Oracle Berkeley DB XML

Getting Started with Transaction Processing For C++

Release 2.5

Legal Notice

Published 8/25/2009

Table of Contents

Preface

This document describes how to use transactions with your Berkeley DB XML applications. It is intended to describe how to transaction protect your application's data. The APIs used to perform this task are described here, as are the environment infrastructure and administrative tasks required by a transactional application. This book also describes multi-threaded BDB XML applications and the requirements they have for deadlock detection.

This book is aimed at the software engineer responsible for writing a transactional BDB XML application.

This book assumes that you have already read and understood the concepts contained in the *Getting Started with Berkeley DB XML* guide.

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: "`DbEnv::open()` is a `DbEnv` class method."

Structure names are represented in `monospaced font`, as are `method names`. For example: "`DB->open()` is a method on a `DB` handle."

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

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

```
typedef struct vendor {
    char name[MAXFIELD];            // Vendor name
    char street[MAXFIELD];          // Street name and number
    char city[MAXFIELD];            // City
    char state[3];                  // Two-digit US state code
    char zipcode[6];                // US zipcode
    char phone_number[13];          // Vendor phone number
} VENDOR;
```

In some situations, programming examples are updated from one chapter to the next. When this occurs, the new code is presented in **`monospaced bold`** font. For example:

```
typedef struct vendor {
    char name[MAXFIELD];            // Vendor name
    char street[MAXFIELD];          // Street name and number
    char city[MAXFIELD];            // City
    char state[3];                  // Two-digit US state code
    char zipcode[6];                // US zipcode
    char phone_number[13];          // Vendor phone number
    char sales_rep[MAXFIELD];       // Name of sales representative
```

```
    char sales_rep_phone[MAXFIELD];  // Sales rep's phone number
} VENDOR;
```

Finally, notes of special interest are represented using a note block such as this.

For More Information

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

- Introduction to Berkeley DB XML [http://www.oracle.com/technology/documentation/berkeley-db/xml/intro_xml/BerkeleyDBXML-Intro.pdf]
- Getting Started with Berkeley DB XML for C++ [http://www.oracle.com/technology/documentation/berkeley-db/xml/gsg_xml/cxx/BerkeleyDBXML-CXX-GSG.pdf]
- Berkeley DB Programmer's Reference Guide [http://www.oracle.com/technology/documentation/berkeley-db/db/programmer_reference/BDB_Prog_Reference.pdf]
- Berkeley DB XML C++ API [http://www.oracle.com/technology/documentation/berkeley-db/xml/api_cxx/frame.html]

Chapter 1. Introduction

This book provides a thorough introduction and discussion on transactions as used with Berkeley DB XML (BDB XML). It begins by offering a general overview to transactions, the guarantees they provide, and the general application infrastructure required to obtain full transactional protection for your data.

This book also provides detailed examples on how to write a transactional application. Both single threaded and multi-threaded (as well as multi-process applications) are discussed. A detailed description of various backup and recovery strategies is included in this manual, as is a discussion on performance considerations for your transactional application.

You should understand the concepts from the *Getting Started with Berkeley DB XML* guide before reading this book.

Why Transactions?

It is entirely possible that your application does not require transactions, and that you can therefore skip this manual. That is, for what applications are transactions useful?

It is true that applications which are mostly involved in read-only access to their containers can and probably should avoid transactions. But even applications that perform a lot of container writes might be able to skip using transactions.

If any of the following conditions are true, then you should consider using transactions, and so need to read this manual:

- You need to perform multiple modifications to your containers in such a way as all the modifications succeed, or they all fail.
- You are writing an application that might attempt to update the same document simultaneously from multiple threads or even processes.
- Your application needs to perform long-running document updates — updates that could take minutes or even hours to complete — and you want to be sure that the document is left in a consistent state at the end of the update, regardless of whether the update succeeds or fails.

Transaction Benefits

Transactions offer your application's data protection from application or system failures. That is, BDB XML transactions offer your application full ACID support:

- **A**tomicity

  Multiple container operations are treated as a single unit of work. Once committed, all write operations performed under the protection of the transaction are saved to your containers. Further, in the event that you abort a transaction, all write operations performed during the transaction are discarded. In this event, your container is left in the state it was in before

the transaction began, regardless of the number or type of write operations you may have performed during the course of the transaction.

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

- **C**onsistency

  Your containers will never see a partially completed transaction. This is true even if your application fails while there are in-progress transactions. If the application or system fails, then either all of the container changes appear when the application next runs, or none of them appear.

  In other words, whatever consistency requirements your application has will never be violated by BDB XML. If, for example, your application requires every record to include an employee ID, and your code faithfully adds that ID to its container records, then BDB XML will never violate that consistency requirement. The ID will remain in the container records until such a time as your application chooses to delete it.

- **I**solation

  While a transaction is in progress, your containers will appear to the transaction as if there are no other operations occurring outside of the transaction. That is, operations wrapped inside a transaction will always have a clean and consistent view of your databases. They never have to see updates currently in progress under the protection of another transaction. Note, however, that isolation guarantees can be relaxed from the default setting. See Isolation (page 43) for more information.

- **D**urability

  Once committed to your containers, your modifications will persist even in the event of an application or system failure. Note that like isolation, your durability guarantee can be relaxed. See Non-Durable Transactions (page 21) for more information.

A Note on System Failure

From time to time this manual mentions that transactions protect your data against 'system or application failure.' This is true up to a certain extent. However, not all failures are created equal and no data protection mechanism can protect you against every conceivable way a computing system can find to die.

Generally, when this book talks about protection against failures, it means that transactions offer protection against the likeliest culprits for system and application crashes. So long as your data modifications have been committed to disk, those modifications should persist even if your application or OS subsequently fails. And, even if the application or OS fails in the middle of a transaction commit (or abort), the data on disk should be either in a consistent state, or there should be enough data available to bring your containers into a consistent state (via a recovery procedure, for example). You may, however, lose whatever data you were committing at the time of the failure, but your containers will be otherwise unaffected.

Be aware that many disks have a disk write cache and on some systems it is enabled by default. This means that a transaction can have committed, and to your application the data may appear to reside on disk, but the data may in fact reside only in the write cache at that time. This means that if the disk write cache is enabled and there is no battery backup for it, data can be lost after an OS crash even when maximum durability mode is in use. For maximum durability, disable the disk write cache or use a disk write cache with a battery backup.

Of course, if your *disk* fails, then the transactional benefits described in this book are only as good as the backups you have taken. By spreading your data and log files across separate disks, you can minimize the risk of data loss due to a disk failure, but even in this case it is possible to conjure a scenario where even this protection is insufficient (a fire in the machine room, for example) and you must go to your backups for protection.

Finally, by following the programming examples shown in this book, you can write your code so as to protect your data in the event that your code crashes. However, no programming API can protect you against logic failures in your own code; transactions cannot protect you from simply writing the wrong thing to your containers.

Application Requirements

In order to use transactions, your application has certain requirements beyond what is required of non-transactional protected applications. They are:

- Environments.

  Environments are always used by BDB XML, but for transactional applications you must instantiate and manage your `DB_ENV` object independently of your `XmlManager`.

  Environment usage is described in detail in Transaction Basics (page 18).

- Transaction subsystem.

  In order to use transactions, you must explicitly enable the transactional subsystem for your application, and this must be done at the time that your environment is first created.

- Logging subsystem.

  The logging subsystem is required for recovery purposes, but its usage also means your application may require a little more administrative effort than it does when logging is not in use. See Managing BDB XML Files (page 53) for more information.

- XmlTransaction handles.

  In order to obtain the atomicity guarantee offered by the transactional subsystem (that is, combine multiple operations in a single unit of work), your application must use transaction handles. These handles are obtained from your XmlManager objects. They should normally be short-lived, and their usage is reasonably simple. To complete a transaction and save the work it performed, you call its `commit()` method. To complete a transaction and discard its work, you call its `abort()` method.

In addition, it is possible to use auto commit if you want to transactional protect a single write operation. Auto commit allows a transaction to be used without obtaining an explicit transaction handle. See Auto Commit (page 23) for information on how to use auto commit.

- Container open requirements

  In addition to using environments and initializing the correct subsystems, your application must call `XmlContainerConfig::setTransactional(true)` for the configuration object used for your container.

- Deadlock detection.

