Getting Started with DBXML

C++ Edition

Steve Sarette

Makers of Berkeley DB

Getting Started with DBXML C++ Edition

Table of Contents

List of Examples

Chapter 1. Introduction to Berkeley DB XML

Welcome to Sleepycat's Berkeley DB XML (DBXML). DBXML is an embedded database specifically designed for the storage and retrieval of modestly sized XML-formatted documents. Built on the award-winning Berkeley DB, DBXML provides for efficient queries against millions of XML documents using XPath. XPath is a query language designed for the examination and retrieval of portions of XML documents.

This document introduces DBXML. It is intended to provide a rapid introduction to the DBXML API set and related concepts. The goal of this document is to provide you with an efficient mechanism with which you can evaluate DBXML against your project's technical requirements. As such, this document is primarily intended for CTOs, Software Architects, and Senior Software Engineers who are responsible for the design of any software that must manage large volumes of XML-formatted data.

Note that while this document uses C++ for its examples, the concepts described here should apply equally to all language bindings in which the DBXML API is available.

Finally, all of the examples used throughout this document are available as a series of compilable programs. You can find them in `<DBXML_HOME>/examples/cxx/gettingStarted`.

Features

DBXML is an embedded database that is tuned for managing and querying hundreds, thousands, or even millions of XML documents. The atom of storage for DBXML is the document, which is stored as a string in the underlying database. For this reason, DBXML works best with small documents (that is, documents under a megabyte in size), although it can be used to improve data access for large documents as well.

DBXML provides a series of features that makes it more suitable for storing XML documents than other common XML storage mechanisms. DBXML'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 DBXML 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 DBXML stores XML data in its native format, DBXML enjoys the same extensible schema that has attracted many developers to XML. It is this flexibility that makes DBXML a better choice than relational database offerings that must translate XML data into internal tables and rows, thus locking the data into the database's inflexible schema.

Languages and Platforms

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

DBXML is officially supported on Apple MacOS X, Linux, Solaris, and Microsoft Windows 2000 and XP. Due to its open source nature, it is possible that parties other than Sleepycat have ported the library to other platforms.

Check with the DBXML mailing lists for the latest news on supported platforms, as well as for information as to whether your preferred language provides DBXML support.

XML Features

DBXML is implemented to conform to the W3C standards for XML, XML Namespaces, and XPath 1.0. 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. All queries against XML data are performed using the container in which the data resides. Further, all indexes declared for the XML data is declared using the container.
- Indexes. DBXML indexes greatly enhance the performance of queries against the corresponding XML dataset. DBXML indexes are based on the structure of your XML documents, and as such you declare indexes based on the nodes that appear in your documents as well the data that appears on those nodes.
- Queries. DBXML queries are performed using the XPath language. XPath is a W3C specification (http://www.w3.org/TR/xpath) and it is designed to identify specific locations within an XML document, or to retrieve information located within an XML document.
- Query results. DBXML retrieves documents that match a given XPath query. DBXML query results are always returned as a set. The set can contain either matching documents, or a set of values from those matching documents. When documents are retrieved, they can be retrieved either as a string or a DOM tree.
- Storage. All XML documents are stored in DBXML using Unicode UTF-8. Documents are stored (and retrieved) in their native format with all whitespace preserved.
- Metadata attribute support. Each document stored in DBXML 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.

Database Features

Beyond XML-specific features, DBXML inherits a great many features from Berkeley DB, which allows DBXML to provide the same fast, reliable, and scalable database support as does Berkeley DB. The result is that DBXML is an ideal candidate for mission-critical applications that must manage XML data.

Important features that DBXML inherits from Berkeley DB are:

- In-process data access. DBXML 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 same IPC-overhead required by traditional SQL-based database implementations.

- Ability to manage databases up to 256 terabytes in size.
- Database environment support. DBXML 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 DBXML 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 DBXML'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. DBXML's transaction support ensures that all saved data is available no matter how the application or system might subsequently fail.
- Concurrent access. Through the combined use of isolation mechanisms built into DBXML, plus deadlock handling supplied by the application, multiple threads and processes can concurrently access the XML dataset in a safe manner.

Getting and Using DBXML

DBXML 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 DBXML library from the Sleepycat download page.

Documentation and Support

DBXML is officially described in the Sleepycat product documentation. For additional help and for late-breaking news on language and platform support, it is best to use the DBXML mailing lists. You can find out how to subscribe to these lists from the Berkeley DB XML product information page.

Library Dependencies

DBXML depends on several external libraries, some of which are currently in a beta state. The result is that build instructions for the DBXML library may change from release to release as it's 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 library distribution.

That said, DBXML currently relies on the following libraries:

- Berkeley DB. Berkeley DB provides the underlying database support for DBXML.
- Xerces. Xerces provides the DOM and SAX support that DBXML employs for XML data parsing.
- Pathan. Pathan provides the XPath support that you use to query your DBXML-stored documents.

Building and Running DBXML Applications

To build a DBXML application, you must make sure that the header files for the following libraries can be found by your compiler:

- Berkeley DB
- Berkeley DBXML
- Xerces

Depending on how you installed these libraries, you may or may not have to explicitly add this information to your compiler's include path. Specifically, make sure the following directories are placed in your compiler's include path:

```
-I$(DBXML)/include/dbxml -I$(DB)/include -I$(XERCES)/include
```

If you downloaded and are using the binary distribution of DBXML for win32, then you need only make sure that your project is configured to use the following directory:

```
dbxml-<version>-win32\include
```

Further, make sure you link the following libraries or DLLs into your application. Note that the actual library name differs from platform to platform, so check the `lib` directory in your library's installation directory for the name that is specific to your platform.

- Berkeley DBXML library.
- Berkeley DB C++ API.
- Xerces C library.
- Pathan library.

Again, depending on how you installed these libraries, you may have to explicitly identify their installation location to your compiler. The installation documentation that came with your distribution provides specific compilation instructions for each of the platforms officially supported by DBXML.

Chapter 2. Using Containers and Environments

A DBXML container is a database structure that contains one or more XML documents. You use containers to:

- Perform all XML documents storage activities (create, replace, and delete).
- Declare DBXML indexes.
- Perform document queries.

Just as is the case with Berkeley DB databases, DBXML containers can be stored in a Berkeley DB environment. Berkeley DB environments provide a great many features but of particular interest, they allow you to:

- Efficiently store, manage, and query one or more DBXML containers.
- Commingle DBXML containers and Berkeley DB databases for efficient application access to both.
- Protect data access with transactions.

Opening and Closing Containers

In order to use a DBXML container, you must open it. To do this, you use the `XmlContainer::open()` method. Note that the name that you provide for you container when you create it is also used as the container's filename on disk. To make it easier to locate these containers on disk, Sleepycat suggests that you end your containers with a `dbxml` suffix.

> **Note**
> You can open as many containers as your system's physical limitations (RAM and file descriptors) allow.

The following example creates, opens, and closes a container. The name of the container is `containerName.dbxml`. Because the provided name is not an absolute path, the container is created in the current working directory.

