Getting Started with Berkeley DB XML for C++

Makers of Berkeley DB

Legal Notice

Published 12/5/2005

Table of Contents

Preface

Welcome to Berkeley DB XML (BDB XML). This document introduces BDB XML, version 2.2. It is intended to provide a rapid introduction to the BDB XML API set and related concepts. The goal of this document is to provide you with an efficient mechanism with which you can evaluate BDB XML against your project's technical requirements. As such, this document is intended for C++ developers and senior software architects who are looking for an in-process XML data management solution. No prior experience with Sleepycat technologies is expected or required.

Conventions Used in this Book

The following typographical conventions are used within in this manual:

Class names are represented in `monospaced font`, as are `method names`. For example: "The `XmlDatabase::openContainer()` method returns an `XmlContainer` class object."

Variable or non-literal text is presented in *italics*. For example: "Go to your *DBXML_HOME* directory."

Program examples are displayed in a monospaced font on a shaded background. For example:

```
#include "DbXml.hpp"

using namespace DbXml;
// exception handling omitted for clarity

int main(void)
{
    // Open an XmlManager.
    XmlManager myManager;
}
```

Chapter 1. Introduction to Berkeley DB XML

Welcome to Sleepycat's Berkeley DB XML (BDB XML). 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.

This document introduces BDB XML. It is intended to provide a rapid introduction to the BDB XML API set and related concepts. The goal of this document is to provide you with an efficient mechanism with which you can evaluate BDB XML against your project's technical requirements. As such, this document is intended for C++ developers and senior software architects who are looking for an in-process XML data management solution. No prior experience with Sleepycat technologies is expected or required.

Note that while this document uses C++ for its examples, the concepts described here should apply equally to all language bindings in which the BDB XML API is available. Be aware that a version of this document also exists for the Java language.

Overview

BDB XML is an embedded database that is tuned for managing and querying hundreds, thousands, or even millions of XML documents. You use BDB XML through a programming API that allows you to manage, query, and modify your documents via an in-process database engine. Because BDB XML is an embedded engine, you compile and link it into your application in the same was as you would any third-party library.

In BDB XML documents are stored in *containers*, which you create and manage using `XmlManager` objects. Each such object can open multiple containers at a time.

Each container can hold millions of documents. For each document placed in a container, the container holds all the document data, any metadata that you have created for the document, and any indices maintained for the documents in the container.

(*Metadata* is information that you associate with your document that might not readily fit into the document schema itself. For example, you might use metadata to track the last time a document was modified instead of maintaining that information from within the actual document.)

XML documents may be stored in BDB XML containers in one of two ways:

- Whole documents.

  Documents are stored in their entirety. This method works best for smaller documents (that is, documents under a megabyte in size).

- As document nodes.

  Documents stored as nodes are broken down into their individual document element nodes and each such node is then stored as an individual record in the container. Along

with each such record, BDB XML also stores all node attributes, and the text node, if any.

This type of storage is best for large XML documents (greater than 1 megabyte in size).

From an API-usage perspective, there are very few differences between whole document and node storage containers. For more information, see Container Types (page 15).

Once a document has been placed in a container, you can use XQuery to retrieve one or more documents. You can also use XQuery to retrieve one or more portions of one or more documents. Queries are performed using `XmlManager` objects. The queries themselves, however, limit the scope of the query to a specified list of containers or documents. documents.

BDB XML supports the entire XQuery working draft. At the time of this printing, the draft is dated July, 2004. However, BDB XML will be updated to track any changes in the working draft that may occur.

Also, because XQuery is an extension to XPath 2.0, BDB XML provides full support for that query language as well.

Finally, BDB XML provides a robust document modification facility that allows you to easily add, delete, or modify selected portions of documents. This means you can avoid writing modification code that manipulates (for example) DOM trees — BDB XML can handle all those details for you.

Benefits

BDB XML provides a series of features that makes it more suitable for storing XML documents than other common XML storage mechanisms. BDB XML's ability to provide efficient indexed queries means that it is a far more efficient storage mechanism than simply storing XML data in the filesystem. And because BDB XML provides the same transaction protection as does Berkeley DB, it is a much safer choice than is the filesystem for applications that might have multiple simultaneous readers and writers of the XML data.

More, because BDB XML stores XML data in its native format, BDB XML enjoys the same extensible schema 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 relational database schema.

XML Features

BDB XML is implemented to conform to the W3C standards for XML, XML Namespaces, and the XQuery working draft. In addition, it offers the following features specifically designed to support XML data management and queries:

- Containers. A container is a single file that contains one or more XML documents, and their metadata and indices. You use containers to add, delete, and modify documents, and to manage indices.

- Indices. BDB XML indices greatly enhance the performance of queries against the corresponding XML data set. BDB XML indices are based on the structure of your XML documents, and as such you declare indices based on the nodes that appear in your documents as well the data that appears on those nodes.

  Note that you can also declare indices against metadata.

- Queries. BDB XML queries are performed using the XQuery 1.0 language. XQuery is a W3C draft specification (http://www.w3.org/XML/Query).

- Query results. BDB XML retrieves documents that match a given XQuery query. BDB XML query results are always returned as a set. The set can contain either matching documents, or a set of values from those matching documents.

- Storage. If you use node-level storage for you documents (see Container Types (page 15)), then BDB XML automatically transcodes your documents to Unicode UTF-8. If you use whole document storage, then the document is stored in whatever encoding that it uses. Note that in either case, your documents must use an encoding supported by Xerces before they can be stored in BDB XML containers.

  Beyond the encoding, documents are stored (and retrieved) in their native format with all whitespace preserved.

- Metadata attribute support. Each document stored in BDB XML can have metadata attributes associated with it. This allows information to be associated with the document without actually storing that information in the document. For example, metadata attributes might identify the last accessed and last modified timestamps for the document.

- Document modification. BDB XML provides a robust mechanism for modifying documents. Using this mechanism, you can add, replace, and delete nodes from your document. This mechanism allows you to modify both element and attribute nodes, as well as processing instructions and comments.

Database Features

Beyond XML-specific features, BDB XML inherits a great many features from Berkeley DB, which allows BDB XML to provide the same fast, reliable, and scalable database support as does Berkeley DB. The result is that BDB XML is 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 and linked 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, 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.

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.

Getting and Using BDB XML

BDB XML exists as a library against which you compile and link in the same way as you would any third-party library. You can download the BDB XML distribution from the Sleepycat DB XML product page [http://www.sleepycat.com/products/xml.shtml].

Documentation and Support

BDB XML is officially described in the Sleepycat product documentation [http://www.sleepycat.com/xmldocs/index.html]. For additional help and for late-breaking news on language and platform support, it is best to use the BDB XML mailing lists. You can find out how to subscribe to these lists from the Berkeley DB XML product information page [http://www.sleepycat.com/products/xml.shtml].

Library Dependencies

BDB XML depends on several external libraries. The result is that build instructions for BDB XML may change from release to release as its dependencies mature. For this reason it is best to check with the installation instructions included with your version of Berkeley DB XML for your library's specific build requirements. These instructions are available from:

```
DBXML_HOME/docs/index.html
```

where *DBXML_HOME* is the location where you unpacked the distribution.

That said, BDB XML currently relies on the following libraries:

- Berkeley DB [http://www.sleepycat.com]. Berkeley DB provides the underlying database support for BDB XML. Currently Berkeley DB version 4.3 is required for BDB XML.
- Xerces [http://xml.apache.org/xerces-c/index.html]. Xerces provides the DOM and SAX support that BDB XML employs for XML data parsing. Xerces 2.6 is required for BDB XML.
- Pathan. BDB XML's XQuery support is built on top of pathan.
- XQuery. BDB XML provides a standalone library that implements the XQuery query language.

Note that the BDB XML package comes with all of the libraries that are required to build and use BDB XML.

Building and Running BDB XML Applications

All BDB XML APIs exist in the `DbXml` namespace.

For information on how to build and run a BDB XML application for your particular platform/compiler, see the build instructions that are available through the `docs` directory in your BDB XML distribution. Alternatively, you can find up-to-date build instructions here:

http://www.sleepycat.com/xmldocs/index.html

Chapter 2. Exception Handling and Debugging

Before continuing, it is helpful to look at exception handling and debugging tools in the BDB XML API.

All BDB XML operations can throw an exception, and so they should be within a `try` block.

BDB XML methods throw `XmlException` objects. BDB XML always re-throws all underlying Berkeley DB exceptions as `XmlException`, so every exception that can be thrown by BDB XML is an `XmlException` instance.

`XmlException` is derived from `std::exception`, so you are only required to catch `std::exception` in order to provide proper exception handling for your BDB XML applications. Of course, you can choose to catch both types of exceptions if you want to differentiate between the two in your error handling or messaging code.

Note that if you are using core Berkeley DB operations with your BDB XML application then you should catch either `DbException` or `std::exception` with this code.

The following example illustrates BDB XML exception handling.

Example 2.1. BDB XML Exception Handling

```
#include "DbXml.hpp"

using namespace DbXml;
int main(void)
{
    // Open an XmlManager and an XmlContainer.
    XmlManager myManager;
    try {
        XmlContainer myContainer =
            myManager.openContainer("container.dbxml");
    } catch (XmlException &xe) {
        // Error handling goes here
    } catch (std::exception &e) {
        // Error handling goes here
    }
}
```

Debugging BDB XML Applications

In some cases, the exceptions thrown by your BDB XML application may not contain enough information to allow you to debug the source of an error. In this case, you can cause BDB XML to issue more information using the error stream.

In order to set up the error stream, you use `set_error_stream()` on the underlying Berkeley DB environment object (see Berkeley DB Environments (page 8) for information on environments):

Example 2.2. Setting Error Streams

```
#include "DbXml.hpp"

using namespace DbXml;

int main(void)
{
    // Open an XmlManager
    XmlManager myManager;
    myManager.getDbEnv()->set_error_stream(std::cerr);
}
```

Once you have set up your error stream, you can control the amount of information that BDB XML reports on that stream using `setLogLevel()` and `setLogCategory`.

`setLogLevel()` allows you to indicate the level of logging that you want to see (debug, info, warning, error, or all of these).

`setLogCategory()` allows you to indicate the portions of DB XML's subsystems for which you want logging messages issued (indexer, query processor, optimizer, dictionary, container, or all of these).

You can call these from anywhere within your DB XML code. For example:

Example 2.3. Setting Log Levels

```
#include "DbXml.hpp"

using namespace DbXml;
int main(void)
{
    // Open an XmlManager and an XmlContainer.
    XmlManager myManager;
    db.getDbEnv()->set_error_stream(std::cerr);
    try {
        XmlContainer myContainer = db.openContainer("container.dbxml");
        DbXml::setLogLevel(DbXml::LEVEL_ALL, true);
        DbXml::setLogCategory(DbXml::CATEGORY_ALL, true);
    } catch (XmlException &xe) {
        // Error handling goes here
    } catch (std::exception &e) {
        // Error handling goes here
    }
}
```

Chapter 3. XmlManager and Containers

While containers are the mechanism that you use to store and manage XML documents, you use `XmlManager` objects to create and open `XmlContainer` objects. We therefore start with the `XmlManager`.

XmlManager

`XmlManager` is a high-level class used to manage many of the objects that you use in a BDB XML application. The following are some of the things you can do with `XmlManager` objects:

- Manage containers. This management includes creating, opening, deleting, and renaming containers (see Managing Containers (page 13)).
- Create input streams used to load XML documents into containers (see Input Streams and Strings (page 18)).
- Create `XmlDocument`, `XmlQueryContext`, and `XmlUpdateContext` objects.
- Prepare and run XQuery queries (see Using XQuery with BDB XML (page 24)).
- Create a transaction object (see Using Transactions (page 80)).

Because `XmlManager` is the only way to construct important BDB XML objects, it is central to your BDB XML application.

Berkeley DB Environments

Before you can instantiate an `XmlManager` object, you have to make some decisions about your Berkeley DB Environment. BDB XML requires you to use a database environment. You can use an environment explicitly, or you can allow the `XmlManager` constructor to manage the environment for you.

If you explicitly create an environment, then you can turn on important features in BDB XML such as logging, transactional support, and support for multithreaded and multiprocess applications. It also provides you with an on-disk location to store all of your application's containers.

If you allow the `XmlManager` constructor to implicitly create and/or open an environment for you, then the environment is only configured to allow multithreaded sharing of the environment and the underlying databases (`DB_PRIVATE` is used). All other features are not enabled for the environment.

The next several sections describe the things you need to know in order to create and open an environment explicitly. We start with this activity first because it is likely to be the first thing you will do for all but the most trivial of BDB XML applications.

Environment Open Flags

In order to use an environment, you must first open it. When you do this, there are a series of flags that you can optionally specify. These flags are bitwise *or*'d together, and they have the effect of enabling important subsystems (such as transactional support).

There are a great many environment open flags and these are described in the Berkeley DB documentation. However, there are a few that you are likely to want to use with your BDB XML application, so we describe them here:

- `DB_CREATE`

  If the environment does not exist at the time that it is opened, then create it. It is an error to attempt to open a database environment that has not been created.

- `DB_INIT_LOCK`

  Initializes the locking subsystem. This subsystem is used when an application employs multiple threads or processes that are concurrently reading and writing Berkeley DB databases. In this situation, the locking subsystem, along with a deadlock detector, helps to prevent concurrent readers/writers from interfering with each other.

  Remember that under the covers BDB XML containers are using Berkeley DB databases, so if you want your containers to be accessible by multiple threads and/or multiple processes, then you should enable this subsystem.

- `DB_INIT_LOG`

  Initializes the logging subsystem. This subsystem is used for database recovery from application or system failures. For more information on normal and catastrophic recovery, see the *Berkeley DB Programmer's Reference Guide* (available with your Berkeley DB distribution).

- `DB_INIT_MPOOL`

  Initializes the shared memory pool subsystem. This subsystem is required for multithreaded BDB XML applications, and it provides an in-memory cache that can be shared by all threads and processes participating in this environment.

- `DB_INIT_TXN`

  Initializes the transaction subsystem. This subsystem provides atomicity for multiple database access operations. When transactions are in use, recovery is possible if an error condition occurs for any given operation within the transaction. If this subsystem is turned on, then the logging subsystem must also be turned on.

  We discuss transactional application in Using Transactions (page 80) later in this manual.

- `DB_RECOVER`

causes normal recovery to be run against the underlying database. Normal recovery ensures that the database files are consistent relative to the operations recorded in the log files. This is useful if, for example, your application experienced an ungraceful shut down and there is consequently an possibility that some write operations were not flushed to disk.

Recovery can only be run if the logging subsystem is turned on. Also, recovery must only be run by a single thread of control; typically it is run by the application's master thread before any other database operations are performed.

Regardless of the flags you decide to set at creation time, it is important to use the same ones on all subsequent environment opens (the exception to this is `DB_CREATE` which is only required to create an environment). In particular, avoid using flags to open environments that were not used at creation time. This is because different subsystems require different data structures on disk, and it is therefore illegal to attempt to use subsystems that were not initialized when the environment was first created.

Opening and Closing Environments

To use an environment, you must first open it. At open time, you must identify the directory in which it resides and this directory must exist prior to the open attempt. At open time, you also specify the open flags, properties, if any, that you want to use for your environment.

When you are done with the environment, you must make sure it is closed. You can either do this explicitly, or you can have the `XmlManager` object do it for you.

If you are explicitly closing your environment, you must make sure an containers opened in the environment have been closed before you close your environment.

