

Oracle Berkeley DB XML

Getting Started with Berkeley DB XML for C++

12c Release 1

Library Version 12.1.6.1



Legal Notice

This documentation is distributed under an open source license. You may review the terms of this license at: <http://www.oracle.com/technetwork/database/berkeleydb/downloads/xmllicense-086890.html>

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

Other names may be trademarks of their respective owners.

To obtain a copy of this document's original source code, please submit a request to the Oracle Technology Network forum at: https://community.oracle.com/community/database/high_availability/berkeley_db_family/berkeley_db_xml

Published 19-Jul-2016

Table of Contents

Preface	vi
Conventions Used in this Book	vi
For More Information	vi
Contact Us	vii
1. Introduction to Berkeley DB XML	1
Overview	1
Benefits	2
XML Features	2
Database Features	3
Languages and Platforms	4
Getting and Using BDB XML	4
Documentation and Support	4
Library Dependencies	4
Building and Running BDB XML Applications	5
2. Exception Handling and Debugging	6
Debugging BDB XML Applications	6
3. XmlManager and Containers	8
XmlManager	8
Berkeley DB Environments	8
Environment Open Flags	9
Opening and Closing Environments	10
XmlManager Instantiation and Destruction	11
Managing Containers	13
Container Properties	14
Container Types	15
Deleting and Renaming Containers	16
4. Adding XML Documents to Containers	18
Input Streams and Strings	18
Adding Documents	18
Constructing Documents using Event Writers	21
Setting Metadata	23
5. Using XQuery with BDB XML	26
XQuery: A Brief Introduction	26
Referencing Portions of Documents using XQuery	27
Predicates	27
Numeric Predicates	27
Boolean Predicates	28
Context	28
Relative Paths	28
Namespaces	28
Wildcards	30
Case Insensitive Searches	31
Navigation Functions	31
collection()	31
doc()	32
Using FLWOR with BDB XML	32

Retrieving BDB XML Documents using XQuery	33
The Query Context	33
Defining Namespaces	34
Defining Variables	34
Defining the Evaluation Type	35
Performing Queries	36
Metadata Based Queries	38
Working with External Functions	39
Implementing XmlExternalFunction	39
Implementing XmlResolver	40
Calling External Functions from XQuery	42
Examining Query Results	43
Examining Document Values	44
Examining Metadata	47
Copying Result Sets	48
Using Event Readers	49
6. Managing Documents in Containers	53
Deleting Documents	53
Replacing Documents	54
Modifying XML Documents	55
XQuery Update Introduction	55
Inserting Nodes Using XQuery Update	55
Position Keywords	57
Insertion Rules	57
Deleting Nodes Using XQuery Update	58
Replacing Nodes Using XQuery Update	58
Replacement Rules	59
Renaming Nodes Using XQuery Update	59
Updating Functions	60
Transform Functions	60
Resolving Conflicting Updates	61
Compressing XML Documents	62
Turning Compression Off	62
Using Custom Compression	63
7. Using BDB XML Indices	66
Index Types	66
Uniqueness	66
Path Types	67
Node Types	68
Element and Attribute Nodes	68
Metadata Nodes	68
Key Types	68
Syntax Types	69
Specifying Index Strategies	69
Specifying Index Nodes	71
Indexer Processing Notes	72
Automatic Indexes	73
Managing BDB XML Indices	74
Adding Indices	75

Deleting Indices	75
Replacing Indices	76
Examining Container Indices	77
Working with Default Indices	78
Looking Up Indexed Documents	79
Verifying Indices using Query Plans	81
Query Plans	82
Using the dbxml Shell to Examine Query Plans	83
8. Administering Berkeley DB XML Applications	85
Temporary Files	85
A Note on Snapshot Isolation	86

Preface

Welcome to Berkeley DB XML (BDB XML). This document introduces BDB XML, version 6.0. 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;
}
```

For More Information

Beyond this manual, you may also find the following sources of information useful when building a BDB XML application:

- [Introduction to Berkeley DB XML](#)
- [Berkeley DB XML Getting Started with Transaction Processing for C++](#)
- [Berkeley DB XML C++ API Reference Guide](#)

To download the latest Berkeley DB XML documentation along with white papers and other collateral, visit <http://www.oracle.com/technetwork/indexes/documentation/index.html>.

For the latest version of the Oracle Berkeley DB XML downloads, visit <http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/downloads/index.html>.

Contact Us

You can post your comments and questions at the Oracle Technology (OTN) forum for Oracle Berkeley DB XML at: https://community.oracle.com/community/database/high_availability/berkeley_db_family/berkeley_db_xml.

For sales or support information, email to: berkeleydb-info_us@oracle.com You can subscribe to a low-volume email announcement list for the Berkeley DB product family by sending email to: bdb-join@oss.oracle.com

Chapter 1. Introduction to Berkeley DB XML

Welcome to 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 BDB XML 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 way as you would any third-party library.

In BDB XML documents are stored in *containers*, which you create and manage using `Xm1Manager` 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 nodes, 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.

BDB XML supports the entire XQuery specification. You can read the specification here:

<http://www.w3.org/TR/xquery/>

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 3.0 language. XQuery is a W3C draft specification (<http://www.w3.org/XML/Query>). The queries are encoded in Unicode UTF-8.
- **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 your 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. Also note that queries on the XML documents are always encoded as Unicode UTF-8, regardless of the underlying encoding of the document.

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.

BDX 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

BDX 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 [BDX XML product page](#).

Documentation and Support

BDX XML is officially described in the [product documentation](#). For additional help and for late-breaking news on language and platform support, it is best to use the [BDX XML forums](#).

Library Dependencies

BDX 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](#). Berkeley DB provides the underlying database support for BDB XML. BDB XML supports Berkeley DB version 4.3 or later.

- [Xerces](#). Xerces provides the DOM and SAX support that BDB XML employs for XML data parsing. Xerces 3.0.1 or later is required for BDB XML.
- [XQilla](#). BDB XML's XQuery support is provided by this library.
- [ZLIB](#). This library is used to support compression in BDB XML. Note that this library is optional in that you can turn off zlib support when you compile BDB XML.

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

Note

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://docs.oracle.com/cd/E17276_01/html//toc.htm

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.