Example 2.1 Simple Open Container

```
#include "DbXml.hpp"
#include "db_cxx.h"

using namespace DbXml;

int main(void)
{

    //Instantiate the container. The container's name is "containerName.dbxml"

    XmlContainer myContainer(0,"containerName.dbxml");

    //Container flags. Set such that if the container does not exist,
    //it is created in the current working directory, and without benefit
    //of a database environment.
    u_int32_t cFlags=DB_CREATE;

    //Open the container.
    myContainer.open(0,cFlags);
```

To close the container, call its `close()` method.

```
    // do work here //

    //close the container and cleanup

    myContainer.close();
    return 0;
}
```

One way to organize this open and close activity is to wrap it in a class. Such a class can carry with it methods that identify the container's name, the Berkeley DB environment in which the container was opened (see below), and error and exception handling code. You can also use the class destructor to handle the container close. An example of a class like this exists in the DBXML examples directory. See `<DBXML>/examples/cxx/gettingStarted/myXmlContainer.hpp`

Using Containers with Berkeley DB Environments

In most cases you will want to open and use containers from within a Berkeley DB environment. Doing so offers you several useful features, including transaction protection as well as the ability to gracefully manage multiple containers and Berkeley DB databases.

Neither Berkeley DB XML nor Berkeley DB impose any limit on the number of containers that you can create in a DB environment. You are only limited by the physical constraints imposed by your filesystem.

To open a container within an environment, you first open the environment. It is required that you initialize the shared memory buffer pool when you do this. Otherwise, subsequent attempts to open any DBXML container in that environment will fail with an invalid argument exception.

Also, the directory containing your database environment must be identified to the `open()` method. The path that you give must currently exist. Further, the path must be provided as a null-terminated string (C++ strings are not supported for this method).

Example 2.2 Simple Open Database Environment

```
#include "DbXml.hpp"
#include "db_cxx.h"

using namespace DbXml;
//exception handling omitted for clarity

int main(void)
{

    std::string path2DbEnv = "/place/valid/path/here";
    //path2DbEnv holds the path to the environment.
    //It must contain a valid path to a currently existing
    //directory. In this example, all containers created in this
    //environment are created in the identified directory.


    u_int32_t cFlags=DB_CREATE|DB_INIT_MPOOL;
    //DB_CREATE causes the environment to be created if it does not exist.
    //DB_INIT_MPOOL initializes the shared memory buffer pool subsystem.

    DbEnv dbEnv(0);

    //Note that you must pass a null-terminated
    //string to this open().
    dbEnv.open(path2DbEnv.c_str(), cFlags, 0);
```

We can then open as many containers in the environment by passing the environment to each container's open method. Note that this example is a bit awkward because it does not encapsulate the environment and container management code in classes. That encapsulation is omitted here for clarity. See `myXmlContainer.hpp` and `myDbEnv.hpp` in the DBXML C++ examples directory for a more robust example of how to manage these objects.

```
    //multiple containers can be opened in the same
    //database environment
    XmlContainer container1(&dbEnv, "myContainer1.dbxml");

    container1.open( 0, DB_CREATE, 0 );

    XmlContainer container2(&dbEnv, "myContainer2.dbxml");
    container2.open( 0, DB_CREATE, 0 );

    XmlContainer container3(&dbEnv, "myContainer3.dbxml");
    container3.open( 0, DB_CREATE, 0 );

    // do work here //

    container1.close();
    container2.close();
    container3.close();

    dbEnv.close( 0 );

    return 0;
}
```

Database Open Flags

The container and database environment examples provided in this chapter used minimal flags on the `open()` calls. The container example used only:

```
DB_CREATE
```

while the database environment example used:

```
DB_CREATE|DB_INIT_MPOOL
```

These examples represent the minimum set of flags that you can use when creating containers and environments. However, real-world applications will typically use considerably more flags than these. The flags that you use for container and environment opens are what determine which database subsystems are available to your application.

For containers, a more common set of flags might be:

```
DB_CREATE|DB_AUTO_COMMIT
```

where `DB_AUTO_COMMIT` is a convenience that causes DBXML to automatically commit transactions when they are completed.

Berkeley DB environments have a large set of flags available to them. However, a common set is:

```
DB_CREATE|DB_INIT_LOCK|DB_INIT_LOG|DB_INIT_MPOOL|DB_INIT_TXN
```

where:

- `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.
- `DB_INIT_LOG` initializes the logging subsystem. This subsystem is used for database recovery from application or system failures.
- `DB_INIT_MPOOL` initializes the shared memory pool subsystem. This subsystem is required for DBXML container usage.
- `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 flag is specified, then `DB_INIT_LOG` must also be specified.

Regardless of the flags you decide to set at creation time, it is important to use the same flags on all subsequent container and environment opens (the exception to this is DB_CREATE which is only required to create a container or environment). In particular, avoid using flags to open containers or environments that were not specified at creation time. This is because different subsystems require different data structures on disk. Therefore, attempts to use subsystems that were not initialized at database creation time can have problematic results.

See the formal DBXML and Berkeley DB documentation for details on the flags that are available to you. Specifically, the API documentation for `XmlContainer->open()` describes the flags available for use when you open containers. The `DBENV->open()` API documentation describes the flags that you can use when opening an environment.

Chapter 3. Adding Documents to DBXML

XML documents are stored in DBXML containers. A prerequisite to storing a document in a container is that the document first be contained in a single string. Otherwise, the storage process itself is simple:

1. Open the container in which you want to store the document.
2. Set the content of an `XmlDocument` instance to the string that contains the document you want to store.
3. Call your container's `putDocument()` method, passing in the `XmlDocument` instance as an argument.

Note that the document is stored in exactly the same format as it is contained in the string. That is, line breaks and whitespace are preserved.

The following example adds two very simple XML documents to a DBXML container.

Example 3.1 Adding Documents to a Container

```
#include "DbXml.hpp"
#include "db_cxx.h"

using namespace DbXml;
//exception handling omitted for clarity

int main(void)
{

    std::string document1 = "<aDoc><title>doc1</title><color>green</color></aDoc>";
    std::string document2 = "<aDoc><title>doc2</title><color>yellow</color></aDoc>";

    //Open a db environment
    const std::string path2DbEnv = "/path/to/my/database/environment";
    dbEnv.open(path2DbEnv.c_str(), DB_CREATE|DB_INIT_MPOOL, 0);

    //Open a container in the db environment
    XmlContainer container(&dbEnv, "myContainer.dbxml");
    container.open( 0, DB_CREATE, 0 );

    //Add the documents
    XmlDocument myXMLDoc;

    //Set the XmlDocument to the relevant string and then put it
    // into the container.
    myXMLDoc.setContent( document1 );
    container.putDocument(0,myXMLDoc);

    //Do it again for the second document
    myXMLDoc.setContent( document2 );
    container.putDocument(0,myXMLDoc);


    //Close the container
    container.close();

    //Close the environment
    dbEnv.close(0);

    return 0;
}
```

For an example of reading XML data from disk and then adding that data to a container, see `exampleLoadContainer.cpp` in the DBXML C++ examples directory.

Chapter 4. Using XPath with DBXML

Documents are retrieved from DBXML containers using XPath expressions. XPath is a language designed to identify locations and data in an XML document by using a combination of Unix-style path notation and simple programming language expressions. XPath is heavily used with the Extensible Stylesheet Language for Transformations (XSLT) language, which is a common mechanism for manipulating and transforming XML documents, as well as with XPointer which is a language used to point to specific locations in an XML document.

XPath is formally described in a W3C specification (http://www.w3.org/TR/xpath). Moreover, any good book on XSLT or XPointer should provide a thorough description of XPath.

This chapter begins with a brief introduction to XPath. This introduction is not meant to be a complete description of the language. Instead, the chapter focuses on XPath as it might be used to retrieve documents and document data from a DBXML container.

If you are already familiar with XPath, then you can skip to Retrieving DBXML Documents using XPath for a description of the DBXML APIs used to perform XPath queries.