  Typically transactional applications use multiple threads of control when accessing the database. Any time multiple threads are used on a single resource, the potential for lock contention arises. In turn, lock contention can lead to deadlocks. See Locks, Blocks, and Deadlocks (page 31) for more information.

  Therefore, transactional applications must frequently include code for detecting and responding to deadlocks. Note that this requirement is not *specific* to transactions – you can certainly write concurrent non-transactional BDB XML applications. Further, not every transactional application uses concurrency and so not every transactional application must manage deadlocks. Still, deadlock management is so frequently a characteristic of transactional applications that we discuss it in this book. See Concurrency (page 29) for more information.

Multi-threaded and Multi-process Applications

BDB XML is designed to support multi-threaded and multi-process applications, but their usage means you must pay careful attention to issues of concurrency. Transactions help your application's concurrency by providing various levels of isolation for your threads of control. In addition, BDB XML provides mechanisms that allow you to detect and respond to deadlocks.

Isolation means that container modifications made by one transaction will not normally be seen by readers from another transaction until the first commits its changes. Different threads use different transaction handles, so this mechanism is normally used to provide isolation between container operations performed by different threads.

Note that BDB XML supports different isolation levels. For example, you can configure your application to see uncommitted reads, which means that one transaction can see data that has been modified but not yet committed by another transaction. Doing this might mean your transaction reads data "dirtied" by another transaction, but which subsequently might change before that other transaction commits its changes. On the other hand, lowering your isolation requirements means that your application can experience improved throughput due to reduced lock contention.

For more information on concurrency, on managing isolation levels, and on deadlock detection, see Concurrency (page 29).

Recoverability

An important part of BDB XML's transactional guarantees is durability. *Durability* means that once a transaction has been committed, the container modifications performed under its protection will not be lost due to system failure.

In order to provide the transactional durability guarantee, BDB XML uses a write-ahead logging system. Every operation performed on your containers is described in a log before it is performed on your containers. This is done in order to ensure that an operation can be recovered in the event of an untimely application or system failure.

Beyond logging, another important aspect of durability is recoverability. That is, backup and restore. BDB XML supports a normal recovery that runs against a subset of your log files. This is a routine procedure used whenever your environment is first opened upon application startup, and it is intended to ensure that your container is in a consistent state. BDB XML also supports archival backup and recovery in the case of catastrophic failure, such as the loss of a physical disk drive.

This book describes several different backup procedures you can use to protect your on-disk data. These procedures range from simple offline backup strategies to hot failovers. Hot failovers provide not only a backup mechanism, but also a way to recover from a fatal hardware failure.

This book also describes the recovery procedures you should use for each of the backup strategies that you might employ.

For a detailed description of backup and restore procedures, see Managing BDB XML Files (page 53).

Performance Tuning

From a performance perspective, the use of transactions is not free. Depending on how you configure them, transaction commits usually require your application to perform disk I/O that a non-transactional application does not perform. Also, for multi-threaded and multi-process applications, the use of transactions can result in increased lock contention due to extra locking requirements driven by transactional isolation guarantees.

There is therefore a performance tuning component to transactional applications that is not applicable for non-transactional applications (although some tuning considerations do exist whether or not your application uses transactions). Where appropriate, these tuning considerations are introduced in the following chapters. However, for a more complete description of them, see the Transaction tuning [http://www.oracle.com/technology/documentation/berkeley-db/db/ref/transapp/tune.html] and Transaction throughput [http://www.oracle.com/technology/documentation/berkeley-db/db/ref/transapp/throughput.html] sections of the *Berkeley DB Programmer's Reference Guide*.

Chapter 2. Enabling Transactions

In order to use transactions with your application, you must turn them on. To do this you must:

- Use an externally-managed environment (see Environments (page 6) for details).
- Turn on transactions for your environment. You do this by providing the `DB_INIT_TXN` flag to the `DB_ENV->open()` method. Note that initializing the transactional subsystem implies that the logging subsystem is also initialized. Also, note that if you do not initialize transactions when you first create your environment, then you cannot use transactions for that environment after that. This is because BDB XML allocates certain structures needed for transactional locking that are not available if the environment is created without transactional support.
- Initialize the in-memory cache by passing the `DB_INIT_MPOOL` flag to the `DB_ENV->open()` method.
- Initialize the locking subsystem. This is what provides locking for concurrent applications. It also is used to perform deadlock detection. See Concurrency (page 29) for more information.

  You initialize the locking subsystem by passing the `DB_INIT_LOCK` flag to the `DB_ENV->open()` method.
- Initialize the logging subsystem. While this is enabled by default for transactional applications, we suggest that you explicitly initialize it anyway for the purposes of code readability. The logging subsystem is what provides your transactional application its durability guarantee, and it is required for recoverability purposes. See Managing BDB XML Files (page 53) for more information.

  You initialize the logging subsystem by passing the `DB_INIT_LOG` flag to the `DB_ENV->open()` method.
- Transaction-enable your containers. If you are using the base API, transaction-enable your containers. You do this by using the `XmlContainerConfig::setTransactional()` method and then pass that object when you open or create the container.

Environments

All BDB XML applications use environments. However, simple BDB XML applications can use a default environment that only provide a small subset of features — most notably, multi-threaded access (but not multi-process) is enabled, as is the in-memory cache. For more advanced features, such as transactional protection, you must use an externally-managed environment.

An *environment*, represents an encapsulation of one or more containers and any associated log and region files. They are used to support multi-threaded and multi-process applications by allowing different threads of control to share the in-memory cache, the locking tables, the logging subsystem, and the file namespace. By sharing these things, your concurrent application is more efficient than if each thread of control had to manage these resources on its own.

By default all BDB XML containers are backed by files on disk. In addition to these files, transactional BDB XML applications create logs that are also by default stored on disk (they can optionally be backed using shared memory). Finally, transactional BDB XML applications also create and use shared-memory regions that are also typically backed by the filesystem. But like containers and logs, the regions can be maintained strictly in-memory if your application requires it. For an example of an application that manages all environment files in-memory, see In-Memory Transaction Example (page 84).

File Naming

In order to operate, your BDB XML application must be able to locate its container files, log files, and region files. If these are stored in the filesystem, then you must tell BDB XML where they are located (a number of mechanisms exist that allow you to identify the location of these files – see below). Otherwise, by default they are located in the current working directory.

Specifying the Environment Home Directory

The environment home directory is used to determine where BDB XML files are located. Its location is identified using one of the following mechanisms, in the following order of priority:

- If no information is given as to where to put the environment home, then the current working directory is used.
- If a home directory is specified on the `DB_ENV->open()` method, then that location is always used for the environment home.
- If a home directory is not supplied to `DB_ENV->open()`, then the directory identified by the `DB_HOME` environment variable is used *if* you specify either the `DB_USE_ENVIRON` or `DB_USE_ENVIRON_ROOT` flags to the `DB_ENV->open()` method. Both flags allow you to identify the path to the environment's home directory using the `DB_HOME` environment variable. However, `DB_USE_ENVIRON_ROOT` is honored only if the process is run with root or administrative privileges.

Specifying File Locations

By default, all BDB XML files are created relative to the environment home directory. For example, suppose your environment home is in `/export/myAppHome`. Also suppose you name your container `data/myContainer.dbxml`. Then in this case, the container is placed in: `/export/myAppHome/data/myContainer.dbxml`.

That said, BDB XML always defers to absolute pathnames. This means that if you provide an absolute filename when you name your container, then that file is *not* placed relative to the environment home directory. Instead, it is placed in the exact location that you specified for the filename.

On UNIX systems, an absolute pathname is a name that begins with a forward slash ('/'). On Windows systems, an absolute pathname is a name that begins with one of the following:

- A backslash ('\').

- Any alphabetic letter, followed by a colon (':'), followed by a backslash ('\').

Try not to use absolute path names for your environment's files. Under certain recovery scenarios, absolute path names can render your environment unrecoverable. This occurs if you are attempting to recover your environment on a system that does not support the absolute path name that you used.

Identifying Specific File Locations

As described in the previous sections, BDB XML will place all its files in or relative to the environment home directory. You can also cause a specific container file to be placed in a particular location by using an absolute path name for its name. In this situation, the environment's home directory is not considered when naming the file.