For information on `XmlManager` instantiation, see XmlManager Instantiation and Destruction (page 11)

For example:

```
#include "DbXml.hpp"
...
u_int32_t env_flags = DB_CREATE     |  // If the environment does not
                                       // exist, create it.
                      DB_INIT_LOCK  |  // Initialize the locking subsystem
                      DB_INIT_LOG   |  // Initialize the logging subsystem
                      DB_INIT_MPOOL |  // Initialize the cache
                      DB_INIT_TXN;     // Initialize transactions
std::string envHome("/export1/testEnv");
DbEnv myEnv(0);

try {
    myEnv.open(envHome.c_str(), env_flags, 0);
} catch(DbException &e) {
    std::cerr << "Error opening database environment: "
```

```
                << envHome << std::endl;
    std::cerr << e.what() << std::endl;
} catch(std::exception &e) {
    std::cerr << "Error opening database environment: "
                << envHome << std::endl;
    std::cerr << e.what() << std::endl;
}

// Do BDB XML work here


try {
    myEnv.close(0);
} catch(DbException &e) {
    std::cerr << "Error closing database environment: "
                << envHome << std::endl;
    std::cerr << e.what() << std::endl;
} catch(std::exception &e) {
    std::cerr << "Error closing database environment: "
                << envHome << std::endl;
    std::cerr << e.what() << std::endl;
}
```

XmlManager Instantiation and Destruction

You create an `XmlManager` object by calling its constructor. You destroy a `XmlManager` object by calling its destructor, either by using the `delete` operator or by allowing the object to go out of scope (if it was created on the stack). Note that `XmlManager` is closed and all of its resources released when the last open handle to the object is destroyed.

To construct an `XmlManager` object, you may or may not provide the destructor with an open `DbEnv` object. If you do instantiate `XmlManager` with an opened environment handle, then `XmlManager` will close and destroy that `DbEnv` object for you if you specify `DBXML_ADOPT_DBENV` for the `XmlManager` constructor. set `XmlManagerConfig::setAdoptEnvironment()` to `true`.

If you provide an `DbEnv` object to the constructor, then you can use that object to use whatever subsystems that you application may require (see Environment Open Flags (page 9) for some common subsystems).

If you do not provide an environment object, then `XmlManager` will implicitly create an environment for you. In this case, the environment will not be configured to use any subsystems and it is only capable of being shared by multiple threads from within the same process. Also, in this case you must identify the on-disk location where you want your containers to reside using one of the following mechanisms:

- Specify the path to the on-disk location in the container's name.
- Specify the environment's data location using the `DB_HOME` environment variable.

In either case, you can pass the XmlManager constructor a flag argument that controls that object's behavior with regard to the underlying containers (the flag is NOT passed directly to the underlying environment or databases). Valid values are:

- DBXML_ALLOW_AUTO_OPEN

  When specified, XQuery queries that reference created but unopened containers will automatically cause the container to be opened for the duration of the query.

- DBXML_ADOPT_DBENV

  When specified, XmlManager will close and destroy the DbEnv object that it was instantiated with when the XmlManager is closed.

- DBXML_ALLOW_EXTERNAL_ACCESS

  When specified, XQuery queries executed from inside BDB XML can access external sources (URLs, files, and so forth).

For example, to instantiate an XmlManager with a default environment:

```
#include "DbXml.hpp"
...

using namespace DbXml;
int main(void)
{
    XmlManager myManager;   // The manager and underlying
                            // environment are closed when
                            // this goes out of scope.

    return(0);
}
```

And to instantiate an XmlManager using an explicit environment object:

```
#include "DbXml.hpp"
...

using namespace DbXml;
int main(void)
{
    u_int32_t env_flags = DB_CREATE     |  // If the environment does not
                                           // exist, create it.
                          DB_INIT_LOCK  |  // Initialize locking
                          DB_INIT_LOG   |  // Initialize logging
                          DB_INIT_MPOOL |  // Initialize the cache
                          DB_INIT_TXN;     // Initialize transactions

    std::string envHome("/export1/testEnv");
    DbEnv myEnv(0);
```

```
    XmlManager *myManager = NULL;

    try {
        myEnv.open(envHome.c_str(), env_flags, 0);
        myManager =
            new XmlManager(myEnv,
                           DBXML_ADOPT_DBENV); // The manager and
                                               // environemnt is closed
                                               // when this object is
                                               // destroyed.
    } catch(DbException &e) {
        std::cerr << "Error opening database environment: "
                  << envHome << std::endl;
        std::cerr << e.what() << std::endl;
    } catch (XmlException &xe) {
                // Error handling goes here
        std::cerr << "Error opening database environment: "
                  << envHome
                  << " or opening XmlManager." << std::endl;
        std::cerr << xe.what() << std::endl;
    }

    try {
        if (myManager != NULL) {
            delete myManager;
        }
    } catch(DbException &e) {
        std::cerr << "Error closing database environment: "
                << envHome << std::endl;
        std::cerr << e.what() << std::endl;
    } catch(XmlException &xe) {
        std::cerr << "Error closing database environment: "
                << envHome << std::endl;
        std::cerr << xe.what() << std::endl;
    }
}
```

Managing Containers

In BDB XML you store your XML Documents in *containers*. A container is a file on disk that contains all the data associated with your documents, including metadata and indices.

To create and open a container, you use `XmlManager::createContainer()`. Once a container has been created, you can not use `createContainer()` on it again. Instead, simply open it using: `XmlManager::openContainer()`.

Note that you can test for the existence of a container using the `XmlManager::existsContainer()` method. This method should be used on closed containers.

It returns `0` if the named file is not a BDB XML container. Otherwise, it returns the underlying database format number.

Alternatively, you can cause a container to be created and opened by calling `openContainer()` and pass it the necessary `flags` to allow the container to be created (see the following section for information on container open flags).

You can open a container multiple times. Each time you open a container, you receive a reference-counted handle for that container.

You close a container by allowing the container object to go out of scope. Note that the container is not actually closed until the last handle for the container is off the stack.

For example:

```
#include "DbXml.hpp"
...

using namespace DbXml;
int main(void)
{
    XmlManager myManager;   // The manager is closed when
                            // it goes out of scope.

    // Open the container. If it does not currently exist,
    // then create it. This container is closed when the last
    // handle to it goes out of scope.
    XmlContainer myContainer =
        myManager.createContainer("/export/xml/myContainer.bdbxml");

    // Obtain a second handle to the container.
    XmlContainer myContainer2 =
        myManager.openContainer("/export/xml/myContainer.bdbxml");

    return(0);
}
```

Container Flags

When you create or open a container, there are a large number of flags that you can specify which control various aspects of the container's behavior. The following are the flags commonly used by BDB XML applications. For a complete listing of the flags available for use, see the BDB XML API Reference.

As is the case with environment flags, to set multiple flags you must bitwise *or* them together:

```
        DB_CREATE | DB_EXCL
```

- `DB_CREATE`

Causes the container and all underlying databases to be created. It is not necessary to specify this flag on the call to `XmlManager::createContainer()`. In addition, you need specify it for `XmlManager::openContainer()`. only if the container has not already been created.

- `DB_EXCL`

  Causes the container creation to fail if the container already exists. It is not necessary to specify this flag on the call to `XmlManager::createContainer()`. Note that this flag should only be used if `DB_CREATE` is also used.

- `DB_RDONLY`

  The container is opened for read-access only.

- `DBXML_ALLOW_VALIDATION`

  Causes documents to be validated when they are loaded into the container. The default behavior is to not validate documents.

- `DBXML_INDEX_NODES`

  Causes indices for the container to return nodes rather than documents. The default is to index at the document level. This flag has no meaning if the container type is `WholedocContainer` (see below).

  This flag is only meaningful at container creation time; you cannot change the indexing level once the container has been created.

  For more information on index nodes, see Specifying Index Nodes (page 64).

- `DBXML_TRANSACTIONAL`

  The container supports transactions. For more information, see Using Transactions (page 80)

Container Types

At creation time, every container must have a type defined for it. This container type identifies how XML documents are stored in the container. As such, the container type can only be determined at container creation time; you cannot change it on subsequent container opens.

Containers can have one of the following types specified for them:

- Wholedoc Containers

  The container contains entire documents; the documents are stored "as is" without any manipulation of line breaks or whitespace. To cause the container to hold whole documents, set `XmlContainer::WholedocContainer` on the call to `XmlManager::createContainer()`.

- Node containers

  XmlDocuments are stored as individual nodes in the container. That is, each record in the underlying database contains a single leaf node, its attributes and attribute values if any, and its text nodes, if any. BDB XML also keeps the information it needs to reassemble the document from the individual nodes stored in the underlying databases.

  This is the default, and preferred, container type.

  To cause the documents to be stored as individual nodes, set XmlContainer::NodeContainer on the call to XmlManager::createContainer().

- Default container type.

  The default container type is used. You can set the default container type using XmlManager::setDefaultContainerType(). If you never set a default container type, then the container will use node-level storage.

Note that NodeContainer is generally faster to query than is WholedocContainer. On the other hand, WholedocContainer provides faster document loading times into the container than does NodeContainer because BDB XML does not have to deconstruct the document into its individual leaf nodes. For the same reason, WholedocContainer is faster at retrieving whole documents for the same reason — the document does not have to be reassembled.

Because of this, you should use NodeContainer unless one of the following conditions are true:

- Load performance is more important to you than is query performance.
- You want to frequently retrieve the entire XML document (as opposed to just a portion of the document).
- Your documents are so small in size that the query advantage offered by NodeContainer is negligible or vanishes entirely. The size at which this threshold is reached is of course dependent on the physical resources available to your application (memory, CPU, disk speeds, and so forth).

Note that you should avoid using WholedocContainer if your documents tend to be greater than a megabyte in size. WholedocContainer is tuned for small documents and you will pay increasingly heavy performance penalties as your documents grow larger.

For example:

```
#include "DbXml.hpp"
...

using namespace DbXml;
int main(void)
{
    XmlManager myManager;   // The manager is closed when
                            // it goes out of scope.
```

```
    myManager.setDefaultContainerType(XmlContainer::WholedocContainer);

    // Create and open the container.
    XmlContainer myContainer =
        myManager.createContainer("/export/xml/myContainer.bdbxml");
    return(0);
}
```

Deleting and Renaming Containers

You can delete a container using `XmlManager::removeContainer()`. It is an error to attempt to remove an open container.

You can rename a container using `XmlManager::renameContainer()`. It is an error to attempt to rename an open container.

For example:

```
#include "DbXml.hpp"
...

using namespace DbXml;
int main(void)
{
    XmlManager myManager;   // The manager is closed when
                            // it goes out of scope.

    // Assumes the container currently exists.
    myManager.renameContainer("/export/xml/myContainer.bdbxml",
                              "/export2/xml/myContainer.bdbxml");


    myManager.removeContainer("/export2/xml/myContainer.bdbxml");

    return(0);
}
```

Chapter 4. Adding XML Documents to Containers

To manage XML documents in BDB XML, you must load them into a container. Typically you will do this by using the `XmlContainer` handle directly. You can also load a document into an `XmlDocument` instance, and then load that instance into the container using the `XmlContainer` handle. This book will mostly use the first, most direct, method.

Input Streams and Strings

When you add a document to a container, you must identify the location where the document resides. You can do this by using:

- A string object that holds the entire document.
- An input stream that is created from a filename. Use `XmlManager::createLocalFileInputStream()` to create the input stream.
- An input stream created from a URL. In this case, the URL can be any valid URL. However, if the URL requires network activity in order to access the identified content (such as is required if you, for example, supply an HTTP URL), then the input stream is valid only if you have compiled Xerces with socket support.

  Use `XmlManager::createURLInputStream()` to create the input stream.
- An input stream that refers to a memory buffer.

  Use `XmlManager::createMemBufInputStream()` to create the input stream.
- An input stream that refers to standard input (the console under Windows systems).

  Use `XmlManager::createStdInInputStream()` to create the input stream.

Note that BDB XML does not validate an input stream until you actually attempt to put the document to your container. This means that you can create an input stream to an invalid location or to invalid content, and BDB XML will not throw an exception until you actually attempt to load data from that location.

We provide an example of creating input streams in the following section.

Adding Documents

To add a document to a container, you use `XmlContainer::putDocument()`. When you use this method, you must:

1. Create an input stream to the content, or load the XML document into a string object. Alternatively, you can create an `XmlDocument` object, set the input stream or string to that object, and then provide the `XmlDocument` object to `XmlContainer::putDocument()`.

2. Provide a name for the document. This name must be unique or BDB XML will throw `XmlException::UNIQUE_ERROR`.

   If you are using an `XmlDocument` object to add the document, use `XmlDocument::setName()` to set the document's name. Otherwise, you can set the name directly on the call to `XmlContainer::putDocument()`.

   Note that if you do not want to explicitly set a name for the document, you can set a flag, `DBXML_GEN_NAME`, on the call to `XmlContainer::putDocument()`. This causes BDB XML to generate a unique name for you. The name that it generates is a concatenation of a unique value, an underscore, and the value that you provide for the document's name, if any. For example:

   ```
   myDocName_a
   ```

   where `myDocName` is the name that you set for the document and `a` is the unique value generated by BDB XML.

   If you do not set a name for the document, but you do specify that a unique name is to be generated, then `dbxml` is used as the name's prefix.

   ```
   dbxml_b
   ```

   If you do not set a name for the document and if you do not use `DBXML_GEN_NAME`, then BDB XML throws `XmlException::UNIQUE_ERROR`.

3. Create an `XmlUpdateContext` object. This object encapsulates the context within which the container is updated. Reusing the same object for a series of puts against the same container can improve your container's write performance.

Note that the content that you supply to `XmlContainer::putDocument()` must contain well-formed XML. Also, note that while your documents are stored in the container with their shared text entities (if any) as-is, the underlying XML parser does attempt to expand them for indexing purposes. Therefore, you must make sure that any entities contained in your documents are resolvable at load time.

For example, to add a document that is held in a string:

```
#include "DbXml.hpp"
...

using namespace DbXml;
int main(void)
{
    // The document
    std::string docString = "<a_node><b_node>Some text</b_node></a_node>";

    // The document's name.
    std::string docName = "testDoc1";

    // Get a manager object.
```

```
    XmlManager myManager;   // The manager is closed when
                            // it goes out of scope.

    // Load the document in its entirety. The document's formatting is
    // preserved.
    myManager.setDefaultContainerType(XmlContainer::WholedocContainer);

    // Open the container. The container is closed when it goes
    // out of scope.
    XmlContainer myContainer =
        myManager.openContainer("container.bdbxml");

    // Need an update context for the put.
    XmlUpdateContext theContext = myManager.createUpdateContext();

    // Put the document
    try {
        myContainer.putDocument(docName,      // The document's name
                                docString,    // The actual document,
                                              // in a string.
                                theContext,   // The update context
                                              // (required).
                                0);           // Put flags.
    } catch (XmlException &e) {
        // Error handling goes here. You may want to check
        // for XmlException::UNIQUE_ERROR, which is raised
        // if a document with that name already exists in
        // the container. If this exception is thrown,
        // try the put again with a different name, or
        // use XmlModify to update the document.
    }

    return(0);
}
```

To load the document from an input stream, the code is identical except that you use the appropriate method on `XmlManager` to obtain the stream. For example, to load an XmlDocument directly from a file on disk:

```
#include "DbXml.hpp"
...

using namespace DbXml;
int main(void)
{
    // The document
    std::string fileName = "/export/testdoc1.xml";

    // The document's name.
```

```
    std::string docName = "testDoc1";

    // Get a manager object.
    XmlManager myManager;   // The manager is closed when
                            // it goes out of scope.

    // Load the document in its entirety. The document's formatting is
    // preserved.
    myManager.setDefaultContainerType(XmlContainer::WholedocContainer);

    // Open the container. The container is closed when it goes
    // out of scope.
    XmlContainer myContainer =
        myManager.openContainer("container.bdbxml");

    // Need an update context for the put.
    XmlUpdateContext theContext = myManager.createUpdateContext();

    try {
        // Get the input stream.
        XmlInputStream *theStream =
            myManager.createLocalFileInputStream(fileName);

        // Put the document
        myContainer.putDocument(docName,      // The document's name
                                theStream,   // The actual document.
                                theContext,  // The update context
                                             // (required).
                                0);          // Put flags.
    } catch (XmlException &e) {
        // Error handling goes here. You may want to check
        // for XmlException::UNIQUE_ERROR, which is raised
        // if a document with that name already exists in
        // the container. If this exception is thrown,
        // try the put again with a different name, or
        // use XmlModify to update the document.
    }

    return(0);
}
```

Setting Metadata

Every XML document stored in BDB XML actually consists of two kinds of information: the document itself, and metadata.