DBXML uses the Pathan library for its XPath support.

XPath: A Brief Introduction

XPath views an XML document as a collection of element, text, and attribute nodes. Element nodes are identified by the documents tags. XPath uses Unix-style path notation to identify a specific element node in a document. For example, consider the following XML document:

Example 4.1 A Simple XML Document

```
<?xml version="1.0"?>
<Node0>
    <Node1 class="myValue1>

        Node1 text
    </Node1>
    <Node2>
        <Node3>
            Node3 text
        </Node3>
        <Node4>
            300
        </Node4>
    </Node2>
</Node0>
```

Given this document, the XPath expression to reference each of the document's element nodes are:

Node	XPath Expression
<Node0>	/Node0
<Node1>	/Node0/Node1
<Node2>	/Node0/Node2
<Node3>	/Node0/Node2/Node3

Note that the first node is a document is technically not considered to be an element node. The XPath literature will usually refer to this node as the root node . You can refer to the root node using either of the following notations:

```
/Node0
```

or

```
/
```

Finally, XPath provides expression predicates with which you can filter the nodes selected by an XPath expression. Predicates are contained in an XPath expression using square brackets ([]) and they always evaluate to an boolean value. They are described in greater detail in the section on predicates later in this chapter.

Selecting Text Nodes

Any text contained in an element node is processed as an text node. You select a text node using the `text()` test. For example, `Node3` in A Simple XML Document contains a text node who's contents are:

```
Node3 text
```

To retrieve just this text from DBXML, use:

```
/Node0/Node2/Node3/text()
```

If you are testing the value of a text node against some other value, you can use any of the four following forms:

```
/Node0/Node2/Node3/text()="foo"
```

or

```
/Node0/Node2/Node3="foo"
```

or (as a predicate)

```
/Node0/Node2[Node3="foo"]
```

or

```
/Node0/Node2/Node3[text()="foo"]
```

All of these forms work without complaint, but for clarity this documentation and supporting examples usually use the last form shown here.

Selecting Attribute Nodes

Any attributes found on an element node are processed as attribute nodes. To select an attribute node, use an at-sign (@) with the attribute name. For example, `Node1` in A Simple XML

Document contains the `class` attribute. To select this attribute, use the following expression:

```
/Node0/Node1/@class
```

Context

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

Relative Paths

Just like Unix filesystem paths, any path that does not begin with a slash (`/`) is relative to your current location in a document. Your current location in a document is determined by your context. Thus, if in A Simple XML Document 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:

```
.
```

Note
Remember that DBXML supports relative paths only from within predicates.

Namespaces

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

Consider the following document:

Example 4.2 XML Documents and Namespaces

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

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

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

Example 4.3 Namespace Declaration

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

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

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

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

Example 4.4 Namespace Prefixes

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

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

Example 4.5 Namespaces with Attributes

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

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

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

Note that unlike element nodes, attribute nodes do not terminate with a text() node. So to test the value of a attribute node, use:

```
/clubMembers/surveyResults/@school:class="foo"
```

Predicates

Predicates are expressions that evaluate to a boolean result based on some feature of the targeted document. Predicate expressions are always contained within square brackets ([]). For example:

```
/clubMembers/surveyResults[@school:class="Middle"]
```

matches all the nodes where the `school:class` attribute is set to `"Middle"`

Examples of some other XPath predicate expressions are:

- Selects all the Node1 nodes who's text node is equal to `"An Example Node"`:

  ```
  /Node0/Node1[text()="An Example Node"]
  ```

- Selects all the Node4 nodes who's text node contains a number that is less than 200. Note that in XPath, all numbers are actually evaluated as floats (there are no integers):

  ```
  /Node0/Node2/[number(Node4)<200]
  ```

- Selects the document who's Node4 node is greater than 200 and who's text node is equal to 'test1':

```
/Node0[Node2/number(text())>=200 and Node1/text()="test1"]
```

- Selects the last Node6 node that is a child of Node5:

```
/Node0/Node5/Node6[last()]
```

- Selects the first Node6 node that is a child of Node5.

```
/Node0/Node5/Node6[position()=1]
```

These are just a few examples of the sorts of tests that you can perform using XPath. XPath comes with a fairly large library of functions and tests, which should be fully described in any good book on XSLT or XPointer.

Wildcards

Use wildcards when document elements are unknown. For example:

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

selects all the Node6 nodes that are 3 nodes deep in the document. 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 beneath Node5:

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

- Selects all of Node5's attributes:

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

Retrieving DBXML Documents using XPath

Documents are retrieved from DBXML when they match an XPath query. To retrieve documents, you issue XPath queries against DBXML containers. To create these queries, you first define whatever query context might be required, and you then issue the XPath expression against the container. All documents that match the query are returned in the form of a result set. You then loop through this result set, processing each document in the set as is required by your application.

DBXML query contexts are responsible for:

- Defining the result type. That is, the type of information returned in the result type (see below).

- Defining the namespaces to be used in the query.
- Defining any variables that might be needed for the query.
- Defining whether the query is processed “eagerly” or “lazily”. If lazy processing is selected, then the final evaluation of the query is deferred until your code is actually stepping through the result set.

The result set can be set to return different types of information. For example, two common result types are:

Result Type	Description
`ResultDocuments`	Returns entire documents that match your XPath query. This is the default result type.
`ResultValues`	Returns the actual value or document fragment that matched the query. For example, if you query for a node, then the document fragment(s) that match that query are returned. If you query for a text node, then the text in that node is returned. If you query for an attribute node, then the values for all matching attributes are returned. This result type is frequently used to retrieve document data from individual documents returned in a document set. See Retrieving Document Data using XPath for more information.

You use an `XmlValue` object to retrieve the individual elements in the result set. The element contained in the `XmlValue` object can be any one of a number of types supported by DBXML (for example, a boolean, a number, a string, and so forth). You can retrieve the value stored in the `XmlValue` object as one of these types, provided that the object contains an element of the appropriate type.

Note that `XmlValue` also provides methods to test the type of value contained by the `XmlValue` object. See the Berkeley DBXML C++ API Reference for details on the `XmlValue` class methods available to you.

In addition to retrieving elements as a simple type, you can also retrieve elements as an `XmlDocument` object. `XmlDocument` is the unit of storage within DBXML, and as such you can use it to perform operations such as setting and retrieving document content, setting and retrieving metadata, and applying XPath queries to the individual document. All of these activities are described in chapters later in this document.

`XmlDocument` can be retrieved directly from the results set, or it can be retrieved from an `XmlValue` object.

> **Note**
> For the C++ DBXML API only, you can also use `XmlDocument` to obtain a Xerces DOM nodelist. You can manage and manipulate this nodelist in the same way as you would any nodelist within your application.

Examples Document Set

The remaining sections in this chapter contain examples that assume a specific document set. All of the data required to exercise these examples is available in your DBXML c++ examples directory. Use the `loadExampleData.[sh|cmd]` script that is also available in your examples directory to load this data into the appropriate DBXML containers.

Note that before you can run the `loadExampleData.[sh|cmd]` script, you must first compile `exampleLoadContainer.cpp`.

Performing Requests

To issue an XPath query against a container, use the `XmlContainer::queryWithXPath()` method. Use the results of this query to construct an `XmlResults` object. You then construct an `XmlValue` object and use it to iterate through the `XmlResults` set so as to process each document in the set.

The following example provides a very simple function that performs an XPath query against a container. Because no context is declared in this function, the `XmlQueryContext::ResultDocument` result type is used by default. Each item in the result set is retrieved as a string and then printed to the console.