It is frequently desirable to place container, log, and region files on separate disk drives. By spreading I/O across multiple drives, you can increase parallelism and improve throughput. Additionally, by placing log files and container files on separate drives, you improve your application's reliability by providing your application with a greater chance of surviving a disk failure.

You can cause BDB XML's files to be placed in specific locations using the following mechanisms:

File Type	To Override
container files	You can cause container files to be created in a directory other than the environment home by using the `DB_ENV->set_data_dir()` method. The directory identified here must exist. If a relative path is provided, then the directory location is resolved relative to the environment's home directory. This method modifies the directory used for container files created and managed by a single environment handle; it does not configure the entire environment. This method may not be called after the environment has been opened. You can also set a default data location that is used by the entire environment by using the `set_data_dir` parameter in the environment's `DB_CONFIG` file. Note that the `set_data_dir` parameter overrides any value set by the `DB_ENV->set_data_dir()` method.
Log files	You can cause log files to be created in a directory other than the environment home directory by using the `DB_ENV->set_lg_dir()` method. The directory identified here must exist. If a relative path is provided, then the

File Type	To Override
	directory location is resolved relative to the environment's home directory. This method modifies the directory used for container files created and managed by a single environment handle; it does not configure the entire environment. This method may not be called after the environment has been opened. You can also set a default log file location that is used by the entire environment by using the `set_lg_dir` parameter in the environment's `DB_CONFIG` file. Note that the `set_lg_dir` parameter overrides any value set by the `DB_ENV->set_lg_dir()` method.
Region files	If backed by the filesystem, region files are always placed in the environment home directory.

Note that the `DB_CONFIG` must reside in the environment home directory. Parameters are specified in it one parameter to a line. Each parameter is followed by a space, which is followed by the parameter value. For example:

```
    set_data_dir /export1/db/env_data_files
```

Error Support

To simplify error handling and to aid in application debugging, environments offer several useful methods. They are:

- `set_errcall()`

  Defines the function that is called when an error message is issued by BDB XML. The error prefix and message are passed to this callback. It is up to the application to display this information correctly.

  This is the recommended way to get error messages from BDB XML.

- `set_errfile()`

  Sets the C library `FILE *` to be used for displaying error messages issued by the BDB XML library.

- `set_errpfx()`

  Sets the prefix used to for any error messages issued by the BDB XML library.

- `err()`

Issues an error message based upon a BDB XML error code a message text that you supply. The error message is sent to the callback function as defined by `set_errcall()`. If that method has not been used, then the error message is sent to the file defined by `set_errfile()` or `set_error_stream()`. If none of these methods have been used, then the error message is sent to standard error.

The error message consists of the prefix string (as defined by `set_errprefix()`), an optional `printf`-style formatted message, the BDB XML error message associated with the supplied error code, and a trailing newline.

- `errx()`

  Behaves identically to `err()` except that you do not provide the BDB XML error code and so the BDB XML message text is not displayed.

In addition, you can use the `db_strerror()` function to directly return the error string that corresponds to a particular error number. For more information on the `db_strerror()` function, see the `Error Returns` section of the *Getting Started with Berkeley DB* guide.

Shared Memory Regions

The subsystems that you enable for an environment (in our case, transaction, logging, locking, and the memory pool) are described by one or more regions. The regions contain all of the state information that needs to be shared among threads and/or processes using the environment.

Regions may be backed by the file system, by heap memory, or by system shared memory.

Regions Backed by Files

By default, shared memory regions are created as files in the environment's home directory (*not* the environment's data directory). If it is available, the POSIX `mmap` interface is used to map these files into your application's address space. If `mmap` is not available, then the UNIX `shmget` interfaces are used instead (again, if they are available).

In this default case, the region files are named `__db.###` (for example, `__db.001`, `__db.002`, and so on).

Regions Backed by Heap Memory

If heap memory is used to back your shared memory regions, the environment may only be accessed by a single process, although that process may be multi-threaded. In this case, the regions are managed only in memory, and they are not written to the filesystem. You indicate that heap memory is to be used for the region files by specifying `DB_PRIVATE` to the `DB_ENV->open()` method.

(For an example of an entirely in-memory transactional application, see In-Memory Transaction Example (page 84).)

Regions Backed by System Memory

Finally, you can cause system memory to be used for your regions instead of memory-mapped files. You do this by providing `DB_SYSTEM_MEM` to the `DB_ENV->open()` method.

When region files are backed by system memory, BDB XML creates a single file in the environment's home directory. This file contains information necessary to identify the system shared memory in use by the environment. By creating this file, BDB XML enables multiple processes to share the environment.

The system memory that is used is architecture-dependent. For example, on systems supporting X/Open-style shared memory interfaces, such as UNIX systems, the `shmget(2)` and related System V IPC interfaces are used.

On Windows platforms, the use of system memory for the region files is problematic because the operating system uses reference counting to clean up shared objects in the paging file automatically. In addition, the default access permissions for shared objects are different from files, which may cause problems when an environment is accessed by multiple processes running as different users. See Windows notes or more information.

Security Considerations

When using environments, there are some security considerations to keep in mind:

- Database environment permissions

  The directory used for the environment should have its permissions set to ensure that files in the environment are not accessible to users without appropriate permissions. Applications that add to the user's permissions (for example, UNIX `setuid` or `setgid` applications), must be carefully checked to not permit illegal use of those permissions such as general file access in the environment directory.

- Environment variables

  Setting so that environment variables can be used during file naming can be dangerous. Setting those flags in BDB XML applications with additional permissions (for example, UNIX `setuid` or `setgid` applications) could potentially allow users to read and write containers to which they would not normally have access.

  For example, suppose you write a BDB XML application that runs `setuid`. This means that when the application runs, it does so under a userid different than that of the application's caller. This is especially problematic if the application is granting stronger privileges to a user than the user might ordinarily have.

  Now, if then the environment that the application is using is modifiable using the `DB_HOME` environment variable. In this scenario, if the uid used by the application has sufficiently broad privileges, then the application's caller can read and/or write containers owned by another user simply by setting his `DB_HOME` environment variable to the environment used by that other user.

Note that this scenario need not be malicious; the wrong environment could be used by the application simply by inadvertently specifying the wrong path to DB_HOME.

As always, you should use setuid sparingly, if at all. But if you do use setuid, then you should refrain from specifying for the environment open. And, of course, if you must use setuid, then make sure you use the weakest uid possible – preferably one that is used only by the application itself.

- File permissions

  By default, BDB XML always creates container and log files readable and writable by the owner and the group (that is, S_IRUSR, S_IWUSR, S_IRGRP and S_IWGRP; or octal mode 0660 on historic UNIX systems). The group ownership of created files is based on the system and directory defaults, and is not further specified by BDB XML.

- Temporary backing files

  If your BDB XML application is also using Berkeley DB databases, then you should pay attention to temporary backing files.

  If an unnamed database is created and the cache is too small to hold the database in memory, Berkeley DB will create a temporary physical file to enable it to page the database to disk as needed. In this case, environment variables such as TMPDIR may be used to specify the location of that temporary file. Although temporary backing files are created readable and writable by the owner only (S_IRUSR and S_IWUSR, or octal mode 0600 on historic UNIX systems), some filesystems may not sufficiently protect temporary files created in random directories from improper access. To be absolutely safe, applications storing sensitive data in unnamed databases should use the DB_ENV->set_tmp_dir() method to specify a temporary directory with known permissions.