```
#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
    }
}
```

Note that, you can obtain more information on the cause of the `XmlException` by examining the underlying error code. Do this using the `XmlException::getExceptionCode()` method. See the *Berkeley DB XML C++ API Reference Guide* reference for details on the exception codes available through this class.

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. For this reason, it is always a good idea to create a custom error handler.

Once you have implemented your error handler, you make it known to your BDB XML application using `DB_ENV->set_errcall()`

For example:

```
#include "DbXml.hpp"

using namespace DbXml;

void
errcall_fun(const DB_ENV *dbenv,
            const char *errpfx,
            const char *msg) {
    std::cerr << errpfx << " : " << msg << std::endl;
}

int main(void)
{
    // Open an XmlManager
    DB_ENV *myEnv;
    XmlManager myManager;

    myEnv = myManager.getDB_ENV();
    myEnv->set_errcall(myEnv, errcall_fun);
}
```

Once you have set up your error handler, you can control the amount of information that BDB XML reports to that handler 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).

```
#include "DbXml.hpp"

using namespace DbXml;
int main(void)
{
    ...
    // Skipped environment and manager open
    ...
    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 \(page 26\)](#)).
- Create a transaction object (see the *Berkeley DB XML Getting Started with Transaction Processing* guide for details).

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 XML Getting Started with Transaction Processing* guide.

- **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 writing transactional applications in the *Berkeley DB XML Getting Started with Transaction Processing* guide.

- **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 any 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

char *envHome = "/export1/testEnv";
DB_ENV *myEnv = NULL;
int dberr;

dberr = db_env_create(&myEnv, 0);
if (dberr) {
    std::cout << "Unable to create environment: " <<
                db_strerror(dberr) << std::endl;
    if (myEnv)
        myEnv->close(myEnv, 0);
    return (EXIT_FAILURE);
}
```

```
myEnv->open(myEnv, envHome, env_flags, 0);

// Do BDB XML work here

myEnv->close(myEnv, 0);
```

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 constructor with an open DB_ENV object. If you do instantiate XmlManager with an opened environment handle, then XmlManager will close and destroy that DB_ENV object for you if you specify DBXML_ADOPT_DBENV for the XmlManager constructor.

If you provide an DB_ENV 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 can optionally 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 DB_ENV 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

    char *envHome = "/export1/testEnv";
    DB_ENV *myEnv = NULL;
    int dberr;

    XmlManager *myManager = NULL;

    dberr = db_env_create(&myEnv, 0);
    if (dberr) {
        std::cout << "Unable to create environment: " <<
            db_strerror(dberr) << std::endl;
    }
    if (myEnv)
        myEnv->close(myEnv, 0);
    return (EXIT_FAILURE);

    myEnv->open(myEnv, envHome, env_flags, 0);
    myManager =
        new XmlManager(myEnv,
                       DBXML_ADOPT_DBENV); // The manager and
                                           // environment is closed
                                           // when this object is
```

```

// destroyed.

try {
    if (myManager != NULL) {
        delete myManager;
    }
} catch(XmlException &xe) {
    std::cerr << "Error closing XmlManager: "
               << 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 properties to allow the container to be created (see the following section for information on container open properties).

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 Properties

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

- `XmlContainerConfig::setAllowCreate()`

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

- `XmlContainer::setExclusiveCreate()`

Causes the container creation to fail if the container already exists. It is not necessary to specify this property on the call to `XmlManager::createContainer()`. Note that this property is meaningless unless `XmlContainerConfig::setAllowCreate()` is also used.

- `XmlContainerConfig::setReadOnly()`

The container is opened for read-access only.

- `XmlContainerConfig::setAllowValidation()`

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

- `XmlContainerConfig::setIndexNodes()`

Determines whether indexes for the container will return nodes (if this property is set to `XmlContainerConfig::On`) or documents (if this property is set to `XmlContainerConfig::Off`).

Note that the default index type is determined by the type of container you are creating. If you are creating a container of type `NodeContainer`, then this property is set to `XmlContainerConfig::On` by default. For containers of type `WholedocContainer`, this property is set to `XmlContainerConfig::Off` by default.

If you want to change this property on an existing container, you must re-index the container in order for the new index type to take effect.

For more information on index nodes, see [Specifying Index Nodes \(page 71\)](#).

- `XmlContainerConfig::setTransactional()`

The container supports transactions. For more information, see the *Berkeley DB XML Getting Started with Transaction Processing* guide.

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**

XML documents 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. Somehow obtain the document that you want to put into the container. To do this, you can create an input stream to the content or load the XML document into a string object. Alternatively, you can load the document into an `XmlDocument` object and then provide the `XmlDocument` object to `XmlContainer::putDocument()`. When you do this, you can provide the document to the `XmlDocument` object using an input stream or string, or you can construct the document using an event writer.

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()` is read and validated. By default, this includes any schema or DTDs that the document might reference. Since this can cause you some performance issues, you can cause BDB XML to only examine the document body itself by passing the `DBXML_WELL_FORMED_ONLY` flag to `XmlContainer::putDocument()`. However, using this flag cause parsing errors if the document references information that might have come from a schema or DTD.

Further, 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.
}

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.
}

return(0);
}
```

Constructing Documents using Event Writers

In the previous section we showed you how to load a document into a container by reading that document from disk, or by providing the document as a string object. As an alternative, you can construct your document using an `XmlEventWriter` class object, which stores the

document in an `XmlDocument` object. You can then put that `XmlDocument` object to the container as described in the previous section.

`XmlEventWriter` provides methods that allow you to describe the individual, discrete sections of the document. It is useful if, for example, you are already parsing a document using a SAX parser and you want to write the information your parser discovers to a container.

To use an event writer:

1. Create the `XmlDocument` instance.
2. Give it a name using the `XmlDocument::setName()` method.
3. Put the document to your container using the `XmlContainer::putDocumentAsEventWriter()` method. Note that at this point you have not actually written any document data to the container, since your document is currently empty.

This method returns an `XmlEventWriter` object.

4. Use the `XmlEventWriter` object to start new document. You do this using the `XmlEventWriter::writeStartDocument()` method, which allows you to describe information about the XML document such as its encoding and its XML version identification.
5. Once you have started your document, you can write beginning and end elements, attributes, processing instructions, text, CDATA, and all the other features that you might expect to place on an XML document. `XmlEventWriter` provides methods that allow you to do these things.
6. Once you have completed your document, close it using the `XmlEventWriter::close()` method. This completes the container put operation that you began in step 3.