Example 4.6 Simple XPath Query

```
void doQuery( XmlContainer &container, const std::string &XPath )
{
    //perform the query.
    XmlResults results( container.queryWithXPath(0, XPath, 0 ) );

    //evaluate the results of the query.
    XmlValue value;
    while( results.next(0,value) )
    {
        /// Obtain the value as a string and print it to stdout
            std::cout << value.asString() << std::endl;
    }
}
```

You can then perform your queries:

```
int main(void)
{

    //Open a db environment
    const std::string path2DbEnv = "/path/to/my/database/environment";
    dbEnv.open(path2DbEnv.c_str(), DB_CREATE|DB_INIT_MPOOL, 0);

    //Open a container in the db environment
    XmlContainer container(&dbEnv, "simpleExampleData.dbxml");
    container.open( 0, DB_CREATE, 0 );

    //find all the products that are vegetables
    doQuery( openedContainer, "/product/category[text()='vegetables']");

    //find all the products where the price is less than or equal to 0.11
    doQuery( openedContainer, "/product/inventory[number(price)<=0.11]");

    //find all the vegetables where the price is less than or equal to 0.11
    doQuery( openedContainer,
    "/product[number(inventory/price)<=0.11 and category/text()=\"vegetables\"]");

    openedContainer.close();
    dbEnv.close(0);
    return 0;
}
```

Setting a Context

The only difference between performing a query with and without a context is that if you are using a context, then you must pass it to `queryWithXPath()`:

Example 4.7 XPath Query with a Context

```
void doContextQuery( XmlContainer &container, const std::string &XPath,
                     XmlContext &context )
{
    //perform the query using the context
    XmlResults results( container.queryWithXPath(0, XPath, &context ) );

    //evaluate the results of the query.
    XmlValue value;
    while( results.next(0,value) )
    {
        //Obtain the value as a string and print it to stdout
            std::cout << value.asString() << std::endl;
    }
}
```

To define the context, you use the appropriate `set` method on an `XmlContext` object.

- To declare a namespace, use `setNamespace()`.
- To declare a variable, use `setVariableValue()`.

```
int main(void)
{
    //Open a db environment
    const std::string path2DbEnv = "/path/to/my/database/environment";
    dbEnv.open(path2DbEnv.c_str(), DB_CREATE|DB_INIT_MPOOL, 0);

    //Open a container in the db environment
    XmlContainer container(&dbEnv, "simpleExampleData.dbxml");
    container.open( 0, DB_CREATE, 0 );

    //declare a context
    XmlContext context;

    //set namespaces
    context.setNamespace( "fruits", "http://groceryItem.dbxml/fruits");
    context.setNamespace( "vegetables", "http://groceryItem.dbxml/vegetables");
    context.setNamespace( "desserts", "http://groceryItem.dbxml/desserts");

    //set a variable
    context.setVariableValue( "aDessert", "Blueberry Boy Bait");
```

You can then perform your queries:

```
    //returns no documents because a namespace prefix is not provided
    doContextQuery( openedContainer, "/item", context);

    //returns all the documents that describe fruits
    doContextQuery( openedContainer, "/fruits:item", context);

    //returns the document that describes "Blueberry Boy Bait" (a dessert)
    doContextQuery( openedContainer,
    "/desserts:item/product[text()=$aDessert]", context);

    openedContainer.close();
    dbEnv.close(0);
    return 0;
}
```

Retrieving Document Data using XPath

You can retrieve document nodes by setting the `XmlQueryContext::returnType` to `XmlQueryContext::ResultValues`. When you do this, the node(s) that match the provided XPath expression are returned. As always, the results are returned in a set.

It is possible to use `XmlQueryContext::ResultValues` on a DBXML container query. However, because you are returning just a portion of a larger document, in some circumstances you may find the results confusing because you lose the context as to which document the value was retrieved from. A better solution might be to perform an initial query that returns the documents in which you are interested, and then perform secondary queries against each individual document in the result set.

In the following example, we create a function that returns the result value of an XPath query. Note that for our sample application (and the corresponding example document set), we can safely assume that the results of our query produces a set of size 1. If this was not the case, then it would be necessary to make allowances for multiple elements in the result set. Depending on your application, this may mean looping through the result set and, for example, placing each value in a vector. The function would then return the vector instead of the string that we use here.

We also make no allowances here for a result set of size 0. In production code, you should manage all such boundary conditions.

Example 4.8 Obtaining Result Values

```
std::string getValue( XmlDocument &document,
                      const std::string &XPath,
                      XmlQueryContext &context )
{
    // Exception handling omitted....

    //We don't want a document, we want a specific value.
    //So set the return type to Result Values
    context.setReturnType( XmlQueryContext::ResultValues );

    //Perform the query against the document
    XmlResults result = document.queryWithXPath(XPath, &context);

    //We require a result size of exactly 1.
    assert(result.size() = 1 );

    //Get the value. If we allowed the result set to be larger than size 1,
    //we would have to loop through the results, processing each as is
    //required by our application.
    XmlValue value;
    result.next(0,value);

    //Set the result type back to Result Document
    context.setReturnType( XmlQueryContext::ResultDocuments );

    //Return the value as a string.
    return value.asString();
}
```

Given this function, we can call it from inside a normal results processing loop. To simplify things, we just hard-code here the XPath expressions that we pass to `getValue()`.

```
void getDetails( XmlContainer &container, const std::string &XPath,
    XmlContext &context )
{
    //perform the query using the context.
    XmlResults results( container.queryWithXPath(0, XPath, &context ) );

    //evaluate the results of the query.
    XmlDocument theDocument;
    while( results.next(0, theDocument) )
    {
         /// Obtain information of interest from the document. Note that the
         //  wildcard in the XPath expression allows us to not worry about
         //  what namespace this document uses.
         std::string item = getValue( theDocument,
                              "/*/product/text()", context);
         std::string price = getValue( theDocument,
                               "/*/inventory/price/text()", context);
         std::string inventory = getValue( theDocument,
                                  "/*/inventory/inventory/text()", context);

         std::cout << "\t" << item << " : "
                   << price << " : "
                   << inventory << std::endl;
    }
}
```

Finally, we just call `getDetails()` to retrieve the document set for which we want details reported.

```
int main(void)
{


    //Open a db environment
    const std::string path2DbEnv = "/path/to/my/database/environment";
    dbEnv.open(path2DbEnv.c_str(), DB_CREATE|DB_INIT_MPOOL, 0);

    //Open a container in the db environment
    XmlContainer container(&dbEnv, "simpleExampleData.dbxml");
    container.open( 0, DB_CREATE, 0 );

    //create a context and declare the namespaces
    XmlQueryContext context;
    context.setNamespace( "fruits", "http://groceryItem.dbxml/fruits");
    context.setNamespace( "vegetables", "http://groceryItem.dbxml/vegetables");
    context.setNamespace( "desserts", "http://groceryItem.dbxml/desserts");

    //get details on Zulu Nuts
    getDetails( openedContainer,
                "/fruits:item/product[text() = 'Zulu Nut']", context);

    //get details on all fruits that start with 'A'
    getDetails( openedContainer,
                "/vegetables:item/product[starts-with(text(),'A')]", context);

    openedContainer.close();
    dbEnv.close(0);
}
```

Chapter 5. Deleting and Replacing Documents in DBXML

To delete a document from a container, you:

1. Query for the documents you want to delete.
2. Iterate through the result set, retrieving each document in turn as an `XmlDocument`.
3. Call `XmlContainer::deleteDocument()`, passing it the `XmlDocument` you retrieved in the previous step.