Opening a Transactional Environment and Container

To enable transactions for your environment, you must initialize the transactional subsystem. Note that doing this also initializes the logging subsystem. In addition, you must initialize the memory pool (in-memory cache). You must also initialize the locking subsystem. For example:

Notice in the following example that you first create the environment handle, and then you provide the handle to the XmlManager constructor. You do this because you cannot use transactions with the XmlManager instance's default internal environment.

```
#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

    DB_ENV *myEnv = NULL;
    XmlManager *myManager = NULL;
    char *envHome = "/export1/testEnv";
    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);
    myManager = new XmlManager(myEnv, 0);

    try {
        if (myManager != NULL) {
            delete myManager;
        }
        myEnv->close(myEnv, 0);
    } catch(XmlException &e) {
        std::cerr << "Error closing manager: "
                << envHome << std::endl;
        std::cerr << e.what() << std::endl;
        return (EXIT_FAILURE);
    } catch(std::exception &e) {
        std::cerr << "Error closing environment: "
                << envHome << std::endl;
        std::cerr << e.what() << std::endl;
        return (EXIT_FAILURE);
    }
    return (EXIT_SUCCESS);
}
```

You then create and/or open your containers as normal. The only difference is that you must set `XmlContainerConfig::setTransactional()` to `true` and pass that object to the `openContainer()` or `createContainer()` method. For example:

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

using namespace DbXml;
int main(void)
{
```

```
u_int32_t env_flags = DB_CREATE     |  // If the environment does not
                                       // exist, create it.
                      DB_INIT_LOCK  |  // Initialize locking
                      DB_INIT_LOG   |  // Initialize logging
                      DB_INIT_MPOOL |  // Initialize the cache
                      DB_INIT_TXN;     // Initialize transactions

DB_ENV *myEnv = NULL;
XmlManager *myManager = NULL;
char *envHome = "/export1/testEnv";
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);
myManager = new XmlManager(myEnv, 0);

try {
    XmlContainerConfig cconfig;
    cconfig.setAllowCreate(true);    // If the container does not
                                     // exist, create it.
    cconfig.setTransactional(true);  // Enable transactions.

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

} catch(XmlException &e) {
    std::cerr << "Error opening environment: "
              << envHome << std::endl;
    std::cerr << e.what() << std::endl;
    return (EXIT_FAILURE);
} catch(std::exception &e) {
    std::cerr << "Error opening environment: "
              << envHome
              << " or opening XmlManager or XmlContainer." << std::endl;
    std::cerr << e.what() << std::endl;
    return (EXIT_FAILURE);
}

try {
    if (myManager != NULL) {
```

```
            delete myManager;
        }
        myEnv.close(0);
    } catch(DbException &e) {
        std::cerr << "Error closing environment: "
                << envHome << std::endl;
        std::cerr << e.what() << std::endl;
        return (EXIT_FAILURE);
    } catch(std::exception &e) {
        std::cerr << "Error closing environment: "
                << envHome << std::endl;
        std::cerr << e.what() << std::endl;
        return (EXIT_FAILURE);
    }
    return (EXIT_SUCCESS);
}
```

Opening Berkeley DB Databases

It is possible to use Berkeley DB databases along side of BDB XML containers. When you do this, you will typically use both the databases and the containers from within the same environment so that you can combine operations to both using transactions.

There is no difference between opening a Berkeley DB database in an environment that uses containers and opening a database in an environment that does not use containers (see the *Berkeley DB Getting Started with Transaction Processing* guide for details on how to do this). You simply share the same environment handle between the two when you open the database(s) and container(s). For example:

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

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

    u_int32_t db_flags = DB_CREATE | DB_AUTO_COMMIT;
    Db *dbp = NULL;
    const char *file_name = "mydb.db";

    DB_ENV *myEnv = NULL;
    XmlManager *myManager = NULL;
    char *envHome = "/export1/testEnv";
    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);
myManager = new XmlManager(myEnv, 0);
dbp = db_create(&dbp, myEnv, 0);
dbp->open(NULL,       // Txn pointer
            file_name,  // File name
            NULL,       // Logical db name
            DB_BTREE,   // Database type (using btree)
            db_flags,   // Open flags
            0);         // File mode. Using defaults
try {
    XmlContainerConfig cconfig;
    cconfig.setAllowCreate(true);    // If the container does not
                                     // exist, create it.
    cconfig.setTransactional(true);  // Enable transactions.

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

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

try {
    if (dbp != NULL) {
        dbp->close(dbp, 0);
    }

    if (myManager != NULL) {
        delete myManager;
    }
```

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

Never close a database that has active transactions. Make sure all transactions are resolved (either committed or aborted) before closing the database.

Chapter 3. Transaction Basics

Once you have enabled transactions for your environment and your containers, you can use them to protect your container operations. You do this by acquiring a transaction handle and then using that handle for any database operation that you want to participate in that transaction.

You obtain a transaction handle using the `XmlManager::createTransaction()` method.

Once you have completed all of the operations that you want to include in the transaction, you must commit the transaction using the `XmlTransaction::commit()` method.

If, for any reason, you want to abandon the transaction, you abort it using `XmlTransaction::abort()`. Note that if the last remaining `XmlTransaction` handle goes out of scope without being resolved, then the transaction is automatically aborted.

Any transaction handle that has been committed or aborted can no longer be used by your application.

Finally, you must make sure that all transaction handles are either committed or aborted before closing your containers and environment.

If you only want to transaction protect a single container write operation, you can use auto commit to perform the transaction administration. When you use auto commit, you do not need an explicit transaction handle. See Auto Commit (page 23) for more information.