For example, suppose you wanted to write the following document to a container:

```
<a>
<b a1="one" b2="two">b node text</b>
<c>c node text</c>
</a>
```

Then the following code fragment would accomplish that task:

```
// create a new document
XmlDocument doc = mgr.createDocument();
doc.setName(dname);

XmlEventWriter &writer = cont.putDocumentAsEventWriter(doc, uc);
writer.writeStartDocument(NULL, NULL, NULL); // no XML decl

// Write the document's root node. It has no prefixes or
// attributes. This node is not empty.
writer.writeStartElement((const unsigned char *)"a", NULL, NULL,
```

```
    0, false);

    // Write a new start element. This time for the "b" node.
    // It has two attributes and its content is also not empty.
    writer.writeStartElement((const unsigned char *)"b", NULL, NULL,
        2, false);
    // Write the "a1" and "b2" attributes on the "b" node
    writer.writeAttribute((const unsigned char *)"a1", NULL, NULL,
        (const unsigned char *)"one", true);
    writer.writeAttribute((const unsigned char *)"b2", NULL, NULL,
        (const unsigned char *)"two", true);
    // Write the "b" node's content. Note that there are 11
    // characters in this text, and we provide that information
    // to the method. Also, we identify this data as being of type
    // XmlEventType.Characters.
    writer.writeText(Characters, (const unsigned char *)"b node text",
        11);
    // End the "b" node
    writer.writeEndElement((const unsigned char *)"b", NULL, NULL);
    // Start the "c" node. There are no attributes on this node.
    writer.writeStartElement((const unsigned char *)"c", NULL, NULL,
        0, false);
    // Write the "c" node's content
    writer.writeText(Characters, (const unsigned char *)"c node text",
        11);
    // End the "c" node and then the "a" (the root) node
    writer.writeEndElement((const unsigned char *)"c", NULL, NULL);
    writer.writeEndElement((const unsigned char *)"a", NULL, NULL);

    // End the document
    writer.writeEndDocument();
    // Close the document
    writer.close();
```

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 \(page 66\)](#) 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.
}

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. Queries on a BDB XML container are encoded as Unicode UTF-8.

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 partially implements XQuery 3.0. 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 \(page 33\)](#) 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.
4. Modify a document (see [Modifying XML Documents \(page 55\)](#) for more information).

To do this, XQuery views an XML document as a collection of element, text, and attribute nodes. For example, consider the following 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 [XQuery: A Brief Introduction \(page 26\)](#). 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. 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 the document presented in [XQuery: A Brief Introduction \(page 26\)](#) 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:

```
.
```

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:

```
<?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:

```
<?xml version="1.0"?>
<definition>
    <jewelry:ring xmlns:jewelry="http://myDefinition.dbxml/jewelry">
        Jewelry that you wear.
    </jewelry: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
```

Note

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 \(page 34\)](#) 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:

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

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

```
<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/@*

Case Insensitive Searches

It is possible to perform a case-insensitive and diacritic insensitive match using BDB XML's built-in function, `dbxml:contains()`. This function takes two parameters, both strings. The first identifies the attribute or element that you want to examine, and the second provides the string you want to match.

For example, the search:

```
collection('myCollection.dbxml')/book[dbxml:contains(title, "Résumé")]
```

matches "resume", "Resume", "Résumé" and so forth.

Note that searches performed using `dbxml:contains()` can be backed by BDB XML's substring indexes.

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 can 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 \(page 33\)](#) 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 \(page 33\)](#) 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 \(page 31\)](#).

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 \(page 34\)](#)).

- Defining whether the query is processed "eagerly" or "lazily" (see [Defining the Evaluation Type \(page 35\)](#)).

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 \(page 44\)](#) 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");
```

Note

If you pass an empty prefix to `setNamespace()`, the URI you provide is set as the default URI.

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.

```

#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 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 iterate 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 one or more containers, documents, or nodes. Usually you use one of the XQuery navigation functions to do this. See [Navigation Functions \(page 31\)](#) for more information.

Note

You can configure the query 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, use `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::prepare()`.

```
#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);
```

Finally, note that when you perform a query, by default BDB XML will read and validate the document and any attached schema or DTDs. This can cause performance problems, so to avoid it you can pass the `DBXML_WELL_FORMED_ONLY` flag to `XmlQueryExpression::execute()`. This can improve performance by causing the scanner to examine only the XML document itself, but it can also cause parsing errors if the document references information that might have come from a schema or DTD.

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);
```

Working with External Functions

BDX XML allows you to define your own functions that you can access from your XQueries. To do this, you must provide an implementation of `XmlExternalFunction`, and you must implement a `XmlResolver` class that resolves which external function to call.

Implementing `XmlExternalFunction`

`XmlExternalFunction` implementations only require you to implement the `execute()` method with your function code. You must also implement a `close()` method that cleans up after whatever activities your `execute()` method calls.

The `execute()` method offers three parameters:

- `XmlTransaction`

This is the transaction in use, if any, at the time the external function was called.

- `XmlManager`

The `XmlManager` instance in use at the time the function was called.

- `XmlArguments`

An array of `XmlResults` objects which hold the argument values needed by this function.

For example, suppose you wanted to write an external function that takes two numbers and returns the first number to the power of the second number. It would look like this:

```
class MyExternalFunctionPow : public XmlExternalFunction
{
public:
    XmlResults execute(XmlTransaction &txn, XmlManager &mgr,
        const XmlArguments &args) const;
    void close();
};

/* External function pow() implementation */
XmlResults MyExternalFunctionPow::execute(XmlTransaction &txn,
    XmlManager &mgr, const XmlArguments &args) const
{
    XmlResults argResult1 = args.getArgument(0);
    XmlResults argResult2 = args.getArgument(1);
```

```

    XmlValue arg1;
    XmlValue arg2;

    // Retrieve argument as XmlValue
    argResult1.next(arg1);
    argResult2.next(arg2);

    // Call pow() from C++
    double result = pow(arg1.asNumber(),arg2.asNumber());

    // Create an XmlResults for return
    XmlResults results = mgr.createResults();
    XmlValue va(result);
    results.add(va);

    return results;
}

/*
 * In this example implementation, the resolver will return a new
 * instance of this function every time this function is called
 * so this function is required to clean up after itself.
 *
 * You could alternatively write the resolver such that it deletes
 * all function instances when the resolver is destroyed. If you
 * did that, the implementation of this method is not required.
 */
void MyExternalFunctionPow::close()
{
    delete this;
}

```

Implementing XmlResolver

The `XmlResolver` class is used to provide a handle to the appropriate external function, when a given XQuery statement requires an external function. For this reason, your `XmlResolver` implementation must have knowledge of every external function you have implemented.

The resolver is responsible for instantiating an instance of the required external function. It is also responsible for destroying that instance, either once the query has been process or when the resolver instance itself is being destroyed. Which is the correct option for your application is an implementation detail that is up to you.

It is possible for your code to have multiple instances of an `XmlResolver` class, each instance of which can potentially be responsible for different collections of external functions. For this reason, you uniquely identify each resolver class with a URI.

In order to call a specific external function, your XQueries must provide a URI as identification, as well as a function name. You can decide which external function to return

based on the URI, the function name, and/or the number of arguments provided in the XQuery. Which of these are necessary for you to match the correct external function is driven by how many external functions you have implemented, how many resolver classes you have implemented, and how many variations on functions with the same name you have implemented. In theory, a very simple implementation could return an external function instance based only on the function name. Other implementation may need to match based on all possible criteria.

For the absolute most correct and safest implementation, you should match on all three criteria: URI, function name, and number of arguments.

For example, suppose you had two external functions: `SmallFunction` and `BigFunction`. `SmallFunction` is a small function that requires few resources to instantiate and is called infrequently. `BigFunction` is a larger function that opens containers, obtains lots of memory and from a performance perspective is something that is best instantiated once and then not destroyed until program shutdown. Further, `SmallFunction` takes two arguments while `BigFunction` takes five.

And `XmlResolver` implementation for this example would be as follows:

```
// Class declaration
class MyFunResolver : public XmlResolver
{
public:
    MyFunResolver();
    XmlExternalFunction *resolveExternalFunction(XmlTransaction *txn,
        XmlManager &mgr, const std::string &uri, const std::string &name,
        size_t numberOfArgs) const;
    string getUri(){ return uri_; }
private:
    const string uri_;
    XmlExternalFunction *bigFun_;
};

// Class constructor. The only required thing is to initialize the private
// data member, uri_. We also instantiate a BigFunction instance in the
// constructor, just because in our imaginary example we know we will
// always need an instance of this reusable function.
MyFunResolver::MyFunResolver() :
    uri_("my://my.fun.resolver"),
    bigFun_(0)
{
    bigFun_ = new BigFunction();
}

// Class destroyer
MyFunResolver::~MyFunResolver() {
    bigFun_->close();
    delete bigFun_;
}
```

```
// Resolver implementation. Here, we match based on the URI, the name
// of the function, and the number of arguments. However, for a simple
// example such as this, we could potentially match on just the function
// name.
XmlExternalFunction*
MyFunResolver::resolveExternalFunction(XmlTransaction *txn,
    XmlManager &mgr, const std::string &uri,
    const std::string &name, size_t numberOfArgs) const
{
    XmlExternalFunction *fun = 0;

    if (uri == uri_ && name == "sfunc" && numberOfArgs == 2 ) {
        fun = new SmallFunction();
    } else if (uri == uri_ && name == "bfunc" && numberOfArgs == 5) {
        return bigFun_;
    }

    return fun;
}
```

Calling External Functions from XQuery

In order to use your external functions, you must register the resolver that manages them. You do this with the `XmlManager::registerResolver()` method. You then set a URI prefix for the URI that you use to identify your resolver. For example:

```
try {

    // Create an XmlManager
    XmlManager mgr;

    // Create an function resolver
    MyFunResolver resolver;

    // Register the function resolver to XmlManager
    mgr.registerResolver(resolver);

    XmlQueryContext context = mgr.createQueryContext();

    // Set the prefix URI
    context.setNamespace("myxfunc", resolver.getUri());
```

To use the external function, declare them in the preamble of your query, and then use them as you would any XQuery function (for a complete explanation of examining query results, see the next section). For example:

```
declare function myxfunc:sfunc($a as xs:double, $b as xs:double) \
    as xs:double external;
myxfunc:sfunc(2,3);
```

You run this query as if you were running any other query.

```
string query = "declare function ";
query += "myxfunc:sfunc($a as xs:double, $b as xs:double) ";
query += "as xs:double external;\nmy:pow(2,3)";

// The first query returns the result of pow(2,3)
XmlResults results = mgr.query(query, context);

XmlValue va;
while (results.next(va)) {
    cout << "The result of sfunc(2,3) is : "
          << va.asNumber() << endl;
}
} catch (XmlException &xe) {
    cout << "XmlException: " << xe.what() << endl;
}
return 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. Iterate over the result set as normal.
2. 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 newRetVal;

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

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

```

Copying Result Sets

When you create an `XmlResults` object by executing a query, the object actually references database objects. That is, the object is non-transient which means that if the objects in the

database are modified or deleted in some way, then the contents of your `XmlResults` object can also be modified or deleted.

One way to guard against this is to use transactions to provide isolation guarantees for your `XmlResults` objects. Transactions are described in the *Berkeley DB XML Getting Started with Transaction Processing* guide.

Another way to guard against this is to create a transient copy of your `XmlResults` object. You do this by using the `XmlResults::copyResults()` method. This method causes all of the `XmlValue` objects contained in the results set to no longer be references to database objects. As a result, you can safely use the result set outside of a transaction, and you can modify the copied results set without concern that you are modifying the container.

This method simply returns a new `XmlResults` object, which you can use in the same way you would use any `XmlResults` object.

```
...  
  
XmlResults results = myManager.query(myQuery, context);  
XmlResults transResults = results.copyResults();  
  
...
```

It is also possible to concatenate two results sets together using the `XmlResults::concatResults()` method. This method is can only be used with transient results sets (that is, `XmlResults` objects created using the `copyResults()` method. In this case, the results set provided as an argument to the `concatResults()` method is concatenated to the `XmlResults` object that the method is called on.

```
...  
  
XmlResults results1 = myManager.query(myQuery1, context);  
XmlResults results2 = myManager.query(myQuery2, context);  
XmlResults transResults1 = results1.copyResults();  
XmlResults transResults2 = results2.copyResults();  
  
// Concatenate results2 to results1  
transResults1.concatResults(transResults2);  
  
...
```

Using Event Readers

Once you have retrieved a document or node, you can examine that retrieved item using an *event reader*. Event readers provide a pull interface that allows you to move through a document, or a portion of a document, using an iterator-style interface.

When you iterate over a document or node using an event reader, you are examining individual objects in the document. In this, the event reader behaves much like a SAX parser in that it allows you to discover what sort of information you are examining (for example, a start element, an end element, whitespace, characters, and so forth), and then retrieve

relevant information about that data. (Note, however, that the event reader interface differs significantly from SAX in that SAX is a push interface while `XmlEventReader` is a pull interface.)

The document events for which you can test using the event reader are:

- `StartElement`
- `EndElement`
- `Characters`
- `CDATA`
- `Comment`
- `Whitespace`
- `StartDocument`
- `EndDocument`
- `StartEntityReference`
- `EndEntityReference`
- `ProcessingInstruction`
- `DTD`

In addition for testing for specific portions of a document, you can also retrieve information about those portions of the document. For example, if you are examining a starting element, you can retrieve the name of that element. You can also retrieve an attribute count on that element, and then retrieve information about each attribute based on its indexed value in the start node. That is, suppose you have the following document stored in a container:

```
<a>
<b a1="one" b2="two">b node</b>
<c>c node</c>
</a>
```

Then you can examine this document as follows:

```
try {
    // Container declaration and open omitted for brevity
    ...
    std::string docname = "doc1";
    XmlDocument xdoc = container.getDocument(docname);

    // Get an XmlEventReader
    XmlEventReader &reader = xdoc.getContentAsEventReader();

    // Now iterate over the document elements, examining only
```

```

// those of interest to us:
while (reader.hasNext()) {
    XmlEventType type = reader.next();
    if (type == StartElement) {
        std::cout << "Found start node: " <<
            reader.getLocalName() << std::endl;
        std::cout << "There are " << reader.getAttributeCount()
            << " attributes on this node." << endl;
        // Show all the attributes on the start element node
        for (int i = 0; i < reader.getAttributeCount(); i++) {
            std::cout << "Attribute '"
                << reader.getAttributeLocalName(i)
                << "' has a value of '"
                << reader.getAttributeValue(i)
                << "'" << endl;
        }
    }
}

// When we are done, we close the reader to free-up resources.
reader.close();
} catch (XmlException &e) {
    std::cout << "Exception: " << e.what() << std::endl;
}

```

Running this code fragment yields:

```

Found start node: a
There are 0 attributes on this node.
Found start node: b
There are 2 attributes on this node.
Attribute 'a1' has a value of 'one'
Attribute 'b2' has a value of 'two'
Found start node: c
There are 0 attributes on this node.

```

Note that you can also use event readers on `XmlValue` objects, provided that the object is an element node. For example:

```

try {
    // Container declaration and open omitted for brevity
    // As are the manager, query and XmlQueryContext
    // declarations.
    ...
    XmlResults res = mgr.query(myquery, context);
    XmlValue val;
    while ((val = res.next()) != NULL) {
        if (val.isNode() &&
            (val.getNodeType() == XmlManager.ELEMENT_NODE)) {
            XmlEventReader &reader = val.asEventReader();

```

```
        // Now iterate over the document elements
        while (reader.hasNext()) {
            XmlEventType type = reader.next();

            // Handle each event type as required by your
            // application.
        }
        // When we are done, we close the reader to free-up resources.
        reader.close();
    }
}

} catch (XmlException &e) {
    std::cout << "Exception: " << e.what() << 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 \(page 55\)](#).

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 `XmlDocument::setContent()` 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.

Note

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 allows you to modify documents already stored in its containers using XQuery Update statements. This section provides a brief introduction to update statements so as to help you get going with them.

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

XQuery Update Introduction

XQuery Update allows you to insert, delete, replace and rename nodes using built-in keywords (insert, delete, replace and rename, respectively). You can also perform a node update by declaring an update function.

XQuery Update does not perform updates (node insertion, deletion, and so forth) until after the query has completed. This means a couple of things. First, you cannot perform an update and return the results in the same query.

Also, update statements are order independent, although in some cases conflicting updates are performed in an order defined by the XQuery Update statement specification).

Finally, updates are generally expected to be performed in isolation from other queries. You can not, for example, perform a search and then in a subsequent statement perform an update, all in the same query.

Note

XQuery Update is described in the W3C specification, *XQuery Update Facility 1.0*. This specification is currently a working draft. BDB XML implements the version of the specification dated [28 August 2007](#)

Inserting Nodes Using XQuery Update

To insert a node into an existing document, you must identify the node that you want to insert, and the location in the document where you want the insertion to be performed. You indicate that you are performing an insertion operation using the XQuery insert keyword.

The general format of this expression is:

```
insert nodes nodes keyword position
```

where

- *nodes* is the content that you want to insert. This can be a string, or it can be an XQuery selection statement.
- *keywords* indicates how you would like the new content to be inserted.
- *position* indicates the document and the location in that document where the insertion is to occur.

Be aware that *position* must be an XQuery expression that selects exactly one location in the document. Also, *keywords* can be one of several keywords that indicate where the new content is to be inserted relative to the location in the document that is indicated by *position*. See the next section for information on the available keywords.

For example, consider the document:

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

Assuming this document is called 'mydoc.xml', then you can insert a node, b4 after node b2 using the following query expression:

```
insert nodes <b4>inserted child</b4> after
doc("dbxml:/con.dbxml/mydoc.xml")/a/b2
```

The above expression applied to the XML document results in a document like this:

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

Note that if the query expression provided above happens to match more than one node, then the query will fail. For multiple node insertions, use an XQuery FLWOR expression. For example if our original working document is:

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

Then to insert a node after every <b2> node in the document, use this:

```
for $i in doc('dbxml:/con.dbxml/mydoc.xml')/a/b2 return
insert nodes <b4>inserted child</b4> after $i
```

This results in the document:

```
<a>
```



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

Position Keywords

XQuery Update expressions that add content to a document must first select the location in the document where the content is to be added, and then it must identify where the content is to be added relative to the selected location. You do this by specifying the appropriate keywords to the update expression.

Valid keywords are:

- before

The new content precedes the target node.

- after

The new content follows the target node.

- as first into

The new content becomes the first child of the target node.

- as last into

The new content becomes the last child of the target node.

- into

The new content is inserted as the last child of the target node, provided that this keyword is not used in an update expression that also makes use of the keywords noted above. If that happens, the node is inserted so that it does not interfere with the indicated position of the other new nodes.

Note that the behavior described here is an artifact of BDB XML's current implementation of the XQuery Update specification. The specification does not require the inserted node to be placed as the last child of the target node, so this behavior may change for some future release of the product.

Insertion Rules

When inserting elements, the selection expression must be non-updating, and it must not result in an empty set.

If any form of the `into` keyword is specified, the selection expression must result in a single element or document node. Also, if `before` or `after` is provided, the selection expression result must be a single element, text, comment or processing instruction node.

If an attribute node is selected, then the new content must provide an attribute.

Deleting Nodes Using XQuery Update

You can delete zero or more nodes using a `delete nodes` query. For example, given the document named "mydoc.xml" in container "con.dbxml":

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

The following query deletes the `b4` node:

```
delete nodes doc("dbxml:/con.dbxml/mydoc.xml")/a/b4
```

Note that if the document had more than one `<b4>` node, then they all would be deleted by this query.

The selection expression that you provide must be a non-updating expression, and the result must be a sequence of zero or more nodes. If the selection expression selects a node that has no parent, then the result is to delete the entire document from the container.

