a Specific Document 19
    - Using Metadata 23
    - Modifying Documents 24
    - Schema Constraints 26
    - The Berkeley DB XML API 28
3. Wrapping Up 30
    - Benefits 30
    - XML Features 30
    - Database Features 31
    - Languages and Platforms 31
4. Where to Learn More 33
    - Berkeley DB XML Resources 33
    - XML Resources 33
    - XQuery Resources 33

Chapter 1. Overview

Welcome to Berkeley DB XML (BDB XML), a native XML database (NXD) 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.

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.

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 indices 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 appeared on the filesystem. When documents are stored as nodes, the XML document is deconstructed into the smallest pieces – that is, nodes – possible, 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 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.

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 with nodes indexed
```

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, indices, 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/text()'

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

dbxml> print
Jones
Smith
```

To find Lisa's home phone number:

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

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

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")]/text()'
```

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

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/text() which, based on our document structure, will retrieve all last names and present them as text.

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/text()
```

Getting Help

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

```
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
append           - Append to nodes specified in the query expression
commit           - Commits the current transaction, and starts a new one
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 command -- internal use only
delIndex         - Delete an index from the default container
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
insertAfter      - Insert new content after nodes selected by the query
     expression
insertBefore     - Insert new content before nodes selected by the query
     expression
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
print            - Prints most recent results, optionally to a file
putDocument      - Insert a document into the default container
query            - Execute an expression in the context of the XmlManager
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
removeNodes      - Remove content from documents specified by the query
     expression
renameNodes      - Rename nodes specified by the query expression
run              - Runs the given file as a script
setApplyChanges  - Modifies "apply changes" state in the default update
     context
setBaseUri       - Set the base uri in the default context
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
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
transaction      - Create a transaction for all subsequent operations to
     use
updateNodes      - Update node content based on query expression and new
     content
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.

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. Next populate the container with 3000 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.

```
dbxml> putDocument "" '
for $i in (0 to 2999)
return
  <part number="{$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 indices. 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]'

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

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

```
dbxml> print
<part number="540"><description>Description of 540</description>
<category>0</category><parent-part>0</parent-part></part>
<part number="30"><description>Description of 30</description>
<category>0</category><parent-part>0</parent-part></part>
...
<part number="990"><description>Description of 990</description>
<category>0</category><parent-part>0</parent-part></part>
<part number="480"><description>Description of 480</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'

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

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

```
<parent-part>2</parent-part>
<parent-part>2</parent-part>
```

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

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

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

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

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

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

2700 objects returned for eager expression '
collection("parts.dbxml")/part[not(parent-part)]'

dbxml> print
<part number="22"><description>Description of 22</description>
<category>2</category></part>
<part number="1995"><description>Description of 1995</description>
<category>5</category></part>
...
<part number="2557"><description>Description of 2557</description>
<category>7</category></part>
<part number="2813"><description>Description of 2813</description>
<category>3</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 `parts` 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]'

100 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="1840"><description>Description of 1840</description>
<category>0</category><parent-part>1</parent-part></part>
<part number="1330"><description>Description of 1330</description>
<category>0</category><parent-part>1</parent-part></part>
...
<part number="1300"><description>Description of 1300</description>
<category>0</category><parent-part>1</parent-part></part>
<part number="790"><description>Description of 790</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 indices 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 indices, BDB XML has no choice but to review each document in turn. With indices, 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.

To examine the usefulness of our indices, we begin by raising the level of verbosity in the shell:

```
dbxml> setVerbose 2 2
```

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:

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

Query      - Starting eager query execution
Query      - parts.dbxml - U : [3000] 256 512 768 1024 1280 1536 1792 2048
2304 2560 2816 257 513 769 1025 1281 1537 1793 2049 2305 ...
Query      - Finished eager query execution, time taken = 2495.82ms
300 objects returned for eager expression '
collection("parts.dbxml")/part[parent-part]'
```

Notice the query execution time. This query takes almost 2.5 seconds 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> query '
collection("parts.dbxml")/part[parent-part]'

Query      - Starting eager query execution
Query      - parts.dbxml - P(parent-part) : [300] 2 12 22 32 42 52 62 72 82 92
102 112 122 132 142 152 162 172 182 192 ...
Query      - Finished eager query execution, time taken = 173.084ms
300 objects returned for eager expression '
collection("parts.dbxml")/part[parent-part]'
```

Our query time improved from 2.5 seconds to just under 1/5th of a second. As containers grow in size or complexity, indices 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.

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

Query      - Starting eager query execution
Query      - parts.dbxml - P(parent-part) : [300] 2 12 22 32 42 52 62 72 82 92
102 112 122 132 142 152 162 172 182 192 ...
Query      - Finished eager query execution, time taken = 223.821ms
100 objects returned for eager expression '
collection("parts.dbxml")/part[parent-part = 1]'
```