To replace a document in DBXML (that is, overwrite its content in the database), you:

- Delete the document as described above.
- Add the updated document back to the container as described in Adding Documents to DBXML.

DBXML provides `XmlContainer::updateDocument()` as a convenience method to perform this operation.

> **Note**
> To make the replace operation safer, you may want to perform it in a transaction.

Deleting Documents from DBXML Containers

Deleting a document is simply a matter of creating a result set, and then calling `XmlDocument::deleteDocument()` on each item in the set. For example:

Example 5.1 Deleting a document from DBXML

```
void doDocumentDelete( XmlContainer &container,
     const std::string &XPath, XmlContext &context )
{
    //Get the set of documents that we want to delete.
    XmlResults results( container.queryWithXPath(0, XPath, &context ) );

    //iterate through the results, deleting each document in turn.
    XmlDocument theDocument;
    while ( results.next(0, theDocument) )
    {
            container.deleteDocument(0,theDocument);
    }
}
```

Replacing Documents in DBXML Containers

To replace or update a document that is stored in DBXML, you essentially delete the old document and then save the updated version to DBXML. Note that if you do not delete the old document first, then you will have multiple versions of the same document. The only difference between the two being whatever modifications you made to the second document.

You can perform this delete and add operation using the `XmlContainer::updateDocument()` method. To do so:

1. Query for the document you want to modify. This creates a result set that you can iterate over. Make sure that the context you use for the query is set to `ResultDocuments` or `CandidateDocuments`.
2. For each element in the result set, retrieve the element as both an `XmlValue` and an `XmlDocument` object.
3. From either the `XmlValue` or the `XmlDocument` object, retrieve the document in whatever format you want to use to manipulate it. For example, you could retrieve the document as a Xerces DOM nodelist and modify it in the same way as you would any DOM. Or you can retrieve it as a string (availableo from `XmlValue` only, and manipulate the string.
4. Place your modified document into a string. If you are manipulating documents as a DOM nodelist, you must serialize the DOM into a string before you can continue.
5. Use `XmlDocument::setContent()` to set the string representing your modified document to be the document's content.
6. Call `XmlContainer::updateDocument()`, passing to it the now updated `XmlDocument`.

For example:

Example 5.2 Replacing a Document with updateDocument()

```
void doUpdateDocument( XmlContainer &container, const std::string &XPath,
                       XmlQueryContext &context)
{
    //Get the document(s) that we want to update
    XmlResults results( container.queryWithXPath( 0, XPath, &context ) );

    //Iterate through the result set as is normal
    XmlValue value;
    XmlDocument theDocument;
    while( results.next(0, theDocument, value) )
    {
        //We want to modify the document as a string, so get the string.
        std::string docString = value.asString();

        //This next function is one we wrote. It just modifies the document
        //string in a small way.
        std::string newDocString = modifyDocument( docString );

        //Set the document's content to be the new document string.
        theDocument.setContent( newDocString );

        //Now replace the document in the container
        container.updateDocument( 0,theDocument );
    }
}
```

Chapter 6. Document Names and Metadata

It is possible to associate information with a document that does not fit into the document's schema. A common case is to associate a name with the document, and so DBXML provides a special mechanism by which you can do exactly that. In addition, you can associate virtually any other kind of information with the document by using DBXML's metadata mechanism.

You can query for documents based on document names and/or metadata. Both types of information are reflected onto the document as if they were attributes on the document's root node.

Note that documents in DBXML are also associated with a special document ID. This ID is a unique integer used by the underlying Berkeley DB to uniquely identify the document within the database. It is possible for you to obtain this ID from a DBXML document and to perform operations using it. However, document IDs are not guaranteed to be constant over the lifetime of a document and container. Therefore, you should avoid relying on them and instead base your DBXML operations on XPath queries that uniquely identify your documents. Document names and metadata are an excellent way for you to associate a unique ID (generated by your application) with your documents if your XML schema does not provide an element that can serve this purpose.

Using Document Names

You associate a name with a document using `XmlDocument::setName()`:

Example 6.1 Setting Document Names

```
void addDocument( XmlContainer &container, const std::string
                  &docString, const std::string &documentName )
{
    XmlDocument myXmlDoc;
    myXMLDoc.setContent( docString );
    myXmlDoc.setName( documentName);
    container.putDocument(0,myXMLDoc);
}
```

Similarly, you retrieve a document's name using `XmlDocument::getName()`.

Example 6.2 Retrieving Document Names

```
void doQuery( XmlContainer &container, XmlQueryContext &context,
              const std::string &XPath )
{
    XmlResults results( container.queryWithXPath( 0, XPath, &context ) );

    //Iterate through the result set as is normal
    XmlDocument theDocument;
    while( results.next(0, theDocument) )
    {
        std::cout << "Found document named: "
                  << theDocument.getName() << std::endl;
    }

}
```

Note

If you want to change a document's name, follow the procedure for updating a document as described in Replacing Documents in DBXML Containers. The only difference is that you set the document's name before you pass the `XmlDocument` to `XmlContainer::updateDocument()`.

Finally, document names are reflected onto the document's root node as a `dbxml:name` attribute, so you can query for a document using its name like this:

Example 6.3 DBXML Queries using Document Names

```
doQuery( container, context, "/*[@dbxml:name='myDocumentName']" );
```

Note

The `dbxml` prefix is predefined. You are not required to declare it with `XmlQueryContext::setNamespace`.

Using Metadata

You can use metadata to associate arbitrary information with a document. Metadata should be used for information that does not fit with your document's schema. For example, you could use metadata to describe a document's creation or last modified time stamps, or the name of the person who last accessed the document.

Metadata is reflected onto the document in the form of an attribute on the root node. You set this information by providing:

- A namespace prefix and URI for the attribute.
- An attribute name.
- An attribute value.

You provide this information using `XmlDocument::setMetaData()`:

Example 6.4 Setting Document Metadata

```
void addDocument( XmlContainer &container, const std::string
                  &docString, const std::string &metaDataValue )
{
    std::string metaDataURI = "http://dbxmlExamples/metadata";
    std::string metaDataPrefix = "metaDataPrefix";
    std::string metaDataName = "metaDataAttributeName";

    XmlDocument myXmlDoc;
    myXMLDoc.setContent( docString );
    myXmlDoc.setMetaData( metaDataURI, metaDataPrefix,
                          metaDataName, metaDataValue);
    container.putDocument(0,myXMLDoc);
}
```

You can retrieve the value for a metadata attribute using `XmlDocument::getMetaData()`. To do this, you must specify the URI and attribute name that you used when you set the metadata:

Example 6.5 Retrieving Metadata Information

```
void doQuery( XmlContainer &container, XmlQueryContext &context,
              const std::string &XPath )
{
    std::string metaDataURI = "http://dbxmlExamples/metadata";
    std::string metaDataName = "metaDataAttributeName";

    XmlResults results( container.queryWithXPath( 0, XPath, context ) );

    //Iterate through the result set as is normal
    XmlDocument theDocument;
    while( results.next(0, theDocument) )
    {
        std::string mdValue = theDocument.getMetaData( metaDataURI,
                                                       metaDataName );

        std::cout << "Found metadata URI: " << metaDataURI
                  << ", attribute name: " << metaDataName
                  << ", value: " << mdValue << std::endl;
    }

}
```

You can also query for a document based on a metadata attribute value. Note that you must first define the namespace in the query context:

Example 6.6 DBXML Queries using Document Names

```
XmlQueryContext context;
context.setNamespace("metaDataPrefix",
                     "http://dbxmlExamples/metadata" );
doQuery( container, context,
     "/*[@metaDataPrefix:metaDataAttributeName = 'some value']" );
```

Chapter 7. Using DBXML Indexes

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

To make the most effective usage of DBXML indexes, design your indexes for your most frequently occurring XPath queries. Be aware that DBXML indexes can be updated or deleted in-place. This means that you do not have to decide on an indexing strategy upfront – you can take the time necessary to decide what your application's query requirements are before deciding on an indexing strategy. And, if you find over time that your application's queries have changed, then you can always modify your indexes to meet your application's shifting requirements.