Metadata can contain an arbitrarily complex set of information. Typically it contains information about the document that you do not or can not include in the document itself. As an example, you could carry information about the date and time a document was

added to the container, last modified, or possibly an expiration time. Metadata might also be used to store information about the document that is external to BDB XML, such as the on-disk location where the document was originally stored, or possibly notes about the document that might be useful to the document's maintainer.

In other words, metadata can contain anything — BDB XML places no restrictions on what you can use it for. Further, you can both query and index metadata (see Using BDB XML Indices (page 56) for more information). It is even possible to have a document in your container that contains only metadata.

In order to set metadata onto a document, you must:

1. Optionally (but recommended), create a URI for each piece of metadata (in the form of a string).
2. Create an attribute name to use for the metadata, again in the form of a string.
3. Create the attribute value — the actual metadata information that you want to carry on the document — either as an `XmlValue` or as an `XmlData` class object.
4. Set this information on a `XmlDocument` object.
5. Optionally (but commonly) set the actual XML document to the same `XmlDocument` object.
6. Add the `XmlDocument` to the container.

For example:

```
#include "DbXml.hpp"
...

using namespace DbXml;
int main(void)
{
    // The document
    std::string fileName = "/export/testdoc1.xml";

    // The document's name.
    std::string docName = "testDoc1";

    // URI, attribute name, and attribute value used for
    // the metadata. We will carry a timestamp here
    // (hard coded for clarity purposes).
    std::string URI = "http://dbxmlExamples/metadata";
    std::string attrName = "createdOn";
    XmlValue attrValue(XmlValue::DATE_TIME, "2005-10-5T04:18:36");

    // Get a manager object.
    XmlManager myManager;   // The manager is closed when
                            // it goes out of scope.
```

```
    // Load the document in its entirety. The document's formatting is
    // preserved.
    myManager.setDefaultContainerType(XmlContainer::WholedocContainer);

    // Open the container. The container is closed when it goes
    // out of scope.
    XmlContainer myContainer =
        myManager.openContainer("container.bdbxml");

    // Need an update context for the put.
    XmlUpdateContext theContext = myManager.createUpdateContext();

    try {
        // Get the input stream.
        XmlInputStream *theStream =
            myManager.createLocalFileInputStream(fileName);

        // Get an XmlDocument
        XmlDocument myDoc = myManager.createDocument();

        // Set the document's name
        myDoc.setName(docName);
        // Set the content
        myDoc.setContentAsXmlInputStream(theStream);
        // Set the metadata
        myDoc.setMetaData(URI, attrName, attrValue);

        // Put the document into the container
        myContainer.putDocument(myDoc,        // The actual document.
                                theContext,  // The update context
                                             // (required).
                                0);          // Put flags.
    } catch (XmlException &e) {
        // Error handling goes here. You may want to check
        // for XmlException::UNIQUE_ERROR, which is raised
        // if a document with that name already exists in
        // the container. If this exception is thrown,
        // try the put again with a different name, or
        // use XmlModify to update the document.
    }

    return(0);
}
```

Chapter 5. Using XQuery with BDB XML

Documents are retrieved from BDB XML containers using XQuery expressions. XQuery is a language designed to query XML documents. Using XQuery, you can retrieve entire documents, subsections of documents, or values from one or more individual document nodes. You can also use XQuery to manipulate or transform values returned by document queries.

Note that XQuery represents a superset of XPath 2.0, which in turn is based on XPath 1.0. If you have prior experience with BDB XML 1.x, then you should be familiar with XPath as that was the query language offered by that library.

BDB XML supports the entire W3 XQuery specification. As of this printing, the specification is dated July 2004. However, BDB XML will be updated to track any changes in the working specification that may occur. You can find the XQuery specification at http://www.w3.org/XML/Query.

Beyond the W3C specifications, there are several good books on the market today that fully describe XQuery. In addition, there are many freely available resources on the web that provide a good introduction to the language. Searching for 'XQuery' in the Web search engine of your choice ought to return a wealth of information and pointers on the language.

That said, this chapter begins with a very thin introduction to XQuery that should be enough for you to understand any BDB XML concepts required to proceed with usage of the library. In particular, the next section of this manual highlights those aspects of XQuery that have unique meanings relative to BDB XML usage. Be aware, however, that the following introduction is not meant to be complete — a full treatment of XQuery is beyond the scope of an introductory manual such as this.

We follow this brief introduction to XQuery with a general description of querying documents stored in BDB XML containers, and examining the results of those queries. See Retrieving BDB XML Documents using XQuery (page 31) for that information.

XQuery: A Brief Introduction

XQuery can be used to:

1. Query for a document. Note that queries can be formed against an individual document, or against multiple documents.

2. Query for document subsections, including values found on individual document nodes.

3. Manipulate and transform the results of a query.

To do this, XQuery views an XML document as a collection of element, text, and attribute nodes. For example, consider the following XML document:

Example 5.1. A Simple XML Document

```
<?xml version="1.0"?>
<Node0>
<Node1 class="myValue1">Node1 text </Node1>
<Node2>
    <Node3>Node3 text</Node3>
    <Node3>Node3 text 2</Node3>
    <Node3>Node3 text 3</Node3>
    <Node4>300</Node4>
    </Node2>
</Node0>
```

In the above document, `<Node0>` is the *document's root node*, and `<Node1>` is an element node. Further, the element node, `<Node1>`, contains a single attribute node whose name is `class` and whose value is `myValue1`. Finally, `<Node1>` contains a text node whose value is `Node1 text`.

Referencing Portions of Documents using XQuery

A document's root can always be referenced using a single forward slash:

```
/
```

Subsequent element nodes in the document can be referenced using Unix-style path notation:

```
/Node1
```

To reference an attribute node, prefix the attribute node's name with '@':

```
/Node1/@class
```

To return the value contained in a node's text node (remember that not all element nodes contain a text node), use `distinct-values()` function:

```
distinct-values(/Node1)
```

To return the value assigned to an attribute node, you also use the `distinct-values()` function:

```
distinct-values(/Node1/@class)
```

Predicates

When you provide an XQuery path, what you receive back is a result set. You can further filter this result set by using *predicates*. Predicates are always contained in brackets ([]) and there are two types of predicates that you can use: numeric and boolean.

Numeric Predicates

Numeric predicates allow you to select a node based on its position relative to another node in the document (that is, based on its *context*).

For example, consider the document presented in A Simple XML Document (page 25). This document contains three <Node3> elements. If you simply enter the XQuery expression:

```
/Node1/Node2/Node3
```

all <Node3> elements in the document are returned. To return, say, the second <Node3> element, use a predicate:

```
/Node1/Node2/Node3[2]
```

Boolean Predicates

Boolean predicates filter a query result so that only those elements of the result are kept if the expression evaluates to true. For example, suppose you want to select a node only if its text node is equal to some value. Then:

```
/Node1/Node2[Node3="Node3 text 3"]
```

Context

The meaning of an XQuery expression can change depending on the current context. Within XQuery expressions, context is usually only important if you want to use relative paths or if your documents use namespaces. However, BDB XML only supports relative paths from within a predicate (see below). Also, do not confuse XQuery contexts with BDB XML contexts. While BDB XML contexts are related to XQuery contexts, they differ in that BDB XML contexts are a data structure that allows you to define namespaces, define variables, and to identify the type of information that is returned as the result of a query (all of these topics are discussed later in this chapter).

Relative Paths

Just like Unix filesystem paths, any path that does not begin with a slash (/) is relative to your current location in a document. Your current location in a document is determined by your context. Thus, if in A Simple XML Document (page 25) your context is set to `Node2`, you can refer to `Node3` with the simple notation:

```
Node3
```

Further, you can refer to a parent node using the following familiar notation:

```
..
```

and to the current node using:

```
.
```


Remember that BDB XML supports relative paths only from within predicates.

Namespaces

Natural language and, therefore, tag names can be imprecise. Two different tags can have identical names and yet hold entirely different sorts of information. Namespaces are intended to resolve any such sources of confusion.

Consider the following document:

Example 5.2. XML Documents and Namespaces

```
<?xml version="1.0"?>
<definition>
    <ring>
        Jewelry that you wear.
    </ring>
    <ring>
        A sound that a telephone makes.
    </ring>
    <ring>
        A circular space for exhibitions.
    </ring>
</definition>
```

As constructed, this document makes it difficult (though not impossible) to select the node for, say, a ringing telephone.

To resolve any potential confusion in your schema or supporting code, you can introduce namespaces to your documents. For example:

Example 5.3. Namespace Declaration

```
<?xml version="1.0"?>
<definition>
    <jewelry:ring xmlns:jewelry="http://myDefinition.dbxml/jewlery">
        Jewelry that you wear.
    </jewlery:ring>
    <sounds:ring xmlns:sounds="http://myDefinition.dbxml/sounds">
        A sound a telephone makes.
    </sounds:ring>
    <showplaces:ring
        xmlns:showplaces="http://myDefinition.dbxml/showplaces">
        A circular space for exhibitions.
    </showplaces:ring>
</definition>
```

Now that the document has defined namespaces, you can precisely query any given node:

```
/definition/sounds:ring
```

In order to perform queries against a document stored in BDB XML that makes use of namespaces, you must declare the namespace to your query. You do this using `XmlQueryContext::setNamespace()`. See Defining Namespaces (page 32) for more information.

By identifying the namespace to which the node belongs, you are declaring a context for the query.

The URI used in the namespace definition is not required to actually resolve to anything. The only criteria is that it be unique within the scope of any document set(s) in which it might be used.

Also, the namespace is only required to be declared once in the document. All subsequent usages need only use the relevant prefix. For example, we could have added the following to our previous document:

Example 5.4. Namespace Prefixes

```
<jewelry:diamond>
    The centerpiece of many rings.
</jewlery:diamond>
<showplaces:diamond>
    A place where baseball is played.
</showplaces:diamond>
```

Finally, namespaces can be used with attributes too. For an example:

Example 5.5. Namespaces with Attributes

```
<clubMembers>
    <surveyResults school:class="English"
        xmlns:school="http://myExampleDefinitions.dbxml/school"
        number="200"/>
    <surveyResults school:class="Mathematics"
        number="165"/>
    <surveyResults social:class="Middle"
        xmlns:social="http://myExampleDefinitions.dbxml/social"
        number="543"/>
</clubMembers>
```

Once you have declared a namespace for an attribute, you can query the attribute in the following way:

```
/clubMembers/surveyResults/@school:class
```

And to retrieve the value set for the attribute:

```
distinct-values(/clubMembers/surveyResults/@school:class)
```

Wildcards

XQuery allows you to use wildcards when document elements are unknown. For example:

```
/Node0/*/Node6
```

selects all the Node6 nodes that are 3 nodes deep in the document and whose path starts with `Node0`. Other wildcard matches are:

- Selects all of the nodes in the document:

  ```
  //*
  ```

- Selects all of the Node6 nodes that have three ancestors:

  ```
  /*/*/*/Node6
  ```

- Selects all the nodes immediately beneath Node5:

  ```
  /Node0/Node5/*
  ```

- Selects all of Node5's attributes:

  ```
  /Node0/Node5/@*
  ```

Navigation Functions

XQuery provides several functions that can be used for global navigation to a specific document or collection of documents. From the perspective of this manual, two of these are interesting because they have specific meaning from within the context of BDB XML

collection()

Within XQuery, `collection()` is a function that allows you to create a named sequence. From within BDB XML, however, it is also used to navigate to a specific container. In this case, you must identify to `collection()` the literal name of the container. You do this either by passing the container name directly to the function, or by declaring a default container name using the `XmlQueryContext::setDefaultCollection()` method.

Note that the container must have already been opened by the `XmlManager` in order for collection to reference that container. The exception to this is if `XmlManager` was opened using the `DBXML_ALLOW_AUTO_OPEN flag`.

For example, suppose you want to perform a query against a container named `container1.dbxml`. In this case, first open the container using `XmlManager::openContainer()` and then specify the collection() function on the query. For example:

```
collection("container1.dbxml")/Node0
```

Note that this is actually short-hand for:

```
collection("dbxml:/container1.dbxml")/Node0
```

`dbxml:/` is the default base URI for BDB XML. You an change the base URI using `XmlQueryContext::setBaseURI()`.

If you want to perform a query against multiple containers, use the union ("|") operator. For example, to query against containers `c1.dbxml` and `c2.dbxml`, you would use the following expression:

```
(collection("c1.dbxml") | collection("c2.dbxml"))/Node0
```

See Retrieving BDB XML Documents using XQuery (page 31) for more information on how to prepare and perform queries.

doc()

XQuery provides the `doc()` function so that you can trivially navigate to the root of a named document. `doc()` is required to take a URI.

To use `doc()` to navigate to a specific document stored in BDB XML, provide an XQuery path that uses the `dbxml:` base URI, and that identifies the container in which the document can be found. The actual document name that you provide is the same name that was set for the document when it was added to the container (see Adding Documents (page 18) for more information).

For example, suppose you have a document named "mydoc1.xml" in container "container1.dbxml". Then to perform a query against that specific document, first open `container1.dbxml` and then provide a query something like this:

```
doc("dbxml:/container1.dbxml/mydoc1.xml")/Node0
```

See Retrieving BDB XML Documents using XQuery (page 31) for more information on how to prepare and perform queries.

Using FLWOR with BDB XML

XQuery offers iterative and transformative capabilities through FLWOR (pronounced "flower") expressions. *FLWOR* is an acronym that stands for the five major clauses in a FLWOR expression: `for, let, where, order, by` and `return`. Using FLWOR expressions, you can iterate over sequences (frequently result sets in BDB XML), use variables, and filter, group, and sort sequences. You can even use FLWOR to perform joins of different data sources.

For example, suppose you had documents in your container that looked like this:

```
<product>
    <name>Widget A</name>
    <price>0.83</price>
</product>
```

In this case, queries against the container for these documents return the documents in order by their document name. But suppose you wanted to see all such documents in your container, ordered by price. You can do this with a FLWOR expression:

```
for $i in collection("myContainer.dbxml")/product
order by $i/price descending
return $i
```

Note that from within BDB XML, you must provide FLWOR expressions in a single string. Lines can be separated either by a carriage return ("\n") or by a space. Thus, the above expression would become:

```
std::string flwor="for $i in collection('myContainer.dbxml')/product\n";
flwor += "order by $i/price descending\n";
flwor += "return $i"
```

Retrieving BDB XML Documents using XQuery

Documents are retrieved from BDB XML when they match an XQuery path expression. Queries are either performed or prepared using an `XmlManager` object, but the query itself usually restricts its scope to a single container or document using one of the XQuery Navigation Functions (page 29).

When you perform a query, you must provide:

1. The XQuery expression to be used for the query contained in a single string object.
2. An `XmlQueryContext` object that identifies contextual information about the query, such as the namespaces in use and what you want for results (entire documents, or document values).