This query time also improved from 2.4 seconds to just over 1/5th of a second. Because the node's content that we are examining involves a number, we can improve our query performance time even more by indexing that node as a decimal value. To do this, use `node-element-equality-decimal`.

```
dbxml> addIndex "" parent-part node-element-equality-decimal

Adding index type: node-element-equality-decimal to node: {}:parent-part

dbxml>  query '
```

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

Query      - Starting eager query execution
Query      - parts.dbxml - V(parent-part,=,'1') : [100] 12 42 72 102 132 162
192 222 252 282 312 342 372 402 432 462 492 522 552 582 ...
Query      - Finished eager query execution, time taken = 69.803ms
100 objects returned for eager expression '
collection("parts.dbxml")/part[parent-part = 1]'
```

With this second index, the query runs in less than 1/10th of a second, or three times faster than without the index.

Additional indices will improve performance for the other value queries.

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

Query      - Starting eager query execution
Query      - parts.dbxml - U : [3000] 256 512 768 1024 1280 1536 1792 2048
2304 2560 2816 257 513 769 1025 1281 1537 1793 2049 2305 ...
Query      - Finished eager query execution, time taken = 6938.48ms
4 objects returned for eager expression '
collection("parts.dbxml")/part[@number > 100 and @number < 105]'
```

At almost 7 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-decimal

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

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

Query      - Starting eager query execution
Query      - parts.dbxml - V(@number,>,'100') : [2899] 103 104 105 106 107
108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 ...
Query      - parts.dbxml - V(@number,<,'105') : [105] 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20 21 ...
Query      - parts.dbxml - n(V(@number,>,'100'),V(@number,<,'105')) : [4]
103 104 105 106
Query      - Finished eager query execution, time taken = 29.967ms
4 objects returned for eager expression '
collection("parts.dbxml")/part[@number > 100 and @number < 105]'
```

This query's execution time has been reduced to less than 1/10 of a second. Proper use of indices can dramatically effect query performance.

BDB XML provides a wide variety of different index types to improve the performance of queries.

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.

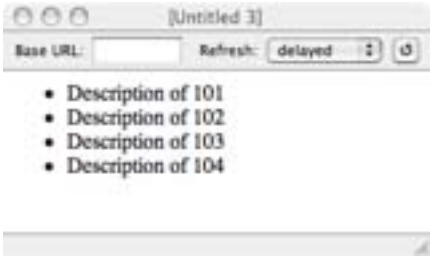
Again, begin with the same value query seen earlier, modify it using XQuery and generate an XHTML version of the result suitable for display in a web browser:

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

Query      - Starting eager query execution
Query      - parts.dbxml - V(@number,>,'100') : [2899] 103 104 105 106 107
108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 ...
Query      - parts.dbxml - V(@number,<,'105') : [105] 2 3 4 5 6 7 8 9 10 11
12 13 14 15 16 17 18 19 20 21 ...
Query      - parts.dbxml - n(V(@number,<,'105'),V(@number,>,'100')) : [4]
103 104 105 106
Query      - Finished eager query execution, time taken = 22.561ms
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/text()}</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/text()}</li>
        }
    </ul></body></html>'

Query      - Starting query execution
Query      - parts.dbxml - R(@number,>,'100',<,'105') : [4] 103 104 105 106
Query      - Finished eager query execution, time taken = 29.869ms
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/text()}</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/text()}</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/text()}</p>
          }
        </li>
    }
  </ul>
</body></html>'

Query      - Starting query execution
Query      - components.dbxml - U : [2] 2 3
Query      - parts.dbxml - V(@number,=,'89') : [1] 91
Query      - parts.dbxml - V(@number,=,'150') : [1] 152
Query      - parts.dbxml - V(@number,=,'899') : [1] 901
Query      - parts.dbxml - V(@number,=,'901') : [1] 903
Query      - parts.dbxml - V(@number,=,'87') : [1] 89
Query      - parts.dbxml - V(@number,=,'189') : [1] 191
Query      - Finished eager query execution, time taken = 19.495ms
1 objects returned for eager expression '<html><body>
  <ul>
    {
      for $component in collection("components.dbxml")/component
      return
        <li>
          <b>Component number: {$component/@number/text()}</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
```