For example, the following example opens a transactional-enabled environment and container, obtains a transaction handle, and then performs a write operation under its protection. In the event of any failure in the write operation, the transaction is aborted and the container is left in a state as if no operations had ever been attempted in the first place.

```
#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

    DB_ENV *myEnv = 0;
    XmlManager *myManager = NULL;
    char *envHome = "/export1/testEnv";
    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);
    myManager = new XmlManager(myEnv, 0);

    try {
        XmlContainerConfig cconfig;
        cconfig.setAllowCreate(true);    // If the container does not
                                         // exist, create it.
        cconfig.setTransactional(true);  // Enable transactions.

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

    } catch(DbException &e) {
        std::cerr << "Error opening container: "
                  << e.what() << std::endl;
        return (EXIT_FAILURE);
    } catch(std::exception &e) {
        std::cerr << "Error opening container: "
                  << e.what() << std::endl;
        return (EXIT_FAILURE);
    }

    // File to open
    std::string file1 = "doc1.xml";

    // Transaction handle
    XmlTransaction txn = myManager.createTransaction();
    try {
        // Need an update context for the put.
        XmlUpdateContext theContext = myManager.createUpdateContext();

        // Get the input stream.
        XmlInputStream *theStream =
            myManager.createLocalFileInputStream(file1);

        // Put the first document
        myContainer.putDocument(txn,         // the transaction object
                                file1,       // The document's name
                                theStream,   // The actual document.
```

```
                                theContext,  // The update context
                                             // (required).
                                0);          // Put flags.

        // Finished. Now commit the transaction.
        txn.commit();

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

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

Committing a Transaction

In order to fully understand what is happening when you commit a transaction, you must first understand a little about what BDB XML is doing with the logging subsystem. Logging causes all container write operations to be identified in logs, and by default these logs are backed by files on disk. These logs are used to restore your containers in the event of a system or application failure, so by performing logging, BDB XML ensures the integrity of your data.

Moreover, BDB XML performs *write-ahead* logging. This means that information is written to the logs *before* the actual container is changed. This means that all write activity performed under the protection of the transaction is noted in the log before the transaction is committed. Be aware, however, that container maintains logs in-memory. If you are backing your logs on disk, the log information will eventually be written to the log files, but while the transaction is on-going the log data may be held only in memory.

When you commit a transaction, the following occurs:

- A commit record is written to the log. This indicates that the modifications made by the transaction are now permanent. By default, this write is performed synchronously to disk so the commit record arrives in the log files before any other actions are taken.
- Any log information held in memory is (by default) synchronously written to disk. Note that this requirement can be relaxed, depending on the type of commit you perform. See Non-Durable Transactions (page 21) for more information. Also, if you are maintaining your logs entirely in-memory, then this step will of course not be taken. To configure your logging system for in-memory usage, see Configuring In-Memory Logging (page 69).
- All locks held by the transaction are released. This means that read operations performed by other transactions or threads of control can now see the modifications without resorting to uncommitted reads (see Reading Uncommitted Data (page 45) for more information).

To commit a transaction, you simply call `XmlTransaction::commit()`.

Notice that committing a transaction does not necessarily cause data modified in your memory cache to be written to the files backing your containers on disk. Dirtied database pages are written for a number of reasons, but a transactional commit is not one of them. The following are the things that can cause a dirtied database page to be written to the backing database file:

- Checkpoints.

  Checkpoints cause all dirtied pages currently existing in the cache to be written to disk, and a checkpoint record is then written to the logs. You can run checkpoints explicitly. For more information on checkpoints, see Checkpoints (page 53).

- Cache is full.

  If the in-memory cache fills up, then dirtied pages might be written to disk in order to free up space for other pages that your application needs to use. Note that if dirtied pages are written to the database files, then any log records that describe how those pages were dirtied are written to disk before the database pages are written.

Be aware that because your transaction commit caused container modifications recorded in your logs to be forced to disk, your modifications are by default "persistent" in that they can be recovered in the event of an application or system failure. However, recovery time is gated by how much data has been modified since the last checkpoint, so for applications that perform a lot of writes, you may want to run a checkpoint with some frequency.

Note that once you have committed a transaction, the transaction handle that you used for the transaction is no longer valid. To perform container activities under the control of a new transaction, you must obtain a fresh transaction handle.

Non-Durable Transactions

As previously noted, by default transaction commits are durable because they cause the modifications performed under the transaction to be synchronously recorded in your on-disk log files. However, it is possible to use non-durable transactions.

You may want non-durable transactions for performance reasons. For example, you might be using transactions simply for the isolation guarantee. In this case, you might not want a durability guarantee and so you may want to prevent the disk I/O that normally accompanies a transaction commit.

There are several ways to remove the durability guarantee for your transactions:

- Specify `DB_TXN_NOSYNC` using the `DB_ENV->set_flags()` method. This causes BDB XML to not synchronously force any log data to disk upon transaction commit. That is, the modifications are held entirely in the in-memory cache and the logging information is not forced to the filesystem for long-term storage. Note, however, that the logging data will eventually make it to the filesystem (assuming no application or OS crashes) as a part of BDB XML's management of its logging buffers and/or cache.

  This form of a commit provides a weak durability guarantee because data loss can occur due to an application or OS crash.

  This behavior is specified on a per-environment handle basis. In order for your application to exhibit consistent behavior, you need to specify this flag for all of the environment handles used in your application.

  You can achieve this behavior on a transaction by transaction basis by specifying `DB_TXN_NOSYNC` to the `XmlTransaction::commit()` method.

- Specify `DB_TXN_WRITE_NOSYNC` using the `DB_ENV->set_flags()` method. This causes logging data to be synchronously written to the OS's file system buffers upon transaction commit. The data will eventually be written to disk, but this occurs when the operating system chooses to schedule the activity; the transaction commit can complete successfully before this disk I/O is performed by the OS.

  This form of commit protects you against application crashes, but not against OS crashes. This method offers less room for the possibility of data loss than does `DB_TXN_NOSYNC`.

  This behavior is specified on a per-environment handle basis. In order for your application to exhibit consistent behavior, you need to specify this flag for all of the environment handles used in your application.

- Maintain your logs entirely in-memory. In this case, your logs are never written to disk. The result is that you lose all durability guarantees. See Configuring In-Memory Logging (page 69) for more information.

Aborting a Transaction

When you abort a transaction, all database modifications performed under the protection of the transaction are discarded, and all locks currently held by the transaction are released. In this event, your data is simply left in the state that it was in before the transaction began performing data modifications.

Once you have aborted a transaction, the transaction handle that you used for the transaction is no longer valid. To perform database activities under the control of a new transaction, you must obtain a fresh transactional handle.

To abort a transaction, call `XmlTransaction::abort()`.

Auto Commit

While transactions are frequently used to provide atomicity to multiple container operations, it is sometimes necessary to perform a single container operation under the control of a transaction. Rather than force you to obtain a transaction, perform the single write operation, and then either commit or abort the transaction, you can automatically group this sequence of events using *auto commit*.

To use auto commit:

1. Open your environment and your containers so that they support transactions. See Enabling Transactions (page 6) for details.

   Note that frequently auto commit is used for the environment or container open. To use auto commit for either your environment or container open, specify `DB_AUTO_COMMIT` to the or method. If you specify auto commit for the environment open, then you do not need to also specify auto commit for the container open.

2. Do not provide a transactional handle to the method that is performing the container write operation.

Never have more than one active transaction in your thread at a time. This is especially a problem if you mix an explicit transaction with another operation that uses auto commit. Doing so can result in undetectable deadlocks.

For example, the following uses auto commit to perform the container write operation:

```
#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

    DB_ENV *myEnv = 0;
    XmlManager *myManager = NULL;
    char *envHome = "/export1/testEnv";
    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);
myManager = new XmlManager(myEnv, 0);

try {
    XmlContainerConfig cconfig;
    cconfig.setAllowCreate(true);    // If the container does not
                                     // exist, create it.
    cconfig.setTransactional(true);  // Enable transactions.

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

} catch(DbException &e) {
    std::cerr << "Error opening container: "
              << e.what() << std::endl;
    return (EXIT_FAILURE);
} catch(std::exception &e) {
    std::cerr << "Error opening container: "
              << e.what() << std::endl;
    return (EXIT_FAILURE);
}

// File to open
std::string file1 = "doc1.xml";

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

    // Get the input stream.
    XmlInputStream *theStream =
        myManager.createLocalFileInputStream(file1);

    // Put the document. Because the container was opened to
    // support transactions, this write is performed using
    // auto commit.
    myContainer.putDocument(file1,       // The document's name
                            theStream,   // The actual document.
```

```
                                theContext,  // The update context
                                             // (required).
                                0);          // Put flags.

    } catch(XmlException &e) {
        std::cerr << "Error in write: "
                  << e.what() << std::endl;
    }

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

Nested Transactions

A *nested transaction* is used to provide a transactional guarantee for a subset of operations performed within the scope of a larger transaction. Doing this allows you to commit and abort the subset of operations independently of the larger transaction.

The rules to the usage of a nested transaction are as follows:

- While the nested (child) transaction is active, the parent transaction may not perform any operations other than to commit or abort, or to create more child transactions.
- Committing a nested transaction has no effect on the state of the parent transaction. The parent transaction is still uncommitted. However, the parent transaction can now see any modifications made by the child transaction. Those modifications, of course, are still hidden to all other transactions until the parent also commits.
- Likewise, aborting the nested transaction has no effect on the state of the parent transaction. The only result of the abort is that neither the parent nor any other transactions will see any of the container modifications performed under the protection of the nested transaction.

- If the parent transaction commits or aborts while it has active children, the child transactions are resolved in the same way as the parent. That is, if the parent aborts, then the child transactions abort as well. If the parent commits, then whatever modifications have been performed by the child transactions are also committed.
- The locks held by a nested transaction are not released when that transaction commits. Rather, they are now held by the parent transaction until such a time as that parent commits.
- Any container modifications performed by the nested transaction are not visible outside of the larger encompassing transaction until such a time as that parent transaction is committed.
- The depth of the nesting that you can achieve with nested transaction is limited only by memory.

To create a nested transaction, use the `XmlTransaction::createChild()` method. For example:

```
// parent transaction
XmlTransaction parentTxn = myManager.createTransaction();
// child transaction
XmlTransaction childTxn = parentTxn.createChild();
```

Using BDB XML Transactions with Berkeley DB Transactions

`XmlTransaction` objects are actually wrappers around Berkeley DB `DB_TXN` objects. It is therefore possible for you to use both `XmlTransaction` and `DB_TXN` handles for the same transaction simultaneously.

(`DB_TXN` objects are what Berkeley DB use to manage transactions. See the Berkeley DB C++ version of this guide for details on their usage.)

This is interesting if you want to write an application that makes use of both Berkeley DB databases and Berkeley DB XML containers. Because BDB XML uses Berkeley DB for storage and transactions, all of the Berkeley DB APIs are available to your BDB XML application.

To obtain the underlying `DB_TXN` object from an `XmlTransaction` object, use `XmlTransaction::getDB_TXN()`. You can also create an `XmlTransaction` object around an existing `DB_TXN` object by passing that object to `XmlManager::createTransaction()`.

When you use both BDB XML and Berkeley DB transaction handles for the same transaction simultaneously, there are a few things you need to keep in mind:

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

- If the parent DB_TXN object goes out scope while the XmlTransaction object is still active, then the underlying transaction is still active until such a time as the XmlTransaction object calls either commit or abort.
- If all XmlTransaction objects go out of scope and there are no in-scope DB_TXN objects, then the underlying transaction is automatically aborted.

Configuring the Transaction Subsystem

Most of the configuration activities that you need to perform for your transactional BDB XML application will involve the locking and logging subsystems. See Concurrency (page 29) and Managing BDB XML Files (page 53) for details.

However, you can also configure the maximum number of simultaneous transactions needed by your application. In general, you should not need to do this unless you use deeply nested transactions or you have many threads all of which have active transactions. In addition, you may need to configure a higher maximum number of transactions if you are using snapshot isolation. See Snapshot Isolation Transactional Requirements (page 48) for details.