Replacing Nodes Using XQuery Update

You can use XQuery Update statements to either replace an entire node, or a node's value. To replace a node, use the `replace node` query. For example, given the document named "mydoc.xml" in container "con.dbxml":

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

You can replace node `b2` with a different node such as `<r1>replacement child</r1>` using the following query:

```
replace node doc("dbxml:/con.dbxml/mydoc.xml")/a/b2
with <z1>replacement node</z1>
```

The result of this replacement query is:

```
<a>
  <b1>first child</b1>
  <z1>replacement node</z1>
  <b3>third child</b3>
</a>
```

To replace multiple nodes in the document, use an XQuery `FLWOR` statement like this:

```
for $i in doc("dbxml:/con.dbxml/mydoc.xml")/a/b2 return
replace node $i with <z1>replacement node</z1>
```

The replacement value can also be a selection expression. For example, suppose you had a second document named `replace.xml`:

```
<a>
  <rep>more replacement data</rep>
</a>
```

Then you can replace node z1 with the rep node using the following query:

```
replace node doc("dbxml:/con.dbxml/mydoc.xml")/a/z1
with doc("dbxml:/con.dbxml/replace.xml")/a/rep
```

Or, as an XQuery FLWOR expression:

```
for $i in doc("dbxml:/con.dbxml/mydoc.xml")/a/z1 return
replace node $i with doc("dbxml:/con.dbxml/replace.xml")/a/rep"
```

Either expression results in the document:

```
<a>
  <b1>first child</b1>
  <rep>more replacement data</rep>
  <b3>third child</b3>
</a>
```

In addition to the replace node ... with ... form, you can also replace node values. Do this using replace value of node ... with ... queries.

For example, to replace the value of the rep node, above, use:

```
replace value of node doc("dbxml:/con.dbxml/mydoc.xml")/a/rep
with "random replacement text".
```

The results of this query is:

```
<a>
  <b1>first child</b1>
  <rep>random replacement text</rep>
  <b3>third child</b3>
</a>
```

Replacement Rules

When replacing elements, the selection expression used to select the target must be non-updating, and it must not result in an empty set.

Selection results must consist of a single element, text, comment or processing instruction. In addition, the selection expression must not select a node without a parent node.

Finally, If you replace an attribute node, its replacement value must not have a namespace property that conflicts with the namespaces property of the parent node.

Renaming Nodes Using XQuery Update

You can rename a node using rename node query. For example, given the document named "mydoc.xml" in container "con.dbxml":

```
<a>
  <b1>first child</b1>
```

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

You can rename node b3 to z1 using the following query:

```
rename node doc("dbxml:/con.dbxml/mydoc.xml")/a/b3 as "z1"
```

Or, as an XQuery FLWOR expression if you are renaming multiple nodes:

```
for $i in doc("dbxml:/con.dbxml/mydoc.xml")/a/b3 return
rename node $i as "z1"
```

The selection expression that you provide must be a non-updating expression, and the result must be non-empty and consist of a single element, attribute, or processing instruction node.

Updating Functions

You can create a function that performs an update, so long as it is declared to be an updating function. In addition, this function must not have a return value, and the argument passed to the function cannot be an update query.

For example, the following query creates a function that renames any element node passed to it, to the value passed in the second argument. The function is then called for b1 in document mydoc.xml, which is stored in container con.dbxml:

```
declare updating function
  local:renameNode($elem as element(),
                  $rep as xs:string)
{
  rename node $elem as $rep
};

local:renameNode(doc("dbxml:/con.dbxml/mydoc.xml")/a/b1, "aab1")
```

If the prior query is called on a document such as this:

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

then that document becomes:

```
<a>
  <aab1>first child</aab1>
  <b2>second child</b2>
  <b3>third child</b3>
</a>
```

Transform Functions

While it is true that you cannot run an update query and simultaneously return the results, there is a way to almost do the same thing. You do this by making a copy of the nodes that

you want to modify, then perform the modifications against that copy. The result of the modification is returned to you. This type of an operation is called a *transformation*.

Note that when you perform a transformation, the original nodes that you copied are *not* modified. For this reason, transformations are often limited only to situations where you want to modify a query result – for reporting purposes, for example.

To run a transformation, use the

1. `copy` keyword to copy the nodes of interest
2. `modify` keyword to perform the XQuery Update against the newly copied nodes
3. `return` keyword to return the result of the transformation.

For example, given the following XML document (which is identified as document `mydoc.xml`, and is stored in container `con.dbxml`):

```
<a>
  <aab1>first child</aab1>
  <b2>second child</b2>
  <b3>third child</b3>
</a>
```

then the following transformation:

```
copy $c := doc("dbxml:/con.dbxml/mydoc.xml")/a
  modify (delete nodes $c/aab1,
          replace value of node $c/b2 with "replacement value")
  return $c
```

results in the following document:

```
<a>
  <b2>replacement value</b2>
  <b3>third child</b3>
</a>
```

Resolving Conflicting Updates

Modifications that you specify as a part of an update query are not actually made until after the query is completed. The order in which update statements are made may or may not be relevant when it comes time to apply the update. As a result, it's possible to request an update that on its own is acceptable, but when used with other update statement may result in an error.

Keep the following rules in mind as you use update expressions:

1. An exception is raised if:
 - a. Two or more rename expressions target the same node.

- b. Two or more replace expressions or replace value of expressions target the same node.
2. The following expressions are made effective, in the following order:
 - a. All insert into, insert attributes and replace value expressions in the order they are supplied.
 - b. All insert before, insert after, insert as first, and insert as last expressions in the order they are supplied.
 - c. All replace expressions.
 - d. All replace value of expressions.
 - e. All delete expressions.

Note that atomicity of the expression is guaranteed; either the entire expression is made effective with regard to the original document, or no aspect of the expression is made effective.

Compressing XML Documents

By default all documents stored in a BDB XML whole document containers are compressed when they are stored in those containers, and uncompressed when they are retrieved from those containers. This requires a little bit of overhead on document storage and retrieval, but it also saves on disk space.

Note that only documents are compressed; metadata and indexes are not compressed.

You can cause compression to be turned off. You can also implement your own custom compression routine.

Note that whatever compression you use when you initially add documents to your container must be used for the lifetime of the container. You cannot, for example, turn compression off for some documents in the container and leave it on for others. You also cannot use more than one compression technique for the container.

Turning Compression Off

You turn compression off by setting `XmlContainerConfig::NO_COMPRESSION` for the `XmlContainerConfig::setCompressionName()` method. Note that you must do this on every container open or an `XmlException` is thrown when you attempt to retrieve a document from the container.

For example:

```
try {  
    // Set the container type as WholedocContainer and turn off  
    // compression
```

```

XmlContainerConfig contConf;
contConf.setAllowCreate(true);
contConf.setContainerType(XmlContainer::WholedocContainer);
contConf.setCompressionName(XmlContainerConfig::NO_COMPRESSION);

// Open container

// mgr is the XmlManager, opened at some point prior to this
// code fragment.
XmlContainer cont = mgr.openContainer("container.dbxml", contConf);

// From here you store and retrieve documents exactly in the same
// way as you always would.
} catch (XmlException &e) {
    // If you are turning off compression for a container that has
    // already stored compressed documents, BDB XML will not notice
    // until you try to retrieve a document that is compressed.
}

```

Using Custom Compression

You can implement custom compression routine for use with you BDB XML whole document containers. When you do this, you must register the compression routine when you create and open your container, and you must always use the same compression for all subsequent uses of the container.