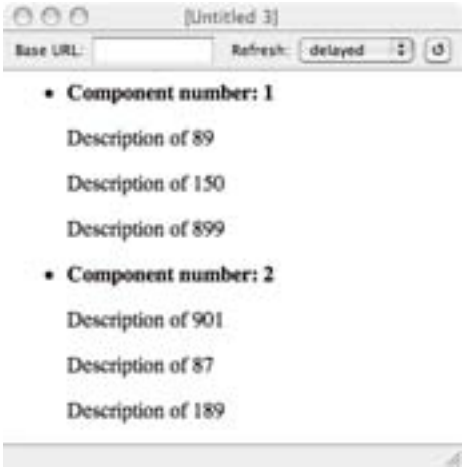
```
                <p>{$part/description/text()}</p>
            }
        </li>
    }
  </ul>
</body></html>'
```

This query will take advantage of one of the indices 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'

Query      - Starting query execution
Query      - components.dbxml - D('component1',U) : [1] 2
Query      - components.dbxml - U : [2] 2 3
Query      - components.dbxml - D('component1',U) : [1] 2
Query      - Finished query execution, time taken = 3.574ms
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 . Sleepycat has such a simulated service set up to support this example. There is an XML file provided by a web server at xml.sleepycat.com. It is possible to access that pricing data using the `doc` function in an XQuery. The URL for the prices file is `http://xml.sleepycat.com/intro2xml/prices.xml`. The content of that file will provide the prices of the parts that make up our components.

The contents of `http://xml.sleepycat.com/intro2xml/prices.xml`. looks something like this:

```
<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>
```

With that done, 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/text()}</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/text()}</td>
                <td>{
                  doc("http://xml.sleepycat.com/intro2xml/prices.xml")//part[
                        @number = $part/@number]/text()
                }</td></tr>
         }
         </table>
      </li>
    }
  </ul>
</body></html>'
```

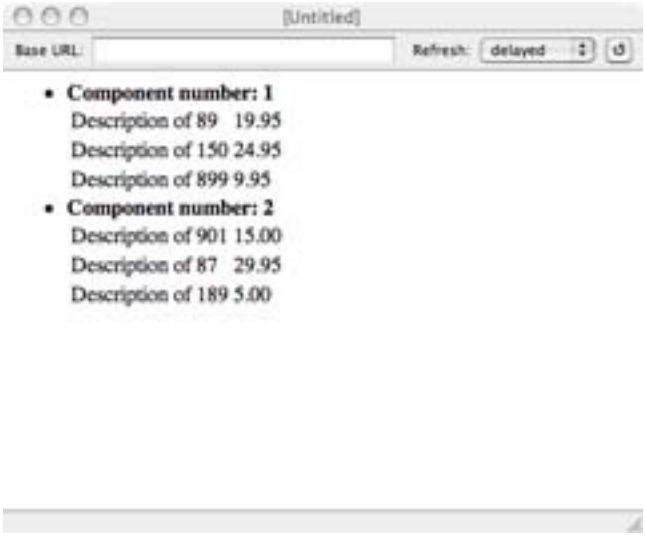
```
Query      - Starting query execution
Query      - components.dbxml - U : [2] 2 3
Query      - parts.dbxml - V(@number,=,'89') : [1] 91
Query      - parts.dbxml - V(@number,=,'150') : [1] 152
Query      - parts.dbxml - V(@number,=,'899') : [1] 901
Query      - parts.dbxml - V(@number,=,'901') : [1] 903
Query      - parts.dbxml - V(@number,=,'87') : [1] 89
Query      - parts.dbxml - V(@number,=,'189') : [1] 191
Query      - Finished query execution, time taken = 2098.29ms
1 objects returned for eager expression '<html><body>
  <ul>
    {
      for $component in collection("dbxml:components.dbxml")/component
      return
        <li>
          <b>Component number: {$component/@number/text()}</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/text()}</td>
                <td>{
                  doc("http://xml.sleepycat.com/intro2xml/prices.xml")//part[
                        @number = $part/@number]/text()
                }</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
```

Metadata is essentially contained within its own, unique namespace (`dbxml:metadata`), so queries against metadata must identify this namespace:

```
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

XQuery as a language does not yet define any way to modify documents stored within an XML database. When it does, BDB XML will add support for this functionality. Meanwhile, BDB XML does include an excellent API for document modification. Using this API it is possible to add new data into an existing document, modify (replace) existing data in the document, and delete data from a document.