> **Note**
> The time it takes to re-index a container is proportional to the container's size.

When you define an index in DBXML, you must identify the node for which you want the index created, and the type of index you want to use.

Index Types

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

- Path Types
- Node Types
- Key Types
- Syntax Types

Index types are declared as a string that uses the following format:

```
pathtype-nodetype-keytype-syntaxtype
```

For example, a legal index type is:

```
node-element-substring-string
```

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 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 DBXML query processor prefers `edge` -type indexes because they are more specific than an `element` -type index. This means that the query processor will use a edge-type index over a node-type if both indexes provide similar information.

Consider the following document:

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

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

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

`edge`	There are two edge nodes in the document that involve the `name` node:
	`/vendor/name`
	and
	`salesrep/name`
	Indexes 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 indexes with the `address`, `city`, `state`, `zipcode`, and `salesrep` elements because they appear in only one context within the document.
- `edge` path types to improve query performance when a node name is used in multiple contexts within the document. In the preceding document, use edge path types for the `name` and `phonenumber` elements because they appear in multiple (2) contexts within the document.

Node Types

DBXML can index two types of nodes: `element` or `attribute`. 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 walk XPath paths or that test the value of an element node. Use the attribute node type to improve any query that examines an attribute or attribute value.

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.
`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 XPath `contains()` substring function.

Syntax Types

Examines how indexed values are compared. You can use one of three syntax types:

Syntax Type	Description
`string`	Performs evaluations as a string compare. Use this syntax type with `equality` and `substring` key types.
`number`	Performs evaluations as a number compare. Use this syntax type with `equality` key types when you are evaluating node values as a number.
`none`	Indicates that the index will not be used to perform comparisons. Use this syntax type with the `presence` key type.

Legal Index Types

Only a subset of the possible permutations of the index types are supported by DBXML. Remember that to declare an index, you identify both the index type and the node that you want indexed. The following list identifies what effect the index type has on the node for which it is used.

- `none-none-none-none`

  Turns off indexing entirely for the node. Any other indexes that may apply to the node are ignored.

- `node-element-presence`

  Defines a presence index for an element node. Improves performance for queries that retrieve data based on the presence of a specific element.

- `node-attribute-presence`

  Defines a presence index for an attribute node. Improves performance for queries that retrieve data based on the presence of a specific attribute.

- `node-element-equality-string`

  Defines a string equality index for an element node. Improves performance for queries that retrieve data based on the (string) value to which the element node's text() node is set.

- `node-element-equality-number`

  Defines a number equality index for an element node. Improves performance for queries that retrieve data based on the (number) value to which the element node's text() node is set.

- `node-element-substring-string`

  Defines a substring index for an element node. Improves performance for queries that retrieve data based on substring characteristics of the element node's text() node.

- `node-attribute-equality-string`

  Defines a string equality index for an attribute node. Improves performance for queries that retrieve data based on the attribute's value.

- `node-attribute-equality-number`

  Defines a number equality index for an attribute node. Improves performance for queries that retrieve data based on the attribute's value, when that value is a number.

- `node-attribute-substring-string`

  Defines a substring index for an attribute node. Improves performance for queries that retrieve data based on substring characteristics of the attribute's value.

- `edge-element-presence`

  Defines a presence index for an element's edge. Improves performance for queries that routinely walk a specific element path in your documents.

- `edge-attribute-presence`

  Defines a presence index for an attribute's edge. Improves performance for queries that routinely walk a specific element/@attribute path.

- `edge-element-equality-string`

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

- `edge-element-substring-string`

  Defines an substring index for an element's edge. Improves performance for queries that examine whether a specific path contains an identified substring.

- `edge-attribute-equality-string`

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

- `edge-attribute-equality-number`

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

- `edge-attribute-substring-string`

  Defines an substring index for an attribute's edge. Use this index type to improve performance for queries that examine whether a specific element/@attribute path contains, starts with, or ends with an identified substring.

Indexer Processing Notes

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

- As with all indexing mechanisms, the more indexes that you maintain the slower your write performance will be. Substring indexes 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 indexes as element and attribute values (as indexed) may differ from what you expect.
- The indexer substitutes internal entity references with their replacement text.
- The indexer concatenates character data mixed with child data into a single value. For example, as indexed the fragment:

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

has two elements. ‹node1› has the value:

"This is some text with some data"

while ‹inline› has the value:

“inline”

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

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

is indexed as if ‹node1› has the value:

“Reserved XML characters are ‘<’, ‘>’, and ‘&’”

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

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

is indexed as

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

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

Managing DBXML Indexes

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

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, DBXML 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.

Adding Indexes

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. If the index already exists for the node that you specify on this method, then the method silently does nothing.
3. Set the updated index specification back to the container.

For example:

Example 7.1 Adding an Index to a Container

```
void addIndex( XmlContainer &container, const std::string &URI,
               const std::string &nodeName, const std::string
               &indexType )
{
     //retrieve the XmlIndexSpecification from the container
     XmlIndexSpecification idxSpec=container.getIndexSpecification(0);

     //Add the index to the specification.
     //If it already exists, then this does nothing.
     idxSpec.addIndex( URI, nodeName, indexType );

     //Set the specification back to the container
     container.setIndexSpecification( 0, idxSpec );
}
```

You can then add an index like this:

```
//add an string equality index for the "product" element node.
addIndex( container, "", "product", "node-element-equality-string" );

//add an edge presence index for the product node
addIndex( container, "", "product", "edge-element-presence" );
```

Deleting Indexes

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:

Example 7.2 Deleting an Index from a Container

```
void deleteIndex( XmlContainer &container, const std::string &URI,
                  const std::string &nodeName, const std::string &indexType )
{
     //retrieve the XmlIndexSpecification from the container
     XmlIndexSpecification idxSpec=container.getIndexSpecification(0);

     //Add the index to the specification.
     //If it already exists, then this does nothing.
     idxSpec.deleteIndex( URI, nodeName, indexType );

     //Set the specification back to the container
     container.setIndexSpecification( 0, idxSpec );
}
```

You can then delete an index like this:

```
//delete the string equality index for the "product" element node.
deleteIndex( container, "", "product", "node-element-equality-string" );

//delete the an edge presence index for the "product" node
deleteIndex( container, "", "product", "edge-element-presence" );
```

Replacing Indexes

You can replace the indexes maintained for a specific node by using: `XmlIndexSpecification::replaceIndex()`

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

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

Replacing the node's index, then, sets this string to whatever index types you include on the string.