What you then receive back is a result set that is returned in the form of an `XmlResults` object. You iterate over this result sets in order to obtain the individual documents or values returned as a result of the query.

The Query Context

Context is a term that is heavily used in both BDB XML and XQuery. While overlap exists in how the term is used between the two, it is important to understand that differences exist between what BDB XML means by context and what the XQuery language means by it.

In XQuery, the context defines aspects of the query that aid in query navigation. For example, the XQuery context defines things like the namespace(s) and variables used by the query, the query's focus (which changes over the course of executing the query), and the functions and collations used by the query. Most thorough descriptions of XQuery will describe these things in detail.

In BDB XML, however, the context is a physical object (`XmlQueryContext`) that is used for very limited things (compared to what is meant by the XQuery context). You can use `XmlQueryContext` to control only part of the XQuery context. You also use `XmlQueryContext`

to control BDB XML's behavior toward the query in ways that have no corresponding concept for XQuery contexts.

Specifically, you use `XmlQueryContext` to:

- Define the namespaces to be used by the query.
- Define any variables that might be needed for the query, although, these are not the same as the variables used by XQuery FLWOR expressions (see Defining Variables (page 33)).
- Define the return type. That is, the type of information returned as a result of the query (see Defining Return Types (page 33)).
- Defining whether the query is processed "eagerly" or "lazily" (see Defining the Evaluation Type (page 34)).

Note that BDB XML also uses the `XmlQueryContext` to identify the query's focus as you iterate over a result set. See Examining Document Values (page 39) for more information.

Defining Namespaces

In order for you to use a namespace prefix in your query, you must first declare that namespace to BDB XML. When you do this, you must identify the URI that corresponds to the prefix, and this URI must match the URI in use on your documents.

You can declare as many namespaces as are needed for your query.

To declare a namespace, use `XmlQueryContext::setNamespace()`. For example:

```
#include "DbXml.hpp"
...

using namespace DbXml;

...

// Get a manager object.
XmlManager myManager;

// Open a container
XmlContainer myContainer =
    myManager.openContainer("exampleData.dbxml");

// Get a query context
XmlQueryContext context = myManager.createQueryContext();

// Declare a namespace
context.setNamespace("fruits", "http://groceryItem.bdbxml/fruits");
context.setNamespace("vegetables", "http://groceryItem.bdbxml/vegetables");
```

Defining Variables

In XQuery FLWOR expressions, you can set variables using the `let` clause. In addition to this, you can use variables that are defined by BDB XML You define these variables using `XmlQueryContext::setVariableValue()`.

You can declare as many variables using `XmlQueryContext::setVariableValue()` as you need. Note that the variables that you declare this way can only be used from within a predicate. For example:

```
#include "DbXml.hpp"
...

using namespace DbXml;

...

// Get a manager object.
XmlManager myManager;

// Open a container
XmlContainer myContainer =
    myManager.openContainer("exampleData.dbxml");

// Get a query context
XmlQueryContext context = myManager.createQueryContext();

// Declare a variable. Note that this method really wants an XmlValue
// object as the variable's argument. However, we just give it a
// string here and allow XmlValue's string constructor to create
// the XmlValue object for us.
context.setVariableValue("myVar", "Tarragon");

// Declare the query string
std::string myQuery =
    "collection('exampleData.dbxml')/product[item=$myVar]";
```

Defining Return Types

When you perform a BDB XML query, you can define the type of data that you want returned. The return types that you are most likely to use are:

Return Type	Description
`DeadValues`	The value returned is a copy of the value found in the actual document.
`LiveValues`	The value returned is a reference to the actual document stored in BDB XML. This is the default return type.

You use `XmlQueryContext::setReturnType()` to set a query's return type. For example:

```
#include "DbXml.hpp"
...

using namespace DbXml;

...

// Get a manager object.
XmlManager myManager;

// Open a container
XmlContainer myContainer =
    myManager.openContainer("exampleData.dbxml");

// Get a query context
XmlQueryContext context = myManager.createQueryContext();

// Set the return type to DeadValues
context.setReturnType(XmlQueryContext::DeadValues);
```

Defining the Evaluation Type

The evaluation type defines how much work BDB XML performs as a part of the query, and how much it defers until the results are evaluated. There are two evaluation types:

Evaluation Type	Description
`Eager`	The query is executed and its resultant values are derived and stored in-memory before the query returns. This is the default.
`Lazy`	Minimal processing is performed before the query returns, and the remaining processing is deferred until you enumerate over the result set.

You use `XmlQueryContext::setEvaluationType()` to set a query's return type. For example:

```
#include "DbXml.hpp"
...

using namespace DbXml;

...

// Get a manager object.
XmlManager myManager;

// Open a container
XmlContainer myContainer =
    myManager.openContainer("exampleData.dbxml");

// Get a query context
```

```
XmlQueryContext context = myManager.createQueryContext();

// Set the evaluation type to Lazy.
context.setEvaluationType(XmlQueryContext::Lazy);
```

Performing Queries

You perform queries using an `XmlManager` object. When you perform a query, you can either:

1. Perform a one-off query using `XmlManager::query()`. This is useful if you are performing queries that you know you will never repeat within the process scope. For example, if you are writing a command line utility to perform a query, display the results, then shut down, you may want to use this method.

2. Perform the same query repeatedly by using `XmlManager::prepare()` to obtain an `XmlQueryExpression` object. You can then run the query repeatedly by calling `XmlQueryExpression::execute()`.

   Creation of a query expression is fairly expensive, so any time you believe you will perform a given query more than one time, you should use this approach over the `query()` method.

Regardless of how you want to run your query, you must restrict the scope of your query to a container, document, or node. Usually you use one of the XQuery navigation functions to do this. See Navigation Functions (page 29) for more information.

Note that you can indicate that the query is to be performed lazily. If it is performed lazily, then only those portions of the document that are actually required to satisfy the query are returned in the results set immediately. All other portions of the document may then be retrieved by BDB XML as you iterate over and use the items in the result set.

If you are using node-level storage, then a lazy query may result in only the document being returned, but not its metadata, or the metadata but not the document itself. In this case, `XmlDocument::fetchAllData()` to ensure that you have both the document and its metadata.

To specify laziness for the query, use DBXML_LAZY_DOCS as a flag value to either `XmlManager::query()` or `XmlQueryExpression::execute()`.

Be aware that lazy docs is different from lazy evaluation. Lazy docs determines whether all document data and document metadata is returned as a result of the query. Lazy evaluation determines how much query processing is deferred until the results set is actually examined.

For example, the following executes a query against an `XmlContainer` using `XmlManager::query()`.

```
#include "DbXml.hpp"
...

using namespace DbXml;

...
```

```
// Get a manager object.
XmlManager myManager;

// Open a container
XmlContainer myContainer =
    myManager.openContainer("exampleData.dbxml");

// Get a query context
XmlQueryContext context = myManager.createQueryContext();

// Declare a namespace
context.setNamespace("fruits", "http://groceryItem.dbxml/fruits");

// Declare the query string
std::string myQuery =
    "collection('exampleData.dbxml')/fruits:product[item=$myVar]";

// Prepare (compile) the query
XmlQueryExpression qe = myManager.prepare(myQuery, context);

// Run the query. Note that you can perform this query many times
// without suffering the overhead of re-creating the query expression.
// Notice that the only thing we are changing is the variable value,
// which allows us to control exactly what gets returned for the query.
XmlResults results = qe.execute(context, 0);

context.setVariableValue(myVar, "Tarragon");
XmlResults results = qe.execute(context);

// Do something with the results

context.setVariableValue(myVar, "Oranges");
results = qe.execute(context);

// Do something with the results

context.setVariableValue(myVar, "Kiwi");
results = qe.execute(context);
```

Metadata Based Queries

You can query for documents based on the metadata that you set for them. To do so, do the following:

- Define a namespace for the query that uses the URI that you set for the metadata against which you will perform the query. If you did not specify a namespace for your metadata when you added it to the document, then use an empty string.

- Perform the query using the special `dbxml:metadata()` from within a predicate.

For example, suppose you placed a timestamp in your metadata using the URI '`http://dbxmlExamples/timestamp`' and the attribute name '`timeStamp`'. Then you can query for documents that use a specific timestamp as follows:

```
#include "DbXml.hpp"
...

using namespace DbXml;

...

// Get a manager object.
XmlManager myManager;

// Open a container
XmlContainer myContainer =
    myManager.openContainer("exampleData.dbxml");
std::string col = "collection('exampleData.dbxml')";

// Get a query context
XmlQueryContext context = myManager.createQueryContext();

// Declare a namespace. The first argument, 'ts', is the
// namespace prefix and in this case it can be anything so
// long as it is not reused with another URI within the same
// query.
context.setNamespace("ts", "http://dbxmlExamples/timestamp");

// Declare the query string
std::string myQuery = col;
myQuery += "/*[dbxml:metadata('ts:timeStamp')=00:28:38]";

// Prepare (compile) the query
XmlQueryExpression qe = myManager.prepare(myQuery, context);

// Run the query.
XmlResults results = qe.execute(context, 0);
```

Examining Query Results

When you perform a query against BDB XML, you receive a results set in the form of an `XmlResults` object. To examine the results, you iterate over this result set, retrieving each element of the set as an `XmlValue` object.

Once you have an individual result element, you can obtain the data encapsulated in the `XmlValue` object in a number of ways. For example, you can obtain the information as a

string object using `XmlValue::asString()`. Alternatively, you could obtain the data as an `XmlDocument` object using `XmlValue::asDocument()`.

It is also possible to use DOM-like navigation on the `XmlValue` object since that class offers navigational methods such as `XmlValue::getFirstChild()`, `XmlValue::getNextSibling()`, `XmlValue::getAttributes()`, and so forth. For details on these and other `XmlValue` attributes, see the BDB XML C++ API Reference documentation.

For example, the following code fragment performs a query and then loops over the result set, obtaining and displaying the document's name from an `XmlDocument` object before displaying the document itself.

```
#include "DbXml.hpp"
...

using namespace DbXml;

...

// Get a manager object.
XmlManager myManager;

// Open a container
XmlContainer myContainer =
    myManager.openContainer("exampleData.dbxml");

// Get a query context
XmlQueryContext context = myManager.createQueryContext();

// Declare a namespace
context.setNamespace("fruits", "http://groceryItem.dbxml/fruits");

// Declare the query string. Find all the product documents
// in the fruits namespace.
std::string myQuery = "collection('exampleData.dbxml')/fruits:product";

// Perform the query.
XmlResults results = myManager.query(myQuery, context);

// Show the size of the result set
std::cout << "Found " << results.size() << " documents for query: '"
          << myQuery << "'" << std::endl;

// Display the result set
XmlValue value;
while (results.next(value)) {
    XmlDocument theDoc = value.asDocument();
    std::string docName = theDoc.getName();
    std::string docString = value.asString();
```

```
    std::cout << "Document " << docName << ":" << std::endl;
    std::cout << docString << std::endl;
    std::cout << "============================\n" << std::endl;
}
```

Examining Document Values

It is frequently useful to retrieve a document from BDB XML and then perform follow-on queries to retrieve individual values from the document itself. You do this by creating and executing a query, except that you pass the specific `XmlValue` object that you want to query to the `XmlQueryExpression::execute()` method. You must then iterate over a result set exactly as you would when retrieving information from a container.

For example, suppose you have an address book product that manages individual contacts using XML documents such as:

```
<contact>
    <familiarName>John</familiarName>
    <surname>Doe</surname>
    <phone work="555 555 5555" home="555 666 777" />
    <address>
        <street>1122 Somewhere Lane</street>
        <city>Nowhere</city>
        <state>Minnesota</state>
        <zipcode>11111</zipcode>
    </address>
</contact>
```

Then you could retrieve individual documents and pull data off of them like this:

```
#include "DbXml.hpp"
...

using namespace DbXml;

...

// Get a manager object.
XmlManager myManager;

// Open a container
XmlContainer myContainer =
    myManager.openContainer("exampleData.dbxml");

// Declare the query string. Retrieves all the documents
// for people with the last name 'Doe'.
std::string myQuery = "collection('exampleData.dbxml')/contact";

// Query to get the familiar name from the
```

```
// document.
std::string fn = "distinct-values(/contact/familiarName)";

// Query to get the surname from the
// document.
std::string sn = "distinct-values(/contact/surname)";

// Work phone number
std::string wrkPhone = "distinct-values(/contact/phone/@work)";

// Get the context for the XmlManager query
XmlQueryContext managerContext = myManager.createQueryContext();

// Get a context for the document queries
XmlQueryContext documentContext = myManager.createQueryContext();

// Prepare the XmlManager query
XmlQueryExpression managerQuery =
   myManager.prepare(myQuery, managerContext);

// Prepare the individual document queries
XmlQueryExpression fnExpr = myManager.prepare(fn, documentContext);
XmlQueryExpression snExpr = myManager.prepare(sn, documentContext);
XmlQueryExpression wrkPhoneExpr =
   myManager.prepare(wrkPhone, documentContext);

// Perform the query.
XmlResults results = managerQuery.execute(managerContext, 0);

// Display the result set
XmlValue value;
while (results.next(value)) {
    // Get the individual values
    XmlResults fnResults = fnExpr.execute(value, documentContext);
    XmlResults snResults = snExpr.execute(value, documentContext);
    XmlResults phoneResults =
        wrkPhoneExpr.execute(value, documentContext);

    std::string fnString;
    XmlValue fnValue;
    if (fnResults.size() > 0) {
        fnResults.next(fnValue);
        fnString = fnValue.asString();
    } else {
        continue;
    }

    std::string snString;
    XmlValue snValue;
```

```
    if (snResults.size() > 0) {
        snResults.next(snValue);
        snString = snValue.asString();
    } else {
        continue;
    }

    std::string phoneString;
    XmlValue phoneValue;
    if (phoneResults.size() > 0) {
        phoneResults.next(phoneValue);
        phoneString = phoneValue.asString();
    } else {
        continue;
    }

    std::cout << fnString << " " << snString << ": "
              << phoneString << std::endl;

}
```

Note that you can use the same basic mechanism to pull information out of very long documents, except that in this case you need to maintain the query's focus; that is, the location in the document that the result set item is referencing. For example suppose you have a document with 2,000 `contact` nodes and you want to get the `name` attribute from some particular `contact` in the document.

There are several ways to perform this query. You could, for example, ask for the node based on the value of some other attribute or element in the node:

```
/document/contact[category='personal']
```

Or you could create a result set that holds all of the document's `contact` nodes:

```
/document/contact
```

Regardless of how you get your result set, you can then go ahead and query each value in the result set for information contained in the value. To do this:

1. Make sure you use the same `XmlQueryContext` object as you used to generate the result set in the first place. This object will track the result item's focus (that is, the node's location in the larger document — the self axis.) for you.

2. Iterate over the result set as normal.

3. Query for document information as described above. However, in this case change the query so that you reference the self access. That is, for the surname query described above, you would use the following query instead so as to reference nodes relative to the current node (notice the self-access (.) in use in the following query):

   ```
   distinct-values(./surname)
   ```

Examining Metadata

When you retrieve a document from BDB XML, there are two ways to examine the metadata associated with that document. The first is to use `XmlDocument::getMetaData()`. Use this form if you want to examine the value for a specific metadata value.

The second way to examine metadata is to obtain an `XmlMetaDataIterator` object using `XmlDocument::getMetaDataIterator()`. You can use this mechanism to loop over and display every piece of metadata associated with the document.