In the BDB XML shell, modifications occur in two steps. First, you select the set of documents you want to work with. In this example only the `component1` document is selected:

```
dbxml> query doc('components.dbxml/component1')

1 objects returned for eager expression 'doc('components.dbxml/component1')'
```

Second, specify the modification to make against the query result. For our example, there are many possible modification operations. For this example, we'll simply add a new `uses-part` element to the document:

```
dbxml> append ./component element uses-part '12'

Appending into nodes: ./component an object of type: element with name:
uses-part and content: 12

1 modifications made.

dbxml> print
<?xml version="1.0" encoding="UTF-8" standalone="no" ?><component number="1">
<uses-part>89</uses-part>
```

```
<uses-part>150</uses-part>
<uses-part>899</uses-part>
<uses-part>12</uses-part></component>
```

The append command executes relative to the context of the previous query. That context contains the entire `component1` document because we used the doc function to select it. However, modifications only work against nodes within the document so we had to select the root node of the document with the `./component` XPath statement.

Modifications can be made against the result of any query regardless of the number of documents returned.

```
dbxml> query '
collection("components.dbxml")/component'

2 objects returned for eager expression '
collection("components.dbxml")/component'
```

The query selected all the documents in the container. Now, insert a new `uses-part` element after the last `uses-part` element in each document.

```
dbxml> insertAfter ./uses-part[last()] element uses-part '15'

Inserting after nodes: ./uses-part[last()] an object of type: element with
name: uses-part and content: 15

2 modifications made.

dbxml> print
<component number="1">
<uses-part>89</uses-part>
<uses-part>150</uses-part>
<uses-part>899</uses-part>
<uses-part>12</uses-part><uses-part>15</uses-part></component>
<component number="2">
<uses-part>901</uses-part>
<uses-part>87</uses-part>
<uses-part>189</uses-part><uses-part>15</uses-part>
</component>
```

The modification becomes part of the current context . This allows for cascading modifications. Taking advantage of this, the following will remove the nodes added in the last step using the `removeNodes` command.

```
dbxml> removeNodes 'uses-part[. = 15]'
Removing nodes: uses-part[. = 15]

2 modifications made.

dbxml> print
<component number="1">
```

```
<uses-part>89</uses-part>
<uses-part>150</uses-part>
<uses-part>899</uses-part>
<uses-part>12</uses-part></component>
<component number="2">
<uses-part>901</uses-part>
<uses-part>87</uses-part>
<uses-part>189</uses-part>
</component>
```

Such modifications can also be performed on attributes, processing instructions, comments and text nodes by altering the parameters to the modification operation. The following adds a `version` attribute to the root node of each component.

```
dbxml> append . attribute version '1.0'
Appending into nodes: . an object of type: attribute with name: version and
content: 1.0

2 modifications made.

dbxml> print
<component number="1" version="1.0">
<uses-part>89</uses-part>
<uses-part>150</uses-part>
<uses-part>899</uses-part>
<uses-part>12</uses-part></component>
<component number="2" version="1.0">
<uses-part>901</uses-part>
<uses-part>87</uses-part>
<uses-part>189</uses-part>
</component>
```

Modification operations provide a powerful and simple mechanism for altering XML data without the overhead of removing that data from the database. Large documents can be quickly and easily manipulated when combining this feature with node level storage (the default).

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>
```

This schema is available over the network using the URL `http://xml.sleepycat.com/intro2xml/phonebook.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://xml.sleepycat.com/intro2xml/phonebook.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://xml.sleepycat.com/intro2xml/phonebook.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++.

```
#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']/text()", 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. Berkeley DB XML's layers on top of that solid foundation provide efficient XQuery access, indexed queries, and whole or node level document storage organized 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 indices.
- Indices: 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.
- Modification: a mechanism for modifying documents, which allows for addition, replacement, and deletion of document nodes and elements.

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 by Sleepycat.

BDB XML is supported on a very large number of platforms. Check with the BDB XML mailing lists 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

- Berkeley DB XML product information page
  [http://www.sleepycat.com/products/xml.shtml]
- Sleepycat product documentation [http://www.sleepycat.com/xmldocs/index.html]
- BDB XML C++ Getting Started Guide
  [http://www.sleepycat.com/xmldocs/gsg_xml/cxx/index.html]
- BDB XML Java Getting Started Guide
  [http://www.sleepycat.com/xmldocs/gsg_xml/java/index.html]
- BDB XML Programmers Reference Guide
  [http://www.sleepycat.com/xmldocs/ref_xml/toc.html]

XML Resources

- XML Specification [http://www.w3.org/TR/2004/REC-xml-20040204/]
- Namespaces in XML Specification [http://www.w3.org/TR/REC-xml-names/]
- O'Reilly's XML.com [http://www.xml.com]
- IBM developerWorks XML [http://www-128.ibm.com/developerworks/xml/]

XQuery Resources

- XQuery 1.0 Specification [http://www.w3.org/TR/xquery/]
- 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-106.ibm.com/developerworks/xml/library/x-xquery.html]