For example:

Example 7.3 Replacing a Node's Index

```
void replaceIndexes( XmlContainer &container, const std::string &URI,
                     const std::string &nodeName, const std::string &indexType)
{
     XmlIndexSpecification idxSpec=container.getIndexSpecification(0);

     //Replace the indexes for the specified node
     idxSpec.replaceIndex( URI, nodeName, indexType );

     //Set the specification back to the container
     container.setIndexSpecification( 0, idxSpec );
}
```

You can then replace an index like this:

```
replaceIndex( container, "", "product",
              "node-attribute-substring-string node-element-equality-string" );
```

Examining Container Indexes

You can iterate over all the indexes in a container using `XmlIndexSpecification::next()`

For example:

Example 7.4 Counting the Indexes in a Container

```
void countIndexes( XmlContainer &container)
{
        XmlIndexSpecification idxSpec=container.getIndexSpecification(0);

        std::string uri, name, index;
        int count = 0;
        while( idxSpec.next(uri,name,index) )
        {
        // Obtain the value as a string and print it to the console
            std::cout << "For node: '" << name << "' found:\n"
                      << "\tURI: " << uri
                      << "\tIndex: " << index << std::endl;
            count ++;
        }

        std::cout << count << " indexes found." << std::endl;

}
```

Chapter 8. DBXML Exception Handling

Error conditions that occur within DBXML are reported by throwing an `XmlException`. DBXML also re-throws all underlying Berkeley DB exceptions as `XmlException`, so every exception that can be thrown by DBXML 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 DBXML 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.

All DBXML operations can throw an exception, and so they should be within a `try` block. Note that DBXML constructors do not constitute a DBXML operation, and in fact they will never throw an `XmlException`.

Note that if you are using core Berkeley DB operations with your DBXML application (such as opening a Berkeley DB environment), then you should catch `DbException` with this code. For historical reasons, `DbException` is not derived from `std::exception`, so to be safe you need to catch both types of exceptions when performing core Berkeley DB operations.

The following example illustrates DBXML exception handling. It implements a constructor for a class in which we open both a DBXML container and a Berkeley DB environment.

Example 8.1 DBXML Exception Handling

```
myXmlContainer::myXmlContainer( const std::string &containerName,
                                const std::string &envHome )
{
    if ( ! containerName.length() )
    {
        std::cerr << "Attempted to open a DBXML container with a null name\n"
                  << "Giving up.\n";
        exit(-1);
    }

    //open the database environment if a path is given to us
    if ( ! envHome.length() )
    {
        DbEnv dbEnv(0);
    } else {
        try
        {
            u_int32_t cFlags=DB_CREATE | DB_INIT_LOCK | DB_INIT_LOG |
                             DB_INIT_MPOOL | DB_INIT_TXN;
            dbEnv.open(envHome.c_str(),cFlags, 0);
        }
        //catch the DbException if it is thrown.
        catch(DbException &e)
        {
            std::cerr << "Error opening database environment: "
                      << envHome << e.what() << std::endl;
            exit( -1 );
        }
        catch(std::exception &e)
        {
            std::cerr << "Error opening database environment: "
                      << envHome << "\n"
                      << e.what() << std::endl;
            exit( -1 );
        }
    }

    //now open the container
    XmlContainer container( &dbEnv, containerName );

    try
    {
       u_int32_t cFlags= DB_CREATE | DB_AUTO_COMMIT;
       container.open(0,cFlags,0);
    }
    //catches XmlException
    catch(std::exception &e)
    {
       std::cerr << "Error opening container: " << containerName << "\n"
                 << e.what() << std::endl;
       dbEnv.close();
    }
}
```

Chapter 9. Using DBXML with Berkeley DB

As mentioned in Using Containers with Berkeley DB Environments, database environments allow your applications to simultaneously manage data stored in multiple containers and Berkeley DB databases. These resources can be efficiently shared by as many threads and processes as you require. However, it is important to realize that processes that share an environment also have access to any data that resides in the environment's shared memory regions, as well as environment buffer space and locks. For this reason, all processes that share a database environment must trust one another as they will be able to access one another's data.

One of the more interesting usages of database environments is to allow for simultaneous management of XML documents and corresponding data stored in Berkeley DB databases. For example, suppose your XML documents rely on some kind of related information that is difficult or unlikely to be queried, such as graphics images or public keys used for PGP applications. In this case, you might want to separate this data from the XML document by placing it into a Berkeley DB database that resides in an environment common to your container.

Doing this has several advantages. First, if the related information is large (such as is the case for graphics), by removing it from your XML documents your XPath queries will execute more efficiently because your documents are smaller and there is simply less to process. Retrieval is also quicker because you need not retrieve the information stored in the Berkeley DB database until you actually need it.

Note that because the container and database share a common environment, you can use transactions to maintain high integrity between your XML documents and related data stored in the Berkeley DB database. This is especially attractive for write operations where you can ensure that modifications to the container are not successful unless modifications to the database are also successful (and vice-versa).

Transactions

Coordinating data access between containers and databases is an activity that is best performed from within the scope of a transaction. Berkeley DB transaction support is a complex topic that is largely beyond the scope of this document. However, you can achieve minimal transaction protection without a great deal of effort.

Transactions allow you to combine multiple container and database operations into a single atomic unit (in particular, your write operations will not actually appear in the containers/databases until you commit the transaction).

Using transactions requires that you obtain a transaction ID from your database environment. You then hand this transaction ID to the various APIs that perform read and/or write operations on your containers and databases. Once you have completed all read and write operations that, combined, constitute a single atomic operation, you commit the transaction. If any of your read or write operations fail, you can abort the transaction and your data will remain in the same state as it would have been if you had never performed any operations on it.

When using Berkeley DB transactions, keep the following in mind:

- You can not obtain a transaction ID from the environment unless it was originally created with transaction support (the open flags included DB_INIT_TXN).
- You can not use transactions with container or database operations unless the container or database was opened with a transaction. That is, a transaction ID must have been provided to the open() method, and that transaction must have been committed before you can perform any subsequent transaction-protect operations on the container or database.
- If you opened your container or database using a transaction, then subsequent writes must be performed with a transaction as well. No such restriction exists for reads, however.
- If you are going to explicitly commit your transactions (and you will have to do this if you want to group multiple read/write operations inside a transaction), then you can not use the DB_AUTO_COMMIT flag on the container and database opens.

The process that you use to obtain and use a transaction is the same regardless of the database operation that you want to transact. The following illustrates this process using a container open:

Example 9.1 Opening a Container with a Transaction

```
//dbEnv is a database environment object

//begin the transaction
DbTxn *txn;
dbEnv.txn_begin(0, &txn, 0);

//perform the open.
//cFlags_ does NOT include DB_AUTO_COMMIT
container_.open(txn,cFlags_,0);

//commit the operation
txn->commit(0);
```

If your code detects any problems in the container access operations (for example, an exception is thrown on the open), call `txn::abort()` in your error recovery code:

```
//put this in your except block

//... other error recovery code ...
txn->abort();
//... other error recovery code ...
```

Berkeley DB Databases

Using Berkeley DB databases is described formally in the Berkeley DB Programmer's Reference manual that is included in your Berkeley DB docs directory. However, for completeness, a brief introduction is required here.