For example:

```
#include "DbXml.hpp"
...

using namespace DbXml;

...

// Get a manager object.
XmlManager myManager;

// Open a container
XmlContainer myContainer =
    myManager.openContainer("exampleData.dbxml");

// Get a query context
XmlQueryContext context = myManager.createQueryContext();

// Declare a namespace
context.setNamespace("fruits", "http://groceryItem.dbxml/fruits");

// Declare the query string. Find all the product documents
// in the fruits namespace.
std::string myQuery = "collection('exampleData.dbxml')/fruits:product";

// Perform the query.
XmlResults results = myManager.query(myQuery, context);

// Display the result set
XmlValue value;
while (results.next(value)) {
    XmlDocument theDoc = value.asDocument();

    // Display all of the metadata set for this document
    XmlMetaDataIterator mdi = theDoc.getMetaDataIterator();
    std::string returnedURI;
    std::string returnedName;
    XmlValue returnedValue;
```

```
    std::cout << "For document '" << theDoc.getName()
              << "' found metadata:" << std::endl;

    while (mdi.next(returnedURI, returnedName, returnedValue)) {
        std::cout << "\tURI: " << returnedURI
                  << ", attribute name: " << returnedName
                  << ", value: " << returnedValue
                  << std::endl;
    }

    // Display a single metadata value:
    std::string URI = "http://dbxmlExamples/timestamp";
    std::string attrName = "timeStamp";
    XmlValue newRetValue;

    bool gotResult = theDoc.getMetaData(URI, attrName, newRetValue);
    if (gotResult) {
        std::cout << "For URI: " << URI << ", and attribute " << attrName
                  << ", found: " << newRetValue << std::endl;
    }

    std::cout << "==============================\n" << std::endl;
}
```

Chapter 6. Managing Documents in Containers

BDB XML provides APIs for deleting, replacing, and modifying documents that are stored in containers. This chapter discusses these activities.

Deleting Documents

You can delete a document by calling `XmlContainer::deleteDocument()`. This method can operate either on a document's name or on an `XmlDocument` object. You might want to use an `XmlDocument` object to delete a document if you have queried your container for some documents and you want to delete every document in the results set.

For example:

```
#include "DbXml.hpp"
...

using namespace DbXml;

...

// Get a manager object.
XmlManager myManager;

// Open a container
XmlContainer myContainer =
    myManager.openContainer("exampleData.dbxml");

// Get a query context
XmlQueryContext context = myManager.createQueryContext();
// Declare a namespace
context.setNamespace("fruits", "http://groceryItem.dbxml/fruits");

// Declare the query string. Find all the product documents
// in the fruits namespace.
std::string myQuery = "collection('exampleData.dbxml')/fruits:product";

// Perform the query.
XmlResults results = myManager.query(myQuery, context);

// Delete everything in the results set
XmlUpdateContext uc = myManager.createUpdateContext();
XmlDocument theDoc = myManager.createDocument();
while (results.next(theDoc)) {
    myContainer.deleteDocument(theDoc, uc);
}
```

Replacing Documents

You can either replace a document in its entirety as described here, or you can modify just portions of the document as described in Modifying XML Documents (page 46).

If you already have code in place to perform document modifications, then replacement is the easiest mechanism to implement. However, replacement requires that at least the entire replacement document be held in memory. Modification, on the other hand, only requires that the portion of the document to be modified be held in memory. Depending on the size of your documents, modification may prove to be significantly faster and less costly to operate.

You can directly replace a document that exists in a container. To do this:

1. Retrieve the document from the container. Either do this using an XQuery query and iterating through the results set looking for the document that you want to replace, or use `XmlContainer::getDocument()` to retrieve the document by its name. Either way, make sure you have the document as an `XmlDocument` object.

2. Use one of `XmlDocument::setContent()`, `XmlDocument::setContentAsDOM()`, or `XmlDocument::setContentAsXmlInputStream()` to set the object's content to the desired value.

3. Use `XmlContainer::updateDocument()` to save the modified document back to the container.

Alternatively, you can create a new blank document using `XmlManager::createDocument()`, set the document's name to be identical to a document already existing in the container, set the document's content to the desired content, then call `XmlContainer::updateDocument()`.

For example:

```
#include "DbXml.hpp"
...

using namespace DbXml;

...

// Get a manager object.
XmlManager myManager;

// Open a container
XmlContainer myContainer =
    myManager.openContainer("exampleData.dbxml");

// Document to modify
std::string docName = "doc1.xml";
XmlDocument theDoc = myContainer.getDocument(docName);
```

```
// Modify it
theDoc.setContent("<a><b>random content</a></b>");

// Put it back into the container
XmlUpdateContext uc = myManager.createUpdateContext();
myContainer.updateDocument(theDoc, uc);
```

Modifying XML Documents

BDB XML provides a mechanism with which you can modify documents stored in your containers, without using the document update procedure described in the previous section. You do this using `XmlModify` objects.

By using `XmlModify`, you can avoid the need to hold the entire replacement document in memory.

Essentially, you use `XmlModify` methods to identify a series of modification steps to be taken against a document. These steps allow you to add, delete, rename, and replace document nodes. You can also manipulate comments and processing instructions.

Once you have finished identifying the modification steps that you want to perform, you use `XmlModify::execute()` to apply the modifications to either a single document (by passing it an `XmlValue` object), or a set of documents (by passing it an `XmlResults` object).

Modification Parameters

There are a common set of parameters to the `XmlModify` modification methods that are worth examining before proceeding. These arguments have roughly the same meaning, regardless of the modification action being requested. They appear on the modification methods in the following order:

- `XmlQueryExpression selectionExpr`

  This parameter contains an XQuery expression that selects the portion of the document to be modified. For example, if you want to rename a node, then this expression would select the node that you want to rename.

- `XmlObject type`

  This parameter identifies the type of information you are inserting into the document. That is, you use this parameter to indicate whether you are inserting an element node, an attribute node, text node, a processing instruction, or a comment. See the API Reference documentation for information on how to specify these types.

- `std::string name`

  Identifies the name of the information you are inserting. For example, if you are inserting an element or attribute node, then this provides the name of that node. The value of this parameter is ignored if you are inserting a text or comment node.

- `std::string content`

  Identifies the content that you are inserting. If you are inserting an element node, then this must contain either a text node, or a valid child content for the node. For attribute nodes, this contains the value to which the parameter is equal. For processing instructions, this contains all of the information that appears in the processing instruction other than the processing instruction's name.

Modification Methods

`XmlModify` provides a series of methods that you use to identify how a document is to be modified. To define your document modification, you call these methods as many times as is required. When `XmlModify::execute()` is called, the documents are modified according to the instructions provided in the order that they were provided.

The `XmlModify` modification methods are:

- XmlModify::addAppendStep() (page 47)
- XmlModify::addInsertAfterStep() (page 49)
- XmlModify::addInsertBeforeStep() (page 50)
- XmlModify::addRemoveStep() (page 51)
- XmlModify::addRenameStep() (page 52)
- XmlModify::addUpdateStep() (page 52)

XmlModify::addAppendStep()

Appends the provided content to the targeted node's content.

If you are appending an element node, then the new node is by default appended immediately after the targeted node's last child node. Note, however, that this method provides a location parameter that identifies the index of the child node at which the append operation is to be performed. Note also that if the location parameter is specified, then the new node is inserted immediately prior to the identified child node.

For example, consider the following document:

```
<a>
   <b1>first child</b1>
   <b2>second child</b2>
   <b3>third child</b3>
</a>
```

For this document, if you:

- Provide an XQuery selection expression of:

  ```
  /a
  ```

- Indicate you are inserting an element node.
- Provide a name of "b4".
- Provide "my inserted child".
- Leave the location parameter blank.

Then when the modification is executed against the document, the resulting document is:

```
<a>
   <b1>first child</b1>
   <b2>second child</b2>
   <b3>third child</b3>
   <b4>my inserted child</b4>
</a>
```

However, if you give the location parameter a value of "0" (modify at the first child node), then the resulting document is:

```
<a>
   <b4>my inserted child</b4>
   <b1>first child</b1>
   <b2>second child</b2>
   <b3>third child</b3>
</a>
```

If you indicate that the type of information to be inserted is an attribute node, then the location parameter is always ignored and the new attribute is inserted at the node selected by the selection expression. So for a selection expression of

```
/a
```

The resulting document is:

```
<a  b4="my inserted child">
   <b1>first child</b1>
   <b2>second child</b2>
   <b3>third child</b3>
</a>
```

If you indicate that the type of information to be inserted is a comment node, and you leave the location parameter blank, then the resulting document is:

```
<a>
   <b1>first child</b1>
   <b2>second child</b2>
   <b3>third child</b3>
<!-- my inserted child -->
</a>
```

If you indicate a location of 0, then the resulting document is:

```
<a>
<!-- my inserted child -->
   <b1>first child</b1>
   <b2>second child</b2>
   <b3>third child</b3>
</a>
```

And finally, if you are inserting a text node with no location parameter, the resulting document is:

```
<a>
   <b1>first child</b1>
   <b2>second child</b2>
   <b3>third childmy inserted child</b3>
</a>
```

Note that the selection expression you provide here must not select an attribute node or an exception is thrown when the modification is executed.

XmlModify::addInsertAfterStep()

Inserts the identified content after the selected node. Note that the node that you target for this operation cannot select the document root node or an attribute node, or an exception is thrown.

If you are inserting an element node, then the new node is inserted after the closing tag of the targeted node.

For example, consider the following document:

```
<a>
   <b>
     text node
   </b>
</a>
```

For this document, if you:

- Provide an XQuery selection expression of:

  ```
  /a/b
  ```

- Indicate you are inserting an element node.
- Provide a name of "b2".
- Provide "my inserted node".

Then when the modification is executed against the document, the resulting document is:

```
<a>
   <b>
     text node
   </b>
<b2>my inserted node</b2>
</a>
```

If you are inserting an attribute, then the new attribute is placed on the selected node's *parent* node. So for this example, the resulting document would be:

```
<a b2="my inserted node">
   <b>
     text node
   </b>
</a>
```

XmlModify::addInsertBeforeStep()

Identical to XmlModify::addInsertAfterStep() (page 49), except that element nodes, text, comments, and processing instructions are inserted prior to the node selected by the selection expression.

Again, you cannot select the root node or an attribute node or an exception is thrown when this instruction is executed.

For example, consider the following document:

```
<a>
   <b>
     text node
   </b>
</a>
```

For this document, if you:

- Provide an XQuery selection expression of:

  ```
  /a/b
  ```

- Indicate you are inserting an element node.
- Provide a name of "b2".
- Provide "my inserted node".

Then when the modification is executed against the document, the resulting document is:

```
<a>
<b2>my inserted node</b2>
   <b>
     text node
```

```
   </b>
</a>
```

Attribute insertion is handled identically to `XmlModify::addInsertAfterStep()`. If you are inserting an attribute, then the new attribute is placed on the selected node's *parent* node. So for this example, the resulting document would be:

```
<a b2="my inserted node">
   <b>
     text node
   </b>
</a>
```

XmlModify::addRemoveStep()

Removes the node targeted by the selection expression. For example, if you have the following document:

```
<a>
   <b>
     <c>
        text node
     </c>
   </b>
</a>
```

and you provide a selection expression of:

```
/a/b/c
```

then the resulting document is:

```
<a>
   </b>
</a>
```

Similarly, if you have the following document:

```
<a>
   <b>
     <c attr1="foo">
        text node
     </c>
   </b>
</a>
```

and you provide a selection expression of:

```
/a/b/c/@attr1
```

then the resulting document is:

```
<a>
   <b>
```

```
      <c>
         text node
      </c>
   </b>
</a>
```

Again, it is an error to target the document's root node with this method.

XmlModify::addRenameStep()

This method renames the selected node. For example, if you have the following document:

```
<a>
   <b>
     <c attr1="foo">
        text node
     </c>
   </b>
</a>
```

and you provide a selection expression of:

```
/a
```

and you provide a new name of 'z', then the resulting document is:

```
<z>
   <b>
     <c attr1="foo">
        text node
     </c>
   </b>
</z>
```

Similarly, a selection expression of:

```
/a/b/c/@attr1
```

and a new name of 'z' leaves you with:

```
<a>
   <b>
     <c z="foo">
        text node
     </c>
   </b>
</a>
```

XmlModify::addUpdateStep()

This method updates (replaces) the contents of the targeted node with with new content. If an element node is targeted, the content here is expected to be a text node. For example, given the following document:

```
<a>
   <b>
     <c attr1="foo">
        text node
     </c>
   </b>
</a>
```

providing a selection expression of:

```
/a
```

and replacement content:

```
Update content
```

Then the resulting document is:

```
<a>
Update content
</a>
```

If, however, you provide replacement content of:

```
<z>Update content</z>
```

(which includes the reserved characters '<' and '>'), then the method translates this into content that is appropriate for a text node. In this case, the resulting document is:

```
<a>
&lt;z>Update content&lt;/z>
</a>
```

Similarly, providing a selection expression of:

```
/a/b/c/@attr1
```

and replacement content:

```
Update content
```

results in the following document:

```
<a>
   <b>
     <c attr1="Update content">
        text node
     </c>
   </b>
</a>
```

Modification Example

To illustrate document modification, we will:

1. Retrieve a document named "doc1.xml" from a container.
2. Rename an attribute node called 'attr1' to 'myAttribute'.
3. Add a child node called "newChild" to node "node2".
4. Remove a node called "removeNode".
5. Update the contents of attribute node 'myAttribute' with the string "replacement content".

The document that we will update is as follows:

```
<sampleDocument>
    <node1 attr1="an attribute node" />
    <removeNode>Some content to remove</removeNode>
    <node2 />
</sampleDocument>
```

Notice that in performing the modification, we are not required to explicitly save the modified document back into the container; that is done for us under the covers.

```
#include "DbXml.hpp"
...

using namespace DbXml;

...

// Get a manager object.
XmlManager myManager;

// Open a container
XmlContainer myContainer =
    myManager.openContainer("exampleData.dbxml");

XmlQueryContext qc = myManager.createQueryContext();
XmlUpdateContext uc = myManager.createUpdateContext();
XmlModify mod = myManager.createModify();

// Build the modification object.
// Rename the attribute node from 'attr1' to 'myAttribute'.
XmlQueryExpression select =
    myManager.prepare("/sampleDocument/node1/@attr1", qc);
mod.addRenameStep(select, "myAttribute");

// Add '<newChild>' node to '<node2>'
std::string newChildContent = "<c1>some content</c1>";
select = myManager.prepare("/sampleDocument/node2", qc);
mod.addAppendStep(select,
                  XmlModify::Element,
```

```
                    "newChild",
                    newChildContent);

// Remove <removeNode> from the document
select = myManager.prepare("/sampleDocument/removeNode", qc);
mod.addRemoveStep(select);

// Replace the contents of /sampleDocument/node1/@myAttribute. Notice
// the attribute was renamed from attr1 in the first step of this
// modification. Modifications are performed in the specified order.
std::string attrContent = "replacement content";
select = myManager.prepare("/sampleDocument/node1/@myAttribute", qc);
mod.addUpdateStep(select, attrContent);

// Now retrieve the document we want to modify from the container.
// Notice that we could have performed a query against the container,
// and then handed the entire result set to this method. In that case,
// every document contained in the result set is modified.
XmlDocument retDoc = myContainer.getDocument("doc1.xml");
XmlValue docValue(retDoc);
mod.execute(docValue, qc, uc);

// Show that the modification was performed
// and written to the container.
XmlDocument retDoc2 = myContainer.getDocument("doc1.xml");
std::string doc1String;
std::cout << retDoc2.getName() << ":\n"
          << retDoc2.getContent(doc1String)
          << "\n\n" << std::endl;
```

When we run this code, the program displays the modified document which is now:

```
doc1.xml:
<sampleDocument>
    <node1 myAttribute="replacement content" />

    <node2><newChild><c1>some content</c1></newChild></node2>
</sampleDocument>
```

Chapter 7. Using BDB XML Indices

BDB XML provides a robust and flexible indexing mechanism that can greatly improve the performance of your BDB XML queries. Designing your indexing strategy is one of the most important aspects of designing a BDB XML-based application.