By default, your application can support 20 active transactions.

You can set the maximum number of simultaneous transactions supported by your application using the DB_ENV->set_tx_max() method. Note that this method must be called before the environment has been opened.

If your application has exceeded this maximum value, then any attempt to begin a new transaction will fail.

This value can also be set using the DB_CONFIG file's set_tx_max parameter. Remember that the DB_CONFIG must reside in your environment home directory.

For example:

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

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

    DB_ENV *myEnv = NULL;
    XmlManager *myManager = NULL;
    char *envHome = "/export1/testEnv";
```

```
    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);
    }

    // Configure 40 maximum transactions.
    myEnv->set_tx_max(myEnv, 40);

    myEnv->open(myEnv, envHome, env_flags, 0);
    myManager = new XmlManager(myEnv, 0);

    // From here, you open your containers, proceed with your
    // container operations,  and respond to deadlocks as
    // is normal (omitted for brevity).

    ...
```

Chapter 4. Concurrency

BDB XML offers a great deal of support for multi-threaded and multi-process applications even when transactions are not in use. Many of BDB XML's handles are thread-safe, or can be made thread-safe by providing the appropriate flag at handle creation time, and BDB XML provides a flexible locking subsystem for managing containers in a concurrent application. Further, BDB XML provides a robust mechanism for detecting and responding to deadlocks. All of these concepts are explored in this chapter.

Before continuing, it is useful to define a few terms that will appear throughout this chapter:

- *Thread of control*

  Refers to a thread that is performing work in your application. Typically, in this book that thread will be performing BDB XML operations.

  Note that this term can also be taken to mean a separate process that is performing work — BDB XML supports multi-process operations on your containers.

  Also, BDB XML is agnostic with regard to the type or style of threads in use in your application. So if you are using multiple threads (as opposed to multiple processes) to perform concurrent database access, you are free to use whatever thread package is best for your platform and application. That said, this manual will use pthreads for its threading examples because those have the best chance of being supported across a large range of platforms.

- *Locking*

  When a thread of control obtains access to a shared resource, it is said to be *locking* that resource. Note that BDB XML supports both exclusive and non-exclusive locks. See Locks (page 32) for more information.

- *Free-threaded*

  Data structures and objects are free-threaded if they can be shared across threads of control without any explicit locking on the part of the application. Some books, libraries, and programming languages may use the term *thread-safe* for data structures or objects that have this characteristic. The two terms mean the same thing.

  For a description of free-threaded BDB XML objects, see Which BDB XML Handles are Free-Threaded (page 30).

- *Blocked*

  When a thread cannot obtain a lock because some other thread already holds a lock on that object, the lock attempt is said to be *blocked*. See Blocks (page 34) for more information.

- *Deadlock*

  Occurs when two or more threads of control attempt to access conflicting resource in such a way as none of the threads can any longer make further progress.

For example, if Thread A is blocked waiting for a resource held by Thread B, while at the same time Thread B is blocked waiting for a resource held by Thread A, then neither thread can make any forward progress. In this situation, Thread A and Thread B are said to be *deadlocked*.

For more information, see Deadlocks (page 36).

Which BDB XML Handles are Free-Threaded

The following describes to what extent and under what conditions individual handles are free-threaded.

- `DB_ENV`

  Free-threaded so long as the `DB_THREAD` flag is provided to the environment `open()` method.

- `XmlManager`

  This class free-threaded.

- `XmlTransaction`

  Access must be serialized by the application across threads of control.

- `XmlCompression`

  This class is free-threaded if the application is multi-threaded.

- `XmlContainer`

  This class is free-threaded.

- `XmlContainerConfig`

  This class is not free-threaded, and it can be safely used only by one thread of control.

- `XmlDebugListener`

  Implemented by the application, and so whether this class can be shared across multiple threads is up to your local implementation.

- `XmlDocument`

  This class is not free-threaded, and it can be safely used only by one thread of control.

- `XmlExternalFunction`

  If `XmlResolver::resolveExternalFunction()` returns a new object, then it is not free-threaded. However, if the application is multi-threaded and `resolveExternalFunction()` returns a shared instance, then it is free-threaded.

- `XmlIndexSpecification`

This class is not free-threaded, and it can be safely used only by one thread of control.

- `XmlMetaDataIterator`

  This class is not free-threaded, and it can be safely used only by one thread of control.

- `XmlQueryContext`

  This class is not free-threaded, and it can be safely used only by one thread of control at a time.

- `XmlQueryExpression`

  This class is free-threaded, and it can be safely used across multiple threads of control.

- `XmlResolver`

  If an application uses multiple threads, custom implementations of XmlResolver must be free threaded to allow multiple, simultaneous calls for resolution.

- `XmlResults`

  This class is not free-threaded, and it can be safely used only by one thread of control at a time.

- `XmlStatistics`

  This class is not free-threaded, and it can be safely used only by one thread of control at a time.

- `XmlTransaction`

  This class is not free-threaded, and it can be safely used only by one thread of control at a time.

- `XmlUpdateContext`

  This class is not free-threaded, and it can be safely used only by one thread of control at a time.

- `XmlValue`

  This class is not free-threaded, and it can be safely used only by one thread of control at a time.

Locks, Blocks, and Deadlocks

It is important to understand how locking works in a concurrent application before continuing with a description of the concurrency mechanisms BDB XML makes available to you. Blocking and deadlocking have important performance implications for your application. Consequently,

this section provides a fundamental description of these concepts, and how they affect BDB XML operations.

Locks

When one thread of control wants to obtain access to an object, it requests a *lock* for that object. This lock is what allows BDB XML to provide your application with its transactional isolation guarantees by ensuring that:

- no other thread of control can read that object (in the case of an exclusive lock), and
- no other thread of control can modify that object (in the case of an exclusive or non-exclusive lock).

Lock Resources

When locking occurs, there are conceptually three resources in use:

1. The locker.

   This is the thing that holds the lock. In a transactional application, the locker is a transaction handle. For non-transactional operations, the locker is a cursor or a Container, Database, or some document management handle.

2. The lock.

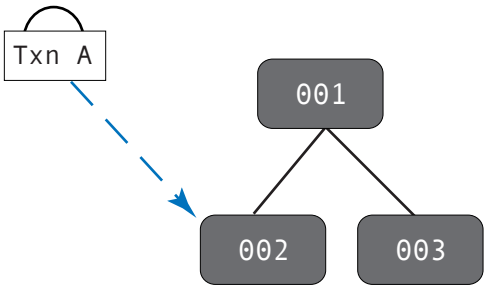
   This is the actual data structure that locks the object. In BDB XML, a locked object structure in the lock manager is representative of the object that is locked.

3. The locked object.

   The thing that your application actually wants to lock. In a BDB XML application, the locked object is usually a database page, which in turn contains multiple database entries (key and data). Note that if you use node-level containers, your documents are split into multiple database records. Therefore, when you read or write an XML document that is stored in a node container, you may be locking multiple database pages.

You can configure how many total lockers, locks, and locked objects your application is allowed to support. See Configuring the Locking Subsystem (page 38) for details.

The following figure shows a transaction handle, `Txn A`, that is holding a lock on database page `002`. In this graphic, `Txn A` is the locker, and the locked object is page `002`. Only a single lock is in use in this operation.



Types of Locks

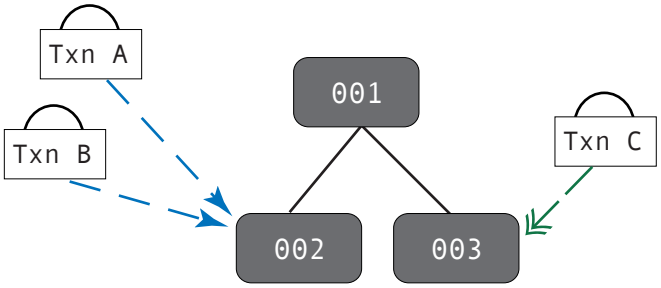
BDB XML applications support both exclusive and non-exclusive locks. *Exclusive locks* are granted when a locker wants to write to an object. For this reason, exclusive locks are also sometimes called *write locks*.

An exclusive lock prevents any other locker from obtaining any sort of a lock on the object. This provides isolation by ensuring that no other locker can observe or modify an exclusively locked object until the locker is done writing to that object.

Non-exclusive locks are granted for read-only access. For this reason, non-exclusive locks are also sometimes called *read locks*. Since multiple lockers can simultaneously hold read locks on the same object, read locks are also sometimes called *shared locks*.

A non-exclusive lock prevents any other locker from modifying the locked object while the locker is still reading the object. This is how transactional cursors are able to achieve repeatable reads; by default, the cursor's transaction holds a read lock on any object that the cursor has examined until such a time as the transaction is committed or aborted. You can avoid these read locks by using snapshot isolation. See Using Snapshot Isolation (page 47) for details.

In the following figure, `Txn A` and `Txn B` are both holding read locks on page `002`, while `Txn C` is holding a write lock on page `003`:



Lock Lifetime

A locker holds its locks until such a time as it does not need the lock any more. What this means is:

1. A transaction holds any locks that it obtains until the transaction is committed or aborted.

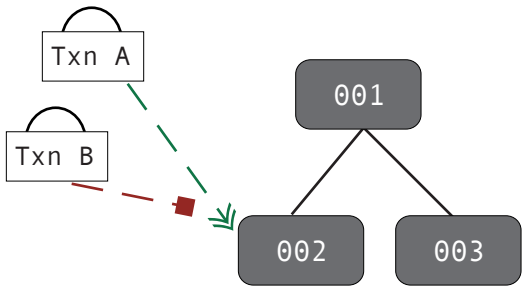
2. All non-transaction operations hold locks until such a time as the operation is completed. For cursor operations, the lock is held until the cursor is moved to a new position or closed.

Blocks

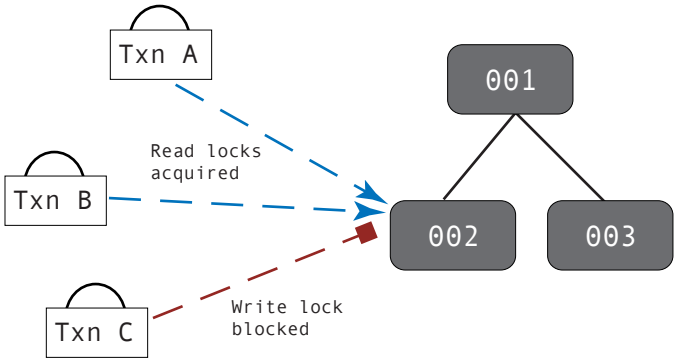
Simply put, a thread of control is blocked when it attempts to obtain a lock, but that attempt is denied because some other thread of control holds a conflicting lock. Once blocked, the thread of control is temporarily unable to make any forward progress until the requested lock is obtained or the operation requesting the lock is abandoned.