Berkeley DB database usage is conceptually the same as DBXML containers, with the following important similarities and differences:

- Instead of instantiating an object of type `XmlContainer`, you instantiate an object of type `Db`. Like an XmlContainer class, you can optionally provide a pointer to a database environment when you instantiate this class, and the class will then be used within that environment.
- Like XmlContainers, you open Berkeley DB databases with an `open()` method. When you use this method, you provide flags that describe the subsystems to be used with this database. The list of potential flags for a Berkeley DB database is a superset of that provided for XmlContainers. You also must identify the type of database that you are opening.
- A Berkeley DB database can be one of four types. There are important usage and performance reasons to use any of the four types, but for this discussion we will only use DB_BTREE which should perform well for most modestly-sized data sets.
- The fundamental unit that you write to a Berkeley DB is an object of type `Dbt`. This class constructor takes two values. The first is a `void *` which points to the data that you want to place in the object, and the second is an `unsigned int` that represents the size of the data.
- You write to the database using `Db::put()`. However, instead of a string that holds an XML document, this method requires you to provide a key/data pair, each of type `Dbt*`.
- You retrieve records from the database using `Db::get()`. This method requires you to provide a key of type `Dbt*` that is used to retrieve the data. You also provide a pointer to a `Dbt` instance that is used to store the retrieved data.
- Just like containers, you close your database when you are done using it.

These points are illustrated in the example that follows next in this chapter.

Database Records Creation Example

Typically, to relate data in Berkeley DB databases to XML documents stored in DBXML containers, you will carry the database key on the XML document in some way. As a result, in order to create database records that correspond to a XML document stored in a DBXML container, you usually first retrieve the appropriate information from a document using XPath queries, construct your database key, and then perform your database operations with that key.

The following example illustrates this point. Note that in order to simplify things, this example only creates database records using keys retrieved from XML documents stored in an XmlContainer. For a more robust example, see `buildDB.cpp` and `retrieveDB.cpp` in the DBXML examples directory.

Example 9.2 Container and Database Write

We start by opening the database environment as normal:

```
DbEnv dbEnv_( 0 );
dbEnv_.open( "/my/environment/home",
         DB_INIT_LOCK|DB_INIT_LOG|DB_INIT_MPOOL|DB_INIT_TXN, 0 );
```

Next, we get a transaction and it use it to open the database. Notice that the database is of type DB_BTREE, and that only DB_CREATE is provided for the database open flags. After the open, we call commit.

```
//Get a transaction ID
DbTxn *txn;
dbEnv_.txn_begin(0, &txn, 0);

//Open the database;
Db database_( &dbEnv_, 0 );
database_.open(txn, "testBerkeleyDB", 0, DB_BTREE, DB_CREATE, 0);

txn->commit( 0 );
```

We then open our container, using a new transaction for this operation.

```
dbEnv_.txn_begin(0, &txn, 0);

//Open the container
XmlContainer container_( &dbEnv_, "namespaceExampleData.dbxml" );
container_.open( txn, 0, 0 ); //No flags
txn->commit( 0 );
```

Now we start doing the real work. We get a transaction and query the container for the documents from which we want to retrieve database keys.

```
//Start the transaction
dbEnv_.txn_begin(0, &txn, 0);

//Perform the container query.

//No query context is used for this example
// Uses the sample data in the examples directory
XmlResults results( container_.queryWithXPath(txn, "/vendor", 0 ) );
```

Now we loop through the results set, retrieving a key and putting data in the database for each document found there.

```
//Get a query context and set the result type to ResultValues

XmlQueryContext resultsContext;
resultsContext.setReturnType( XmlQueryContext::ResultValues );

//loop through results. Database record writes occur here.
XmlDocument theDocument;


while( results.next(0, theDocument) )
{
```

As we loop through the results, we find the key on each of our documents. In this case, we're using the salesrep's name as the key.

```
        //Query on the document for the database key
        XmlResults docResult =
            theDocument.queryWithXPath("/vendor/salesrep/name/text()",
                                       &resultsContext);

        //In this example, we expect the size of the result set to be 1.
        assert( docResult.size() == 1 );

        //Pull the key value out of the document query result set.
        XmlValue docValue;
        docResult.next(txn, docValue);
        std::string theKeyString = docValue.asString();
```

We are now ready to build the key. We instantiate a Dbt object that we use for that purpose, and we set the salesrep's name as its value.

```
        //Build the key
        Dbt theKey( (void *)theKeyString.c_str(), theKeyString.length() + 1 );
```

Finally, we build the data that we want to put in the database. A real world example would be to place a large blob of data (the salesrep's photo, for example) into the database, but for simplicity we just construct a text string and store that in the database.

```
        //Build the record data. This can be anything, but for
        // clarity we'll just use a string.
        std::string theDataString =
             "This is the database record data saved for " + theKeyString;

        Dbt theData( (void *)theDataString.c_str(),
                     theDataString.length() + 1 );

        //Put the data into the database using the key we just created.
        // DB_NOOVERWRITE prevents us from overwriting a record if there is a
        // key collision
        database_.put(txn, &theKey, &theData, DB_NOOVERWRITE );


    } // end results while loop

    //End the transaction. Operation is complete
    txn->commit( 0 );
```

Appendix A. DBXML C++ API Quick Reference

This appendix identifies the classes and methods available in the DBXML C++ API.

Table A.1. XmlContainer

Method	Description
XmlContainer	An XML document container.
setPageSize	Set the page size.
open	Open a XML container.
isOpen	Test if a XML container is open.
close	Close a XML container.
getIndexSpecification	Retrieve indexing of the container.
setIndexSpecification	Specify indexing for the container.
addIndex	Add an index to a container.
deleteIndex	Delete an index from a container.
replaceIndex	Replace an index on a container.
putDocument	Store a document in the container.
updateDocument	Update a document in the container.
getDocument	Retrieve a document from the container.
deleteDocument	Delete a document from the container.
parseXPathExpression	Parse an XPath 1.0 expression.
queryWithXPath	Query a container using XPath 1.0.
getName	Return the container name.
setName	Set the name of the container.
remove	Delete the container from the file system.
rename	Rename the container.
dump	Dump the container.
load	Load the container.
verify	Verify the container.

Table A.2. XmlIndexSpecification

Method	Description
XmlIndexSpecification	An Index Specification.
addIndex	Add an index.
deleteIndex	Delete an index.
replaceIndex	Replace an index.
find	Find an index.
next	Index iterator.
reset	Reset index iterator.

Table A.3. XmlQueryContext

Method	Description
XmlQueryContext	An XPath query context.
setVariableValue	Bind a value to a variable.
clearNamespaces	Delete all namespace mappings.
getNamespace	Return the namespace URI.
getVariableValue	Return the value bound to a variable.
removeNamespace	Delete the namespace URI.
setEvaluationType	Set the query evaluation type.
setNamespace	Set the namespace URI.
setReturnType	Set the query return type.

Table A.4. XmlDocument

Method	Description
XmlDocument	An XML document.
getMetaData	Get a document metadata attribute.
getContent	Get the document content.
getDOM	Return the document as a DOM.
getID	Get the document ID.
getName	Get the document name.
setMetaData	Set a document metadata attribute.
setContent	Set the document content.
setName	Set the document name.
queryWithXPath	Query a document using XPath 1.0.

Table A.5. XmlResults

Method	Description
XmlResults	An XPath query result set.
next	Return the next result.
size	Return the size of the result set.
reset	Reset the result set iterator.

Table A.6. XmlUpdateContext

Method	Description
XmlUpdateContext	Update operation context.

Table A.7. XmlValue

Method	Description
XmlValue	The value of an XML document node.

Table A.8. XmlQueryExpression

Method	Description
`XmlQueryExpression`	A pre-parsed XPath expression.
`getXPathQuery`	Return the XPath query.

Table A.9. XmlException

Method	Description
`XmlException`	A Berkeley DB XML exception.

Table A.10. DbXml

Method	Description
`setLogLevel`	Enable logging for application debugging.

Makers of Berkeley DB