To make the most effective usage of BDB XML indices, design your indices for your most frequently occurring XQuery queries. Be aware that BDB XML indices can be updated or deleted in-place so if you find that your application's queries have changed over time, then you can modify your indices to meet your application's shifting requirements.

The time it takes to re-index a container is proportional to the container's size. Re-indexing a container can be an extremely expensive and time-consuming operation. If you have large containers in use in a production setting, you should not expect container re-indexing to be a routine operation.

You can define indices for both document content and for metadata. You can also define default indices that are used for portions of your documents for which no other index is defined.

When you declare an index, you must identify its type and its syntax. There are two ways that you can provide this information. One way is to provide a string that identifies the type and syntax for the index. The other way is to use enumerated types to do that same thing.

Most of BDB XML's APIs that you use to manage indices allow you to use either form for declaring indices. A few methods, however, only support the string approach.

See Syntax Types (page 59) for information on specifying the index syntax.

Index Types

The index type is defined by the following four types of information:

Uniqueness

Uniqueness indicates whether the indexed value must be unique within the container. For example, you can index an attribute and declare that index to be unique. This means the value indexed for the attribute must be unique within the container.

By default, indexed values are not unique; you must explicitly declare uniqueness for your indexing strategy in order for it to be enforced.

Path Types

If you think of an XML document as a tree of nodes, then there are two types of path elements in the tree. One type is just a node, such as an element or attribute within the document. The other type is any location in a path where two nodes meet. The path type, then, identifies the path element type that you want indexed. Path type `node` indicates that you want to index a single node in the path. Path type `edge` indicates that you want to index the portion of the path where two nodes meet.

Of the two of these, the BDB XML query processor prefers `edge`-type indices because they are more specific than an `node`-type index. This means that the query processor will use a `edge`-type index over a `node`-type if both indices provide similar information.

Consider the following document:

```
<vendor type="wholesale">
    <name>TriCounty Produce</name>
    <address>309 S. Main Street</address>
    <city>Middle Town</city>
    <state>MN</state>
    <zipcode>55432</zipcode>
    <phonenumber>763 555 5761</phonenumber>
    <salesrep>
        <name>Mort Dufresne</name>
        <phonenumber>763 555 5765</phonenumber>
    </salesrep>
</vendor>
```

Suppose you want to declare an index for the `name` node in the preceding document. In that case:

Path Type	Description
`node`	There are two locations in the document where the `name` node appears. The first of these has a value of "TriCounty Produce," while the second has a value of "Mort Dufresne." The result is that the `name` node will require two index entries, each with a different value. Queries based on a `name` node may have to examine both index entries in order to satisfy the query.

`edge`	There are two edge nodes in the document that involve the `name` node:
	
	`/vendor/name`
	
	and
	
	`salesrep/name`
	
	Indices that use this path type are more specific because queries that cross these edge boundaries only have to examine one index entry for the document instead of two.

Given this, use:

- node path types to improve queries where there can be no overlap in the node name. That is, if the query is based on an element or attribute that appears on only one context within the document, then use node path types.

  In the preceding sample document, you would want to use node-type indices with the address, city, state, zipcode, and salesrep elements because they appear in only one context within the document.

- edge path types to improve query performance when a node name is used in multiple contexts within the document. In the preceding document, use edge path types for the name and phonenumber elements because they appear in multiple (2) contexts within the document.

Node Types

BDB XML can index three types of nodes: element, attribute, or metadata. Metadata nodes are, of course, indices set for a document's metadata content.

Element and Attribute Nodes

Element and attribute nodes are only found in document content. In the following document:

```
<vendor type="wholesale">
    <name>TriCounty Produce</name>
</vendor>
```

vendor and name are element nodes, while type is an attribute node.

Use the element node type to improve queries that test the value of an element node. Use the attribute node type to improve any query that examines an attribute or attribute value.

Metadata Nodes

Metadata nodes are found only in a document's metadata content. This indices improve the performance of querying for documents based on metadata information. If you are declaring a metadata node, you cannot use a path type of edge.

Key Types

The Key type identifies what sort of test the index supports. You can use one of three key types:

Key Type	Description
equality	Improves the performances of tests that look for nodes with a specific value.

Key Type	Description
`presence`	Improves the performance of tests that look for the existence of an node, regardless of its value.
`substring`	Improves the performance of tests that look for a node whose value contains a given substring. This key type is best used when your queries use the XQuery `contains()` substring function.

Syntax Types

Beyond the index type, you must also identify the syntax type. The syntax describes what sort of data the index will contain, and it is mostly used to determine how indexed values are compared. There are a large number of syntax types available to you, such as `substring`, `boolean`, or `date`.

For a complete list of the syntax types available to you, see Using Strings to Specify Indices (page 59) or Using Enumerated Types to Specify Indices (page 61).

Specifying Index Strategies

The combined index type and syntax type is called the *index strategy*. There are two ways that you can specify an index; by using a string or by specifying enumerated types. Most of the APIs that you use to manage indices support both mechanisms. However, a few support only strings.

Using Strings to Specify Indices

The string that you use to specify an indexing strategy is formatted as follows:

```
[unique]-{path type}-{node type}-{key type}-{syntax type}
```

where:

- `unique` is the actual value that you provide in this position on the string. If you provide this value, then indexed values must be unique. If you do not want indexed values to be unique, provide nothing for this position in the string.

  See Uniqueness (page 56) for more information.

- `{path type}` identifies the path type. Valid values are:

  - `node`
  - `edge`

  See Path Types (page 57) for more information.

- `{node type}` identifies the type of node being indexed. Valid values are:

  - `element`

  - `attribute`
  - `metadata`

  If metadata is specified, then {path type} must be `node`.

  See Node Types (page 58) for more information.

- {key type} identifies the sort of test that the index supports. The following key types are supported:
  - `presence`
  - `equality`
  - `substring`

  See Key Types (page 58) for more information.

- {syntax type} identifies the syntax to use for the indexed value. Specify one of the following values:
  - `none`
  - `anyURI`
  - `base64Binary`
  - `boolean`
  - `date`
  - `dateTime`
  - `dayTimeDuration`
  - `decimal`
  - `double`
  - `duration`
  - `float`
  - `gDay`
  - `gMonth`
  - `gMonthDay`
  - `gYear`

- gYearMonth
- hexBinary
- NOTATION
- QName
- string
- time
- yearMonthDuration
- untypedAtomic

Note that if the key type is presence, then the syntax type should be none.

The following are some example index strategies:

- node-element-presence-none

  Index an element node for presence queries. That is, queries that test whether the node exists in the document.

- unique-node-metadata-equality-string

  Index a metadata node for equality string compares. The value provided for this node must be unique within the container.

  This strategy is actually used by default for all documents in a container. It is used to index the document's name.

- edge-attribute-equality-float

  Defines an equality float index for an attribute's edge. Improves performance for queries that examine whether a specific element/@attribute path is equal to a float value.

Also, be aware that you can specify multiple indices at a time by providing a space-separated list of index strategies in the string. For example, you can specify two index strategies at a time using:

```
"node-element-presence-none edge-attribute-equality-float"
```

Using Enumerated Types to Specify Indices

When you use enumerated types to specify index strategies, the API will provide two parameters. The first defines the index type, and the second defines the index syntax.

You specify the index syntax using an XmlValue::Type value. See Enumerated Index Syntax (page 63) for more information.

Be aware that, unlike index strategies specified as strings, you cannot specify multiple index strategies in a single API call when you are using enumerated types.

Enumerated Index Types

To specify an index type, you provide a series of `XmlIndexSpecification::Type` values *or*'d together. Each value that you provide identifies a specific aspect of the index type: uniqueness, path type, node type, and key type. To specify:

- uniqueness, provide `XmlIndexSpecification::UNIQUE_ON` to the index type value. If you provide nothing, then uniqueness is not enforced. For code readability purposes, you can optionally use `XmlIndexSpecification::UNIQUE_OFF` to suppress uniqueness as well.
- the path type, use one of:
  - `XmlIndexSpecification::PATH_NODE`
  - `XmlIndexSpecification::PATH_EDGE`
- the node type, use one of:
  - `XmlIndexSpecification::NODE_ELEMENT`
  - `XmlIndexSpecification::NODE_ATTRIBUTE`
  - `XmlIndexSpecification::NODE_METADATA`

  Note that if `XmlIndexSpecification::NODE_METADATA` is used, then `XmlIndexSpecification::PATH_NODE` must also be used.
- the key type, use one of:
  - `XmlIndexSpecification::KEY_PRESENCE`
  - `XmlIndexSpecification::KEY_EQUALITY`
  - `XmlIndexSpecification::KEY_SUBSTRING`.

For example, to specify the type, *or* the values together like this:

```
XmlIndexSpecification::UNIQUE_ON |
XmlIndexSpecification::PATH_NODE |
XmlIndexSpecification::NODE_ATTRIBUTE |
XmlIndexSpecification::KEY_EQUALITY
```

See Enumerated Index Example (page 64) for an example of how to use these types.

Enumerated Index Syntax

To identify the syntax type for the index strategy using enumerated types, use `XmlValue::Type` values. The following values are available for you to use:

- `XmlValue::NONE`
- `XmlValue::NODE`
- `XmlValue::ANY_SIMPLE_TYPE`
- `XmlValue::ANY_URI`
- `XmlValue::BASE_64_BINARY`
- `XmlValue::BOOLEAN`
- `XmlValue::DATE`
- `XmlValue::DATE_TIME`
- `XmlValue::DAY_TIME_DURATION`
- `XmlValue::DECIMAL`
- `XmlValue::DOUBLE`
- `XmlValue::DURATION`
- `XmlValue::FLOAT`
- `XmlValue::G_DAY`
- `XmlValue::G_MONTH`
- `XmlValue::G_MONTH_DAY`
- `XmlValue::G_YEAR`
- `XmlValue::G_YEAR_MONTH`
- `XmlValue::HEX_BINARY`
- `XmlValue::NOTATION`
- `XmlValue::QNAME`
- `XmlValue::STRING`
- `XmlValue::TIME`
- `XmlValue::YEAR_MONTH_DURATION`

- `XmlValue::UNTYPED_ATOMIC`

Enumerated Index Example

Methods that accept numerated types for specifying index strategies always have a parameter for the index type and a parameter for the index syntax. The following are examples of the values you would give to these parameters when specifying an index strategy:

- String equivalent: `node-element-presence-none`

  For index type:

  ```
  XmlIndexSpecification::PATH_NODE |
  XmlIndexSpecification::NODE_ELEMENT |
  XmlIndexSpecification::KEY_PRESENCE
  ```

  For parameter type:

  ```
  XmlValue::NONE
  ```

- String equivalent: `unique-node-metadata-equality-string`

  ```
  XmlIndexSpecification::UNIQUE_ON |
  XmlIndexSpecification::PATH_NODE |
  XmlIndexSpecification::NODE_METADATA |
  XmlIndexSpecification::KEY_EQUALITY
  ```

  For parameter type:

  ```
  XmlValue::STRING
  ```

- `edge-attribute-equality-float`

  ```
  XmlIndexSpecification::EDGE_NODE |
  XmlIndexSpecification::NODE_ATTRIBUTE |
  XmlIndexSpecification::KEY_EQUALITY
  ```

  For parameter type:

  ```
  XmlValue::FLOAT
  ```

Specifying Index Nodes

It is possible to have BDB XML build indices at a node granularity rather than a document granularity. The difference is that document granularity is good for retrieving large documents while node granularity is good for retrieving nodes from within documents.

Indexing nodes can only be performed if your containers are performing node-level storage. You should consider using node indices if you have a few large documents stored in your

containers and you will be performing queries intended to retrieve subsections of those documents. Otherwise, you should use document level indexes.

Because node indices can actually be harmful to your application's performance, depending on the actual read/write activity on your containers, expect to experiment with your indexing strategy to find out whether node or document indexes work best for you.

Node indices contain a little more information, so they may take more space on disk and could also potentially take longer to write. For example, consider the following document:

```
<names>
    <name>joe</name>
    <name>joe</name>
    <name>fred</name>
</names>
```

If you are using document-level indexing, then there is one index entry for each *unique* value ocurring in the document for a given index. So if you have a string index on the `name` element, the above document would result in two index entries; one for `joe` and another for `fred`.

However, for node-level indices, there is one index entry for each node regardless of whether it is unique. Therefore, given an a string index on the `name` element, the above document would result in three index entries.

Given this, imagine that the document in use had 1000 `name` elements, 500 of which contained `joe` and 500 of which contained `fred`. For document-level indexing, you would still only have two index entries, while for node-level indexing you would have 1000 index entries per stored document. Whether the considerably larger size of the node-level index is worthwhile is something that you would have to evaluate based on the number of documents you are storing and the nature of your query patterns.

Note that by default, containers do not use node-level indices. You specify that you want node-level indices by using the `DBXML_INDEX_NODES` flag when you create the container.

You can tell whether a container is using node-level indices using the `XmlContainer::getIndexNodes()` method. If the container is creating node-level indices, this method will return `true`.

If your containers are using node-level storage, you can switch between node-level indices and document-level indices using `XmlManager::reindexContainer()`. Specify `DBXML_INDEX_NODES` to cause a the container to use node-level indices. To switch from node-level to document-level indices, use `DBXML_NO_INDEX_NODES`. Note that this method causes your container to be completely re-indexed. Therefore, for containers containing large amount of data, or large numbers of indices, or both, this method should not be used routinely as it may take some time to write the new indices.

Indexer Processing Notes

As you design your indexing strategy, keep the following in mind:

- As with all indexing mechanisms, the more indices that you maintain the slower your write performance will be. Substring indices are particularly heavy relative to write performance.
- The indexer does not follow external references to document type definitions and external entities. References to external entities are removed from the character data. Pay particular attention to this when using equality and substring indices as element and attribute values (as indexed) may differ from what you expect.
- The indexer substitutes internal entity references with their replacement text.
- The indexer concatenates character data mixed with child data into a single value. For example, as indexed the fragment:

```
<node1>
    This is some text with some
    <inline>inline </inline> data.
</node1>
```

  has two elements. <node1> has the value:

  "This is some text with some data"

  while <inline> has the value:

  "inline"

- The indexer expands CDATA sections. For example, the fragment:

```
<node1>
    Reserved XML characters are <![CDATA['<', '>', and '&']]>
</node1>
```

  is indexed as if <node1> has the value:

  "Reserved XML characters are '<', '>', and '&'"

- The indexer replaces namespace prefixes with the namespace URI to which they refer. For example, the `class` attribute in the following code fragment:

```
<node1 myPrefix:class="test"
xlmns:myPrefix="http://dbxmlExamples/testPrefix />
```

  is indexed as

```
<node1 http://dbxmlExamples/testPrefix:class="test"
xlmns:myPrefix="http://dbxmlExamples/testPrefix />
```

  This normalization ensures that documents containing the same element types, but with different prefixes for the same namespace, are indexed as if they were identical.

Managing BDB XML Indices

Indices are set for a container using the container's index specification. You can specify an index either against a specific node and namespace, or you can define default indices that are applied to every node in the container.

You add, delete, and replace indices using the container's index specification. You can also iterate through the specification, so as to examine each of the indices declared for the container. Finally, if you want to retrieve all the indices maintained for a named node, you can use the index specification to find and retrieve them.