You create a custom compression routine by providing an implementation of `XmlCompression`. You must implement methods that both compress and decompress your documents. Each of these methods must return true on success and false on failure.

The following is the class definition for a custom compression routine:

```

#include "dbxml/DbXml.hpp"
class MyCompression : public XmlCompression
{
    bool compress(XmlTransaction &txn,
                  const XmlData &source,
                  XmlData &dest);
    bool decompress(XmlTransaction &txn,
                    const XmlData &source,
                    XmlData &dest);
};

```

A true custom compression implementation is beyond the scope of this manual, but the following is an example implementation that uses inverse permutation to simulate compression. Notice that these member methods do not perform actual container activity; rather, they operate on the data found in the source `XmlData` parameter, and store the results in the destination `XmlData` parameter.

```

bool MyCompression::compress(XmlTransaction &txn,
                             const XmlData &source,

```

```

                                XmlData &dest)
{
    try {
        // Get the data to compress
        char *pSrc = (char *)source.get_data();
        size_t size = source.get_size();

        // Use inverse permutation to simulate the compression process
        dest.reserve(size);
        char *buf = (char *)dest.get_data();
        for(size_t i=0; i<size; i++)
            buf[i] = pSrc[size-1-i];
        dest.set_size(size);

    } catch (XmlException &xe) {
        cout << "XmlException: " << xe.what() << endl;
        return false;
    }
    return true;
}

bool MyCompression::decompress(XmlTransaction &txn,
                                const XmlData &source,
                                XmlData &dest)
{
    try {
        // Get the data to decompress
        char *pSrc = (char *)source.get_data();
        size_t size = source.get_size();

        // Use inverse permutation to simulate the decompression process
        dest.reserve(size);
        char *buf = (char *)dest.get_data();
        for(size_t i=0; i<size; i++)
            buf[i] = pSrc[size-1-i];
        dest.set_size(size);

    } catch (XmlException &xe) {
        cout << "XmlException: " << xe.what() << endl;
        return false;
    }
    return true;
}

```

To use this class implementation, you register your implementation with BDB XML, giving it a unique name as you do so. You then set that compression name to the container before opening it. All other container operations are performed as normal.

```

void useCompression(XmlManager& mgr,
                    const string& containerName,

```



```
        XmlUpdateContext& uc,  
        XmlCompression& myCompression)  
{  
    string docName = "doc1.xml";  
    string content = "<root><a></a></root>";  
  
    // Setup the document  
    XmlDocument xdoc1 = mgr.createDocument();  
    xdoc1.setName(docName);  
    xdoc1.setContent(content);  
  
    // Define an unique name to use for registering the compression  
    string compressionName = "myCompression";  
  
    // Register custom class  
    mgr.registerCompression(compressionName.c_str(), myCompression);  
  
    // Set the container type as WholedocContainer  
    // and use the custom compression  
    XmlContainerConfig contConf;  
    contConf.setAllowCreate(true);  
    contConf.setContainerType(XmlContainer::WholedocContainer);  
    contConf.setCompressionName(compressionName.c_str());  
  
    // Create container  
    XmlContainer cont = mgr.createContainer(containerName, contConf);  
  
    // Put Document  
    cont.putDocument(xdoc, uc);  
  
    // Get the Document  
    string content1;  
    cont.getDocument(docName).getContent(content1);  
}
```

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.

Note

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. You do this by providing the API with a string that identifies the type and syntax for the index. See [Syntax Types \(page 69\)](#) for information on specifying the index syntax.

Finally, by default BDB XML does automatically index your containers, regardless of whether you added indexes yourself. You can turn this feature off if it is in your way. See [Automatic Indexes \(page 73\)](#) for more information.

Index Types

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

- [Uniqueness \(page 66\)](#)
- [Path Types \(page 67\)](#)
- [Node Types \(page 68\)](#)
- [Key Types \(page 68\)](#)

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: <div style="text-align: center;">/vendor/name</div> and <div style="text-align: center;">salesrep/name</div> 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 phonenum elements because they appear in multiple (2) contexts within the document.

Node Types

BDX 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.
presence	Improves the performance of tests that look for the existence of an node, regardless of its value.

Key Type	Description
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.

See the next section for a complete list of syntax types available to you.

Specifying Index Strategies

The combined index type and syntax type is called the *index strategy*. To specify an index, you declare it using a string that specifies your index strategy. This string 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 \(page 66\)](#) for more information.

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

See [Path Types \(page 67\)](#) 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 \(page 68\)](#) 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 \(page 68\)](#) 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`. Also, for queries that examine numerical data without specifying the cast explicitly, use `double` instead of `decimal` for the index. This is because the XQuery specification requires implicit casts of numerical data to be performed as a `double`.

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"
```

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 occurring 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 of type `NodeContainer` use node-level indexes. Containers of type `WholedocContainer` use document level indexes by default. You can change the default indexing strategy for a container by setting `XmlContainerConfig::setIndexNodes()` to `XmlContainerConfig::On` (for node-level indexes) or to `XmlContainerConfig::Off` (for document-level indexes).

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`.

You can switch between node-level indices and document-level indices using the `XmlManager::reindexContainer()` method. Specify `XmlContainerConfig::On` to `XmlContainerConfig::setIndexNodes()` to cause the container to use node-level indices. Specify `XmlContainerConfig::Off` to cause it to use document-level indices. 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"
xmlns:myPrefix="http://dbxmlExamples/testPrefix />
```

is indexed as

```
<node1 http://dbxmlExamples/testPrefix:class="test"
xmlns: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.

Automatic Indexes

By default, BDB XML will automatically maintain the following indexes for your containers:

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

These automatic indexes will index all leaf nodes and all attributes. Maintaining them does represent a bit of a performance hit for your containers since BDB XML must determine if a document node is a leaf node, and then index it if it is. If a new leaf node is discovered when a document is added to your container (that is, BDB XML has never seen it before), then BDB XML will create the appropriate index for every document in your container.

For this reason, autoindexing is best used for containers that do not contain a mix of document types, and for containers that are not extremely large in size. If your containers do contain a wide variety of document types (and so new leaf nodes will be discovered frequently when adding documents), or your containers are simply extremely large, you should probably turn this feature off.

You turn automatic indexing off by specifying `false` to the `XmlIndexSpecification::setAutoIndexing()` method. Turn automatic indexing back on by specifying `true` to the same method.

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.

Note

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.

Note

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();

// Add the index. We're indexing "node1" using the default
// namespace.
is.addIndex("", "node1", "node-element-presence-none");

// 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();

// Delete the index. We're deleting the index from "node1" in
// the default namespace that has the syntax strategy identified
// above.
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- or comma-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 its 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();

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