An API exists that allows you to retrieve all of the documents or nodes referenced by a given index.

For simple programs, managing the index specification and then setting it to the container (as is illustrated in the following examples) can be tedious. For this reason, BDB XML also provides index management functions directly on the container. Which set of functions your application uses is entirely up to your requirements and personal tastes.

Performing index modifications (for example, adding and replacing indices) on a container that already contains documents can be a very expensive operation — especially if the container holds a large number of documents, or very large documents, or both. This is because indexing a container requires BDB XML to traverse every document in the container.

If you are considering re-indexing a large container, be aware that the operation can take a long time to complete.

Adding Indices

To add an index to a container:

1. Retrieve the index specification from the container.
2. Use `XmlIndexSpecification::addIndex()` to add the index to the container. You must provide to this method the namespace and node name to which the index is applied. You must also identify the indexing strategy.

   If the index already exists for the specified node, then the method silently does nothing.
3. Set the updated index specification back to the container.

For example:

```
#include "DbXml.hpp"
...

using namespace DbXml;

...

// Get a manager object.
```

```
XmlManager myManager;

// Open a container
XmlContainer myContainer =
    myManager.openContainer("exampleData.dbxml");

// Get the index specification
XmlIndexSpecification is = myContainer.getIndexSpecification();

// Get the index type and syntax. This is the equivalent of
//    "node-element-presence-none"
// index strategy

XmlIndexSpecification::Type idxType =
    (XmlIndexSpecification::Type)
    (XmlIndexSpecification::PATH_NODE |
    XmlIndexSpecification::NODE_ELEMENT |
    XmlIndexSpecification::KEY_PRESENCE);

// The syntax must be NONE because this is a presence
// index.
XmlValue::Type syntaxType = XmlValue::NONE;

// Add the index. We're indexing "node1" using the default
// namespace. Note that we could also do this using:
// is.addIndex("", "node1", "node-element-presence-none");
is.addIndex("", "node1", idxType, syntaxType);

// Save the index specification back to the container.
XmlUpdateContext uc = myManager.createUpdateContext();
myContainer.setIndexSpecification(is, uc);
```

Deleting Indices

To delete an index from a container:

1. Retrieve the index specification from the container.
2. Use `XmlIndexSpecification::deleteIndex()` to delete the index from the index specification.
3. Set the updated index specification back to the container.

For example:

```
#include "DbXml.hpp"
...

using namespace DbXml;
```

```
...

// Get a manager object.
XmlManager myManager;

// Open a container
XmlContainer myContainer =
    myManager.openContainer("exampleData.dbxml");

// Get the index specification
XmlIndexSpecification is = myContainer.getIndexSpecification();

// We will delete the equivalent of:
// XmlIndexSpecification::Type idxType =
//     (XmlIndexSpecification::Type)
//     (XmlIndexSpecification::PATH_NODE |
//     XmlIndexSpecification::NODE_ELEMENT |
//     XmlIndexSpecification::KEY_PRESENCE);
// XmlValue::Type syntaxType = XmlValue::NONE;

// Delete the index. We're deleting the index from "node1" in
// the default namespace that has the syntax strategy identified
// above. Note that we could also do this using:
// is.deleteIndex("", "node1", idxType, syntaxType);
is.deleteIndex("", "node1", "node-element-presence-none");

// Save the index specification back to the container.
XmlUpdateContext uc = myManager.createUpdateContext();
myContainer.setIndexSpecification(is, uc);
```

Replacing Indices

You can replace the indices maintained for a specific node by using `XmlIndexSpecification::replaceIndex()`. When you replace the index for a specified node, all of the current indices for that node are deleted and the replacement index strategies that you provide are used in their place.

Note that all the indices for a specific node can be retrieved and specified as a space separated list in a single string. So if you set a node-element-equality-string and a node-element-presence index for a given node, then it's indices are identified as:

```
"node-element-equality-string node-element-presence"
```

For example:

```
#include "DbXml.hpp"
...

using namespace DbXml;
```

```
...

// Get a manager object.
XmlManager myManager;

// Open a container
XmlContainer myContainer =
    myManager.openContainer("exampleData.dbxml");

// Get the index specification
XmlIndexSpecification is = myContainer.getIndexSpecification();

// Replace the index.
std::string idxString =
    "node-element-equality-string node-element-presence";
is.replaceIndex("", "node1", idxString);

// Save the index specification back to the container.
XmlUpdateContext uc = myManager.createUpdateContext();
myContainer.setIndexSpecification(is, uc);
```

Examining Container Indices

You can iterate over all the indices in a container using `XmlIndexSpecification::next()`. You can retrieve indices using either the string or enumerated format.

For example:

```
#include "DbXml.hpp"
...

using namespace DbXml;

...

// Get a manager object.
XmlManager myManager;

// Open a container
XmlContainer myContainer =
    myManager.openContainer("exampleData.dbxml");

// Get the index specification
XmlIndexSpecification is = myContainer.getIndexSpecification();

// Iterate over all of the indices in the container. Note
// that we could use the enumerated types to retrieve
// the indices as well.
std::string uri, name, index;
```

```
int count = 0;
while(is.next(uri,name,index)) {
    // Print the index strategy to the console:
    std::cout << "For node: '" << name << "' found:\n"
              << "\tURI: " << uri
              << "\tIndex: " << index << std::endl;
    count ++;
}
std::cout << count << " indices found." << std::endl;
```

Working with Default Indices

Default indices are indices that are applied to all applicable nodes in the container that are not otherwise indexed. For example, if you declare a default index for a metadata node, then all metadata nodes will be indexed according to that indexing strategy, unless some other indexing strategy is explicitly set for them. In this way, you can avoid the labor of specifying a given indexing strategy for all occurrences of a specific kind of a node.

You add, delete, and replace default indices using:

- `XmlIndexSpecification::addDefaultIndex()`
- `XmlIndexSpecification::deleteDefaultIndex()`
- `XmlIndexSpecification::replaceDefaultIndex()`

When you work with a default index, you identify only the indexing strategy; you do not identify a URI or node name to which the strategy is to be applied.

Note that just as is the case with other indexing methods, you can use either strings or enumerated types to identify the index strategy.

For example, to add a default index to a container:

```
#include "DbXml.hpp"
...

using namespace DbXml;

...

// Get a manager object.
XmlManager myManager;

// Open a container
XmlContainer myContainer =
    myManager.openContainer("exampleData.dbxml");

// Get the index specification
```

```
XmlIndexSpecification is = myContainer.getIndexSpecification();

// Get the index type and syntax. This is the equivalent of
//    "node-metadata-equality-string"
// index strategy
XmlIndexSpecification::Type idxType =
    (XmlIndexSpecification::Type)
    (XmlIndexSpecification::PATH_NODE |
    XmlIndexSpecification::NODE_METADATA |
    XmlIndexSpecification::KEY_EQUALITY);

// Declare the syntax type:
XmlValue::Type syntaxType = XmlValue::STRING;

// Add the default index. Note that we could also do this using:
// is.addDefaultIndex("node-metadata-equality-string");
is.addDefaultIndex(idxType, syntaxType);

// Save the index specification back to the container.
XmlUpdateContext uc = myManager.createUpdateContext();
myContainer.setIndexSpecification(is, uc);
```

Looking Up Indexed Documents

You can retrieve all of the values referenced by an index using an `XmlIndexLookup` object, which is returned by the `XmlManager::createIndexLookup()` method. `XmlIndexLookup` allows you to obtain an `XmlResults` object that contains all of the nodes or documents for which the identified index has keys. Whether nodes or documents is return depends on several factors:

- If your container is of type `WholedocContainer`, then entire documents are always returned in this method's results set.
- If your container is of type `NodeContainer`, and if you specified `DBXML_INDEX_NODES` when you created your container, then this method returns the nodes to which the index's keys refer.

For example, every container is created with a default index that ensures the uniqueness of the document names in your container. The:

- URI is `http://www.sleepycat.com/2002/dbxml`.
- Node name is `name`.
- Indexing strategy is `unique-node-metadata-equality-string`.

Given this, you can efficiently retrieve every document in the container using `XmlIndexLookup`. as follows:

```
#include "DbXml.hpp"
...

using namespace DbXml;

...

// Get a manager object.
XmlManager myManager;

// Open a container
XmlContainer myContainer =
    myManager.openContainer("exampleData.dbxml");

XmlQueryContext qc = myManager.createQueryContext();

// Lookup the index
std::string uri = "http://www.sleepycat.com/2002/dbxml";
std::string name = "name";
std::string idxStrategy = "unique-node-metadata-equality-string";

// Get the XmlIndexLookup Object
XmlIndexLookup xil = myManager.createIndexLookup(myContainer, uri, name,
    idxStrategy);

// Now look it up. This returns every document in the container.
XmlResults res = xil.execute(qc);

// Iterate over the results set, printing each document in it
XmlDocument thedoc = myManager.createDocument();
while (res.next(thedoc)) {
    std::string dummyString;
    std::cout << thedoc.getName() << ": "
              << thedoc.getContent(dummyString) << std::endl;
}
```

In the event that you want to lookup an edge index, you must provide the lookup method with both the node and the parent node that together comprise the XML edge.

For example, suppose you have the following document in your container:

```
<mydoc>
   <node1>
        <node2>
            node2 1
        </node2>
        <node2>
            node2 2
        </node2>
```

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

Further suppose you indexed the presence of the node1/node2 edges. In this case, you can lookup the values referred to by this index by doing the following:

```
#include "DbXml.hpp"
...

using namespace DbXml;

...

// Get a manager object.
XmlManager myManager;

// Open a container
XmlContainer myContainer =
    myManager.openContainer("exampleData.dbxml");

XmlQueryContext qc = myManager.createQueryContext();

// Node to lookup
std::string uri = "";
std::string name = "node2";

// Parent node to lookup
std::string parentURI = "";
std::string parentName = "node1";

std::string idxStrategy = "edge-element-presence";

// Get the XmlIndexLookup Object
XmlIndexLookup xil = myManager.createIndexLookup(myContainer, uri, name,
    idxStrategy);

// Identify the parent node
xil.setParent(parentURI, parentName);

// Now look it up.
XmlResults res = xil.execute(qc);

// Iterate over the results set, printing each value in it
XmlValue retValue;
while (res.next(retValue)) {
    std::cout << "Found: " << retValue.asString() << std::endl;
}
```

Verifying Indices using Query Plans

When designing your indexing strategy, you should create indices to improve the performance of your most frequently occurring queries. Without indices, BDB XML must walk every document in the container in order to satisfy the query. For containers that contain large numbers of documents, or very large documents, or both, this can be a time-consuming process.

However, when you set the appropriate index(es) for your container, the same query that otherwise takes minutes to complete can now complete in milliseconds. So setting the appropriate indices for your container is a key ingredient to improving your application's performance.

That said, the question then becomes, how do you know that a given index is actually being used by a given query? That is, how do you do this without loading the container with enough data that it is noticeably faster to complete a query with an index set than it is to complete the query without the index?

The way to do this is to examine BDB XML's query plan for the query to see if it intends to use an index for the query. And the best and easiest way to examine a query plan is by using the **dbxml** command line utility.

Query Plans

The query plan is literally BDB XML's plan for how it will satisfy a query. When you use `XmlManager::prepare()`, one of the things you are doing is regenerating a query plan so that BDB XML does not have to continually re-create it every time you run the query.

Printed out, the query plan looks like an XML document that describes the steps the query processor will take to fulfill a specific query.

For example, suppose your container holds documents that look like the following:

```
<a>
    <docId id="aaUivth" />
    <b>
        <c>node1</c>
        <d>node2</d>
    </b>
</a>
```

Also, suppose you will frequently want to retrieve the document based on the value set for the `id` parameter on the `docId` node. That is, you will frequently perform queries that look like this:

```
collection("myContainer.dbxml")/a/docId[@id='bar']
```

In this case, if you print out the query plan (we describe how to do this below), you will see something like this:

```
<XQuery>
  <Navigation>
    <QueryPlanFunction result="collection">
      <OQPlan>U</OQPlan>
      <ImpliedSchema>
        <root nodeType="*">
          <child name="a">
            <child name="docId">
              <attribute name="id" nodeType="*">
                <equals/>
              </attribute>
              <descendant uri="*" name="*" nodeType="*"/>
            </child>
          </child>
        </root>
      </ImpliedSchema>
    </QueryPlanFunction>
    <Step axis="child" name="a" nodeType="element"/>
    <Step axis="child" name="docId" nodeType="element">
      <Predicates>
        <Operator name="equal">
          <Step axis="attribute" name="id" nodeType="attribute"/>
          <Sequence>
            <AnyAtomicTypeConstructor value="aaUivth"
            typeuri="http://www.w3.org/2001/XMLSchema"
            typename="string"/>
          </Sequence>
        </Operator>
      </Predicates>
    </Step>
  </Navigation>
</XQuery>
```

While a complete description of the query plan is outside the scope of this manual, we draw your attention to the highlighted element near the top of the query plan:

```
<OQPlan>U</OQPlan>
```

This is the part of the query plan that identifies what, if any, indices will be consulted in order to satisfy the query. Because the text value for this element is only `U`, this query plan is *not* using an index in order to satisfy the query. This means that BDB XML will have to examine every document in the container in order satisfy the query.

Now suppose you add an index designed to support this query:

- URI is "".
- Node name is `id`.
- Indexing strategy is `"node-attribute-equality-string"`

The query plan for:

```
collection("myContainer.dbxml")/a/docId[@id='bar']
```

now reads:

```
<XQuery>
  <Navigation>
    <QueryPlanFunction result="collection">
      <OQPlan>V(node-attribute-equality-string,@id,=,'aaUivth')</OQPlan>
      <ImpliedSchema>
        <root nodeType="*">
          <child name="a">
            <child name="docId">
              <attribute name="id" nodeType="*">
                <equals/>
              </attribute>
              <descendant uri="*" name="*" nodeType="*"/>
            </child>
          </child>
        </root>
      </ImpliedSchema>
    </QueryPlanFunction>
    <Step axis="child" name="a" nodeType="element"/>
    <Step axis="child" name="docId" nodeType="element">
      <Predicates>
        <Operator name="equal">
          <Step axis="attribute" name="id" nodeType="attribute"/>
          <Sequence>
            <AnyAtomicTypeConstructor
            value="aaUivth"
            typeuri="http://www.w3.org/2001/XMLSchema"
            typename="string"/>
          </Sequence>
        </Operator>
      </Predicates>
    </Step>
  </Navigation>
</XQuery>
```

Notice that the `OQPlan` element now shows our index. This indicates that the BDB XML query processor will use that index in order to satisfy the query.

Using the dbxml Shell to Examine Query Plans

dbxml is a command line utility that allows you to gracefully interact with your BDB XML containers. You can perform a great many operations on your containers and documents using this utility, but of interest to the current discussion is the utility's ability to allow you add and delete indices to your containers, to query for documents, and to examine query plans.

The BDB XML command line utilities, including **dbxml**, are described here:
http://www.sleepycat.com/xmldocs/utility/index.html

Note that while you can create containers and load XML documents into those containers using **dbxml**, we assume here that you have already performed these activities using some other mechanism.

In order to examine query plans using **dbxml**, do the following:

```
> dbxml -h myEnvironment

dbxml>
```

To begin, open your container:

```
dbxml> open myContainer.dbxml
```

Next, examine your query plan using the **qplan** command. Note that we assume your container only has the standard, default index that all containers have when they are first created.

```
dbxml> qplan collection("myContainer.dbxml")/a/docId[@id='aaUivth']
<XQuery>
  <Navigation>
    <QueryPlanFunction result="collection">
      <OQPlan>U</OQPlan>
      <ImpliedSchema>
        <root nodeType="*">
          <child name="a">
            <child name="docId">
              <attribute name="id" nodeType="*">
                <equals/>
              </attribute>
              <descendant uri="*" name="*" nodeType="*"/>
            </child>
          </child>
        </root>
      </ImpliedSchema>
    </QueryPlanFunction>
    <Step axis="child" name="a" nodeType="element"/>
    <Step axis="child" name="docId" nodeType="element">
      <Predicates>
        <Operator name="equal">
          <Step axis="attribute" name="id" nodeType="attribute"/>
          <Sequence>
            <AnyAtomicTypeConstructor value="aaUivth"
            typeuri="http://www.w3.org/2001/XMLSchema"
            typename="string"/>
          </Sequence>
        </Operator>
      </Predicates>
```

```
    </Step>
  </Navigation>
</XQuery>
```

Notice that this query plan does not make use of an index (the `OQPlan` element is empty.) Now add the index that you want to test.

```
dbxml> addindex "" id "node-attribute-equality-string"
Adding index type: node-attribute-equality-string to node: {}:id
```

Now try the query plan again. Notice that `OQPlan` is now referencing our new index.

```
dbxml> qplan collection("myContainer.dbxml")/a/docId[@id='aaUivth']
<XQuery>
  <Navigation>
    <QueryPlanFunction result="collection">
      <OQPlan>
        V(node-attribute-equality-string,@id,=,'aaUivth')
      </OQPlan>
      <ImpliedSchema>
        <root nodeType="*">
          <child name="a">
            <child name="docId">
              <attribute name="id" nodeType="*">
                <equals/>
              </attribute>
              <descendant uri="*" name="*" nodeType="*"/>
            </child>
          </child>
        </root>
      </ImpliedSchema>
    </QueryPlanFunction>
    <Step axis="child" name="a" nodeType="element"/>
    <Step axis="child" name="docId" nodeType="element">
      <Predicates>
        <Operator name="equal">
          <Step axis="attribute" name="id" nodeType="attribute"/>
          <Sequence>
            <AnyAtomicTypeConstructor value="aaUivth"
            typeuri="http://www.w3.org/2001/XMLSchema"
            typename="string"/>
          </Sequence>
        </Operator>
      </Predicates>
    </Step>
  </Navigation>
</XQuery>
```

You are done testing your index. To exit **dbxml**, use the **quit** command:

```
dbxml> quit
```

Chapter 8. Using Transactions

Transactions allow you to treat one or more operations on one or more containers as a single unit of work. The BDB XML transactional subsystem is simply a wrapper around Berkeley DB's transactional subsystem. This means that you BDB XML offers the same, full, ACID protection as does Berkeley DB. That is, BDB XML transactions offer you:

- **A**tomicity.

  Multiple container operations (most importantly, write operations) are treated as a single unit of work. In the event that you abort a transaction, all write operations performed during the transaction are discarded. In this event, your container is left in the state it was in before the transaction began, regardless of the number or type of write operations that you may have performed during the course of the transaction.

  Note that BDB XML transactions can span one or more `Container` handles. Also, transactions can span both containers and Berkeley DB databases, provided they exist within the same environment.

- **C**onsistency.

  Your BDB XML containers will never see a partially completed transactions, no matter what happens to your application. This is true even if your application crashes while there are in-progress transactions. If the application or system fails, then either all of the container changes appear when the application next runs, or none of them appear.

- **I**solation.

  While a transaction is in progress, your containers will appear to the transaction as if there are no other operations are occurring outside of the transaction. That is, operations wrapped inside a transaction will always have a clean and consistent view of your databases. They never have to contend with partially updated records (unless you want them to).

- **D**urability.

  Once committed to your containers, your modifications will persist even in the event of an application or system failure. Note that durability is available only if your application performs a sync when it commits a transaction.

Transactionally processing is covered in great detail in the *Berkeley DB Programmer's Reference Guide*. All of the concepts and topics described there are relevant to transactionally protecting an BDB XML application.

The next few sections describe topics that are specific to transactionally protecting a BDB XML application.

Initializing the Transactional Subsystem

In order to use transactions, you must turn on the transactional subsystem. You do this when you open your `XmlManager` by setting the appropriate flags for the manager. You must also turn on transactions for your container when you open it, again through the use of the appropriate flags.

Note that if you do not enable transactions when you first create your environment, then you cannot subsequently use transactions. Also, if your environment is not opened to support transactions, then your containers cannot be opened to support transactions. Finally, you cannot transactionally protect your container operations unless your environment and containers are configured to support transactions.

One final point: the default `XmlManager` constructor does not enable the transactional subsystem for its underlying environment, and there is no way to pass the appropriate flags to that environment using the default constructor. Instead, you must construct your own `DbEnv` object, passing it the flags required to enable transactions, and then hand that `DbEnv` object to the `XmlManager` constructor.

In order to enable transactions, you must enable the memory pool (the cache), the logging subsystem, the locking subsystem, and the transactional subsystem. For example:

```
#include "DbXml.hpp"
...

using namespace DbXml;
int main(void)
{
    u_int32_t env_flags = DB_CREATE     |  // If the environment does not
                                           // exist, create it.
                          DB_INIT_LOCK  |  // Initialize locking
                          DB_INIT_LOG   |  // Initialize logging
                          DB_INIT_MPOOL |  // Initialize the cache
                          DB_INIT_TXN;     // Initialize transactions

    std::string envHome("/export1/testEnv");
    DbEnv myEnv(0);
    XmlManager *myManager = NULL;

    try {
        myEnv.open(envHome.c_str(), env_flags, 0);
        myManager = new XmlManager(myEnv, 0);
    } catch(DbException &e) {
        std::cerr << "Error opening database environment: "
                  << envHome << std::endl;
        std::cerr << e.what() << std::endl;
    } catch(XmlException &e) {
        std::cerr << "Error opening database environment: "
                  << envHome
```

```
                  << " or opening XmlManager." << std::endl;
        std::cerr << e.what() << std::endl;
    }

    try {
        if (myManager != NULL) {
            delete myManager;
        }
        myEnv.close(0);
    } catch(DbException &e) {
        std::cerr << "Error closing database environment: "
                << envHome << std::endl;
        std::cerr << e.what() << std::endl;
    } catch(XmlException &e) {
        std::cerr << "Error closing database environment: "
                << envHome << std::endl;
        std::cerr << e.what() << std::endl;
    }
}
```

Once you have enabled transactions for your environment and your manager, you must enable transactions for the containers that you open. You do this by providing the `DBXML_TRANSACTIONAL` flag when you create or open the container.

The following code updates the previous example to also open a container. The new code is shown in **bold**.

```
#include "DbXml.hpp"
...

using namespace DbXml;
int main(void)
{
    u_int32_t env_flags = DB_CREATE     |  // If the environment does not
                                           // exist, create it.
                          DB_INIT_LOCK  |  // Initialize locking
                          DB_INIT_LOG   |  // Initialize logging
                          DB_INIT_MPOOL |  // Initialize the cache
                          DB_INIT_TXN;     // Initialize transactions

    std::string envHome("/export1/testEnv");
    DbEnv myEnv(0);
    XmlManager *myManager = NULL;

    try {
        myEnv.open(envHome.c_str(), env_flags, 0);
        myManager = new XmlManager(myEnv);

        u_int32_t containerFlags =
```

```
            DB_CREATE |        // If the container does not exist,
                               // create it.
            DB_TRANSACTIONAL;  // Enable transactions.

        std::string containerName = "myContainer.dbxml";
        XmlContainer myContainer =
            myManager.openContainer(containerName, containerFlags);

    } catch(DbException &e) {
        std::cerr << "Error opening database environment: "
                  << envHome << std::endl;
        std::cerr << e.what() << std::endl;
    } catch(XmlException &e) {
        std::cerr << "Error opening database environment: "
                  << envHome
                  << " or opening XmlManager." << std::endl;
        std::cerr << e.what() << std::endl;
    }

    try {
        if (myManager != NULL) {
            delete myManager;
        }
        myEnv.close(0);
    } catch(DbException &e) {
        std::cerr << "Error closing database environment: "
                << envHome << std::endl;
        std::cerr << e.what() << std::endl;
    } catch(XmlException &e) {
        std::cerr << "Error closing database environment: "
                << envHome << std::endl;
        std::cerr << e.what() << std::endl;
    }
}
```

Transactionally Protecting Container Operations

To transactionally protect one or more container operations, do the following:

1. Open your environment and containers such that they support transactions, as described in the previous section.

2. Create an `XmlTransaction` object. These objects are created using `XmlManager::createTransaction()`.

3. Perform your operations, handing the `XmlTransaction` object to each container read and write method that is participating in the transaction.

Be aware that you can use the same `XmlTransaction` for read and write operations performed on different containers and databases, provided that those containers and databases all exist in the same environment; there is no limit to the number of containers and databases that can participate in the transaction.

4. When you have performed all of your transaction's operations, call `XmlManager::commit()` to commit the transaction. This causes the write operations performed under the protection of the transaction to be written to the files backing your containers and databases on disk.

   Once committed, the `XmlTransaction` object is no longer valid. That is, you cannot reuse it. If you want to perform another transaction, you must instantiate another `XmlTransaction` object.

5. If the operations participating in your transaction should throw an exception or otherwise indicate an operational failure, terminate the transaction by calling `XmlManager::abort()`. This causes all of the write operations performed under the control of the transaction to be discarded.

   As is the case with `XmlManager::commit()`, an aborted `XmlTransaction` object is no longer valid and can not be reused.

Note that when you create an `XmlTransaction` object, you can create a transaction based on an existing `DbEnv` object. If you do this, then the following rules apply:

- Any handle for a transaction object can commit or abort that transaction. Once committed or aborted, all handles to the transaction are no longer valid.
- If the `XmlTransaction` object goes out of scope without being committed or aborted, then the external `DbEnv` object that was used to create it is still valid and the underlying transaction is still active (until such a time as the transaction is either committed or aborted in some other location in your code).
- Likewise, if the parent `DbEnv` object goes out scope while the `XmlTransaction` object is still active, then the underlying transaction is still active until such a time as the `XmlTransaction` object calls either commit or abort.
- If all `XmlTransaction` objects go out of scope and there are no in-scope `DbEnv` objects, then the underlying transaction is automatically aborted.

Never perform both transactional and non-transactional writes on the same container. Doing so can cause your underlying databases to no longer be recoverable in the event that recovery is needed.

If you open a container transactionally, and you do not provide a transaction for your write operations, BDB XML will automatically transactionally protect that write for you. However, if you perform writes to a transactional container, close the container and then open it without transactional support, then the writes to that container are not protected by a transaction.

The following provides an example of how to create, use, commit, and abort a transaction:

```
#include "DbXml.hpp"
...

using namespace DbXml;
int main(void)
{
    ...
    // Environment, manager, and container opens omitted
    // for brevity
    ...

    std::string file1 = "doc1.xml";
    std::string file2 = "doc2.xml";
    XmlTransaction txn;
    try {
        txn = myManager.createTransaction();
        // Need an update context for the put.
        XmlUpdateContext theContext = myManager.createUpdateContext();

        // Get the first input stream.
        XmlInputStream *theStream =
            myManager.createLocalFileInputStream(file1);
        // Put the first document
        myContainer.putDocument(txn,         // the transaction object
                                file1,       // The document's name
                                theStream,   // The actual document.
                                theContext,  // The update context
                                             // (required).
                                0);          // Put flags.

        // Get the second input stream.
        theStream = myManager.createLocalFileInputStream(file2);
        // Put the second document
        myContainer.putDocument(txn,         // the transaction object
                                file2,       // The document's name
                                theStream,   // The actual document.
                                theContext,  // The update context
                                             // (required).
                                0);          // Put flags.
        // Finished. Now commit the transaction.
        txn.commit();

    } catch(XmlException &e) {
        std::cerr << "Error in transaction: "
                  << e.what() << "\n"
                  << "Aborting." << std::endl;
        txn.abort();
    }
}
```

Transactions Considerations

Transactionally protecting your container operations is an important ingredient to ensuring the integrity of your containers and databases. However, be aware that transactions may impact your application's performance.

The *Berkeley DB Programmer's Reference Guide* contains a couple of sections that can help you understand the performance impact transactions can have on your application. See the following sections in the *Berkeley DB Programmers Reference Guide* for this information:

- *Transaction Tuning* (http://www.sleepycat.com/docs/ref/transapp/tune.html)
- *Transaction throughput* (http://www.sleepycat.com/docs/ref/transapp/throughput.html)

The next several section in this guide provides a rough introduction to this information.

There are two areas of consideration where it comes to transactional performance. The first is disk I/O and the second has to do with lock contention.

Transaction Disk I/O

Normally when you perform a write to a BDB XML container, the write is not written to disk until a sync is called on the in-memory cache. This syncing occurs either when you force it by using `DbEnv::sync`, or when your environment is closed. (Note that you can suppress the sync when you close your environment, but this is not the normal case.)

When you transactionally protect your database writes, however, the data modified by the write is written to disk every time the transaction is committed. For applications that run for extremely long periods of time, and which perform relatively few write operations, this can will improve your application's performance because the commit only writes those portions of the cache that were dirtied (written) by the transaction. A full sync, on the other hand, writes the entire cache to disk which is considerably more expensive than the partial write performed by a commit.

Transaction and Lock Contention

Because transactions guarantee isolation from all other threads of control, they must perform locking, and hold those locks for the duration of the transaction. Holding these locks may cause other thread of control to have to wait in order to be able to access the locked data. How much this affects your application will depend on its data access patterns.

Additionally, with transactional applications, it is possible that conflicting lock requests from different threads of control can cause a deadlock to occur. To understand more about deadlocks and how to handle them, please refer to the *Deadlock detection* section of the *Berkeley DB Programmer's Reference Guide* (available at: http://www.sleepycat.com/docs/ref/transapp/deadlock.html).

The performance penalty that you might pay due to the additional locking required by your transactions is dependent on a number of factors:

- The amount of time that your transaction lives. If your transaction is short-lived (the ideal situation), then there is less chance that it will be holding a lock required by another transaction.
- The number of operations performed by the transaction. A transaction that must read and write hundreds of documents will hold considerably more locks for potentially longer periods of time than an application that reads and writes only a few documents.
- The number of transactions (typically this means threads of control) in existence at any given time. The more transactions there are, the greater the chance for deadlock contention.

Given this, for best results try to use only short-lived transactions. Also, try to keep the number of operations performed by your transactions small, or try to keep the number of transactions in existence small.

Index Operations and Transactions

One final thing to consider when using transactions with BDB XML has to do with re-indexing containers. If you are performing index add, delete, or replace operations on a very large container (tens of thousands of documents or greater), and you are using transactions to protect these operations, then the operation can potentially fail with the following error message:

```
Lock table is out of available locks
```

When you perform an index operation on a container, you are reading and writing every document node in the container. This means that you are asking Berkeley DB to read and write every record in the underlying database.

Every time Berkeley DB performs a read or a write operation, it acquires one or more locks on the database pages on which it is operating. Normally, Berkeley DB releases those locks once it has completed the operation. However, as discussed above, when you use transactions to protect write operations, Berkeley DB holds all locks that it acquires until the transaction completes (is either committed or aborted).

Locks are a finite resource, and so Berkeley DB maintains an internal data structure that identifies how many locks it can use at any given time. By default, this number is 1,000 locks.

The end result is, if you are performing index operations on large containers and you are using transactions to protect those operations, you can run out of locks. When this happens, Berkeley DB fails the operation with the above noted error message.

To work around this problem, you must increase the number of locks available to Berkeley DB. You do this with `DbEnv::set_lk_max_locks()`. See the online Berkeley DB documentation for more information.