```
// Add the default index.
is.addDefaultIndex("node-metadata-equality-string");

// 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 by default entire documents are always returned in this method's results set.
- If your container is of type `NodeContainer` then by default 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 a time potentially measured 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>
  <QueryPlanToAST>
    <NodePredicateFilterQP uri="" name="#tmp5">
      <StepQP axis="child" name="docId" nodeType="element">
        <StepQP axis="child" name="a" nodeType="element">
          <SequentialScanQP container="myContainer.dbxml"
            nodeType="document"/>
        </StepQP>
      </StepQP>
    <ValueFilterQP comparison="eq" general="true">
      <StepQP axis="attribute" name="id" nodeType="attribute">
        <VariableQP name="#tmp5"/>
      </StepQP>
    <Sequence>
      <AnyAtomicTypeConstructor value="bar"
        typeuri="http://www.w3.org/2001/XMLSchema" typename="string"/>
    </Sequence>
  </ValueFilterQP>
</NodePredicateFilterQP>
</QueryPlanToAST>
</XQuery>
```

While a complete description of the query plan is outside the scope of this manual, notice that there is no element specified in the query plan that includes an index attribute. This

attribute can appear on different element nodes, depending on the nature of the query and the actual index that the query wants to use. For example, queries that use indexes which examine the value of a node might specify a ValueQP node.

```
<ValueQP container="myContainer.dbxml"
index="node-attribute-equality-string" operation="eq" child="id"
value="bar"/>
```

Other indexes that simply test for the presence of a node would specify the index on a PresenceQP element:

```
<PresenceQP container="parts.dbxml"
index="node-element-presence-none" operation="eq"
child="parent-part"/>
```

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 dbxml shell is described in the *Introduction to Berkeley DB XML* guide.

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 (the following assumes the container already exists and contains documents):

```
> dbxml
dbxml> openContainer 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>
  <QueryPlanToAST>
    <NodePredicateFilterQP uri="" name="#tmp5">
      <StepQP axis="child" name="docId" nodeType="element">
        <StepQP axis="child" name="a" nodeType="element">
          <SequentialScanQP container="myContainer.dbxml"
            nodeType="document"/>
        </StepQP>
      </StepQP>
    </NodePredicateFilterQP>
    <ValueFilterQP comparison="eq" general="true">
      <StepQP axis="attribute" name="id" nodeType="attribute">
        <VariableQP name="#tmp5"/>
      </StepQP>
    </ValueFilterQP>
  </QueryPlanToAST>
</XQuery>
```

```

        <AnyAtomicTypeConstructor value="aaUivth"
            typeuri="http://www.w3.org/2001/XMLSchema"
            typename="string"/>
    </Sequence>
</ValueFilterQP>
</NodePredicateFilterQP>
</QueryPlanToAST>
</XQuery>

```

Notice that this query plan does not make use of an index. No index is identified anywhere in the query plan, and it calls for only a sequential scan. 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 there's a ValueQP element that specifies our newly added index using a index attribute.

```

dbxml> qplan collection("myContainer.dbxml")/a/docId[@id='aaUivth']
<XQuery>
  <QueryPlanToAST>
    <ParentOfAttributeJoinQP>
      <ValueQP container="myContainer.dbxml"
        index="node-attribute-equality-string" operation="eq"
        child="id" value="aaUivth"/>
      <StepQP axis="child" name="docId" nodeType="element">
        <StepQP axis="child" name="a" nodeType="element">
          <SequentialScanQP container="myContainer.dbxml"
            nodeType="document"/>
        </StepQP>
      </StepQP>
    </ParentOfAttributeJoinQP>
  </QueryPlanToAST>
</XQuery>

```

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

```

dbxml> quit

```

Chapter 8. Administering Berkeley DB XML Applications

This book has until now been an introduction on how to use the BDB XML API to add a native XML database to your application. But having written that application, there's some administrative concerns that you should keep in mind as your application moves into production. These concerns are described in this chapter.

Temporary Files

All Berkeley DB XML applications are capable of writing temporary files to disk. This happens when the disk cache fills up and so BDB XML is forced to write overflow pages. For the most part, these temporary files can be safely ignored.

However, for some class of applications, the presence of the temporary overflow files can be problematic. You can prevent temporary files from being created on your hard drive by creating your disk cache large enough that it can contain your entire working set of data. You do this using the `DB_ENV->set_cachesize()` method prior to opening your environment.

Note

It is always safe to delete temporary overflow files written by BDB XML after the application has shutdown.

Temporary database files are placed in the directory identified by the `DB_ENV->set_temp_dir()` method. If this method is not called by the application, then the application will use the directory identified on an environment variable, if your application is configured to do this. Assuming that it is appropriately configured, then the following environment variables are checked to see if they have been set. The following order of precedence matters; the first of the following environment variables found to be set is used to determine the location of the temporary directory:

1. `TMPDIR`
2. `TEMP`
3. `TMP`
4. `TempFolder`
5. `TMPDIR`

If none of these environment variables are set, BDB XML checks the value returned by the `GetTempPath` interface to see if that is set. If not, then the default location identified above are attempted.

Note

Environment variables are not used by BDB XML applications unless the `DB_USE_ENVIRON` or `DB_USE_ENVIRON_ROOT` flags are set when the environment is opened.

If no other method of determining the location of the temporary file directory can be found, then BDB XML will resort to using built-in default values. That is, the first of the following locations found to exist is used, if a temporary file directory is not otherwise identified for BDB XML:

1. The directory `/var/tmp`
2. The directory `/usr/tmp`
3. The directory `/temp`
4. The directory `/tmp`
5. The directory `C:/temp`
6. The directory `C:/tmp`

A Note on Snapshot Isolation

Snapshot isolation, or multi-version concurrency control, can be configured when you are using transactions with your BDB XML application. While transactions are not described in this manual, since snapshot isolation is so commonly used for BDB XML applications, it is worth mentioning here that use of Snapshot Isolation means you must:

- Increase the maximum number of concurrent transactions supported by the environment.
- Increase the size of your disk cache.

This is because snapshot isolation causes your BDB XML transactional application to use much larger amounts of resources than does a normal transactional application.

For more information, see the *Berkeley DB XML Getting Started with Transaction Processing* guide.