Be aware that when we talk about blocking, strictly speaking the thread is not what is attempting to obtain the lock. Rather, some object within the thread (such as a cursor) is attempting to obtain the lock. However, once a locker attempts to obtain a lock, the entire thread of control must pause until the lock request is in some way resolved.

For example, if `Txn A` holds a write lock (an exclusive lock) on object 002, then if `Txn B` tries to obtain a read *or* write lock on that object, the thread of control in which `Txn B` is running is blocked:



However, if `Txn A` only holds a read lock (a shared lock) on object `002`, then only those handles that attempt to obtain a write lock on that object will block.



Blocking and Application Performance

Multi-threaded and multi-process applications typically perform better than simple single-threaded applications because the application can perform one part of its workload (updating an XML document, for example) while it is waiting for some other lengthy operation

to complete (performing disk or network I/O, for example). This performance improvement is particularly noticeable if you use hardware that offers multiple CPUs, because the threads and processes can run simultaneously.

That said, concurrent applications can see reduced workload throughput if their threads of control are seeing a large amount of lock contention. That is, if threads are blocking on lock requests, then that represents a performance penalty for your application.

Consider once again the previous diagram of a blocked write lock request. In that diagram, `Txn C` cannot obtain its requested write lock because `Txn A` and `Txn B` are both already holding read locks on the requested object. In this case, the thread in which `Txn C` is running will pause until such a time as `Txn C` either obtains its write lock, or the operation that is requesting the lock is abandoned. The fact that `Txn C`'s thread has temporarily halted all forward progress represents a performance penalty for your application.

Moreover, any read locks that are requested while `Txn C` is waiting for its write lock will also block until such a time as `Txn C` has obtained and subsequently released its write lock.

Avoiding Blocks

Reducing lock contention is an important part of performance tuning your concurrent BDB XML application. Applications that have multiple threads of control obtaining exclusive (write) locks are prone to contention issues. Moreover, as you increase the numbers of lockers and as you increase the time that a lock is held, you increase the chances of your application seeing lock contention.

As you are designing your application, try to do the following in order to reduce lock contention:

- Reduce the length of time your application holds locks.

  Shorter lived transactions will result in shorter lock lifetimes, which will in turn help to reduce lock contention.

  In addition, by default transactional cursors hold read locks until such a time as the transaction is completed. For this reason, try to minimize the time you keep transactional cursors opened, or reduce your isolation levels – see below.

- If possible, access heavily accessed (read or write) items toward the end of the transaction. This reduces the amount of time that a heavily used page is locked by the transaction.

- Reduce your application's isolation guarantees.

  By reducing your isolation guarantees, you reduce the situations in which a lock can block another lock. Try using uncommitted reads for your read operations in order to prevent a read lock being blocked by a write lock.

  In addition, for cursors you can use degree 2 (read committed) isolation, which causes the cursor to release its read locks as soon as it is done reading the record (as opposed to holding its read locks until the transaction ends).

Be aware that reducing your isolation guarantees can have adverse consequences for your application. Before deciding to reduce your isolation, take care to examine your application's isolation requirements. For information on isolation levels, see Isolation (page 43).

- Use snapshot isolation for read-only threads.

  Snapshot isolation causes the transaction to make a copy of the page on which it is holding a lock. When a reader makes a copy of a page, write locks can still be obtained for the original page. This eliminates entirely read-write contention.

  Snapshot isolation is described in Using Snapshot Isolation (page 47).

- Consider your data access patterns.

  Depending on the nature of your application, this may be something that you can not do anything about. However, if it is possible to create your threads such that they operate only on non-overlapping portions of your database, then you can reduce lock contention because your threads will rarely (if ever) block on one another's locks.

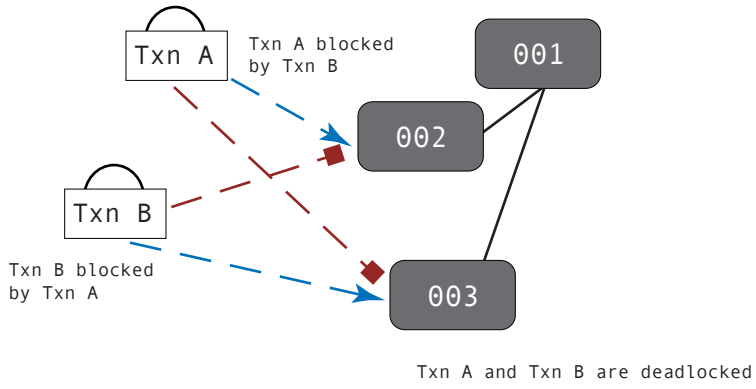
It is possible to configure BDB XML's transactions so that they never wait on blocked lock requests. Instead, if they are blocked on a lock request, they will notify the application of a deadlock (see the next section).

You configure this behavior on a transaction by transaction basis. See No Wait on Blocks (page 51) for more information.

Deadlocks

A deadlock occurs when two or more threads of control are blocked, each waiting on a resource held by the other thread. When this happens, there is no possibility of the threads ever making forward progress unless some outside agent takes action to break the deadlock.

For example, if `Txn A` is blocked by `Txn B` at the same time `Txn B` is blocked by `Txn A` then the threads of control containing `Txn A` and `Txn B` are deadlocked; neither thread can make any forward progress because neither thread will ever release the lock that is blocking the other thread.



When two threads of control deadlock, the only solution is to have a mechanism external to the two threads capable of recognizing the deadlock and notifying at least one thread that it is in a deadlock situation. Once notified, a thread of control must abandon the attempted operation in order to resolve the deadlock. BDB XML's locking subsystem offers a deadlock notification mechanism. See Configuring Deadlock Detection (page 39) for more information.

Note that when one locker in a thread of control is blocked waiting on a lock held by another locker in that same thread of the control, the thread is said to be *self-deadlocked*.

Deadlock Avoidance

The things that you do to avoid lock contention also help to reduce deadlocks (see Avoiding Blocks (page 35)). Beyond that, you can also do the following in order to avoid deadlocks:

- Never have more than one active transaction at a time in a thread. A common cause of this is for a thread to be using auto-commit for one operation while an explicit transaction is in use in that thread at the same time.
- Make sure all threads access data in the same order as all other threads. So long as threads lock database pages in the same basic order, there is no possibility of a deadlock (threads can still block, however).

  Be aware that if you are using secondary databases (indexes), it is not possible to obtain locks in a consistent order because you cannot predict the order in which locks are obtained in secondary databases. If you are writing a concurrent application and you are using secondary databases, you must be prepared to handle deadlocks.
- Declare a read/modify/write lock for those situations where you are reading a record in preparation of modifying and then writing the record. Doing this causes BDB XML to give your read operation a write lock. This means that no other thread of control can share a read lock (which might cause contention), but it also means that the writer thread will not have to wait to obtain a write lock when it is ready to write the modified data back to the database.

  For information on declaring read/modify/write locks, see Read/Modify/Write (page 50).
- Use snapshot isolation for read-only threads that operate concurrently with writer threads. This will avoid read-write contention for your writer threads.

  For information on snapshot isolation, see Using Snapshot Isolation (page 47).

The Locking Subsystem

In order to allow concurrent operations, BDB XML provides the locking subsystem. This subsystem provides inter- and intra- process concurrency mechanisms. It is extensively used by BDB XML concurrent applications, but it can also be generally used for non-BDB XML resources.

This section describes the locking subsystem as it is used to protect BDB XML resources. In particular, issues on configuration are examined here. For information on using the locking subsystem to manage non-BDB XML resources, see the *Berkeley DB Programmer's Reference Guide*.

Configuring the Locking Subsystem

You initialize the locking subsystem by specifying `DB_INIT_LOCK` to the `DB_ENV->open()` method.

Before opening your environment, you can configure various maximum values for your locking subsystem. Note that these limits can only be configured before the environment is opened. Also, these methods configure the entire environment, not just a specific environment handle.

Finally, each bullet below identifies the `DB_CONFIG` file parameter that can be used to specify the specific locking limit. If used, these `DB_CONFIG` file parameters override any value that you might specify using the environment handle.

The limits that you can configure are as follows:

- The maximum number of lockers supported by the environment. This value is used by the environment when it is opened to estimate the amount of space that it should allocate for various internal data structures. By default, 1,000 lockers are supported.

  To configure this value, use the `DB_ENV->set_lk_max_lockers()` method.

  As an alternative to this method, you can configure this value using the `DB_CONFIG` file's `set_lk_max_lockers` parameter.

- The maximum number of locks supported by the environment. By default, 1,000 locks are supported.

  To configure this value, use the `DB_ENV->set_lk_max_locks()` method.

  As an alternative to this method, you can configure this value using the `DB_CONFIG` file's `set_lk_max_locks` parameter.

- The maximum number of locked objects supported by the environment. By default, 1,000 objects can be locked.

  To configure this value, use the `DB_ENV->set_lk_max_objects()` method.

  As an alternative to this method, you can configure this value using the `DB_CONFIG` file's `set_lk_max_objects` parameter.

For a definition of lockers, locks, and locked objects, see Lock Resources (page 32).

For example, to configure the maximum number of locks that your environment can use:

```
#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_THREAD     |  // Free-thread the env handle
                            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);
    }

    // Configure max locks
    myEnv->set_lk_max_locks(envp, 5000);

    myEnv->open(myEnv, envHome, env_flags, 0);
    myManager = new XmlManager(myEnv, 0);

    // Do work here. Clean up when all done.

    if (myManager != NULL) {
        delete myManager;
    }
    myEnv->close(myEnv, 0);

    return (EXIT_SUCCESS);
}
```

Configuring Deadlock Detection

In order for BDB XML to know that a deadlock has occurred, some mechanism must be used to perform deadlock detection. There are three ways that deadlock detection can occur:

1. Allow BDB XML to internally detect deadlocks as they occur.

   To do this, you use `DB_ENV->set_lk_detect()`. This method causes BDB XML to walk its internal lock table looking for a deadlock whenever a lock request is blocked. This method also identifies how BDB XML decides which lock requests are rejected when deadlocks are detected. For example, BDB XML can decide to reject the lock request for the transaction that has the most number of locks, the least number of locks, holds the oldest lock, holds

the most number of write locks, and so forth (see the API reference documentation for a complete list of the lock detection policies).

You can call this method at any time during your application's lifetime, but typically it is used before you open your environment.

Note that how you want BDB XML to decide which thread of control should break a deadlock is extremely dependent on the nature of your application. It is not unusual for some performance testing to be required in order to make this determination. That said, a transaction that is holding the maximum number of locks is usually indicative of the transaction that has performed the most amount of work. Frequently you will not want a transaction that has performed a lot of work to abandon its efforts and start all over again. It is not therefore uncommon for application developers to initially select the transaction with the *minimum* number of write locks to break the deadlock.

Using this mechanism for deadlock detection means that your application will never have to wait on a lock before discovering that a deadlock has occurred. However, walking the lock table every time a lock request is blocked can be expensive from a performance perspective.

2. Use a dedicated thread or external process to perform deadlock detection. Note that this thread must be performing no other container operations beyond deadlock detection.

   To externally perform lock detection, you can use either the `DB_ENV->lock_detect()` method, or use the **db_deadlock** command line utility. This method (or command) causes BDB XML to walk the lock table looking for deadlocks.

   Note that like `DB_ENV->set_lk_detect()`, you also use this method (or command line utility) to identify which lock requests are rejected in the event that a deadlock is detected.

   Applications that perform deadlock detection in this way typically run deadlock detection between every few seconds and a minute. This means that your application may have to wait to be notified of a deadlock, but you also save the overhead of walking the lock table every time a lock request is blocked.

3. Lock timeouts.

   You can configure your locking subsystem such that it times out any lock that is not released within a specified amount of time. To do this, use the `DB_ENV->set_timeout()` method. Note that lock timeouts are only checked when a lock request is blocked or when deadlock detection is otherwise performed. Therefore, a lock can have timed out and still be held for some length of time until BDB XML has a reason to examine its locking tables.

   Be aware that extremely long-lived transactions, or operations that hold locks for a long time, may be inappropriately timed out before the transaction or operation has a chance to complete. You should therefore use this mechanism only if you know your application will hold locks for very short periods of time.

For example, to configure your application such that BDB XML checks the lock table for deadlocks every time a lock request is blocked:

```
#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_THREAD     |  // Free-thread the env handle
                          DB_INIT_TXN;     // Initialize transactions

    DB_ENV *myEnv = NULL;
    XmlManager *myManager = NULL;
    char *envHome = "/export1/testEnv";
    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);
    }

    // Configure db to perform deadlock detection internally, and to
    // choose the transaction that has performed the least amount
    // of writing to break the deadlock in the event that one
    // is detected.
    myEnv->set_lk_detect(DB_LOCK_MINWRITE);

    myEnv->open(myEnv, envHome, env_flags, 0);
    myManager = new XmlManager(myEnv, 0);

    // From here, you open your containers, proceed with your
    // container operations,  and respond to deadlocks as
    // is normal (omitted for brevity).

    ...
```

Finally, the following command line call causes deadlock detection to be run against the environment contained in `/export/dbenv`. The transaction with the youngest lock is chosen to break the deadlock:

```
> /usr/local/db_install/bin/db_deadlock -h /export/dbenv -a y
```

For more information, see the db_deadlock reference documentation. [http://www.oracle.com/technology/documentation/berkeley-db/db/utility/db_deadlock.html]

Resolving Deadlocks

When BDB XML determines that a deadlock has occurred, it will select a thread of control to resolve the deadlock and then throws XmlException in that thread. You must then test this exception to see if it is caused by a deadlock situation. Do this by checking whether XmlException.getDbErrno() is equal to DB_LOCK_DEADLOCK. If a deadlock is detected, the thread must:

1. Cease all read and write operations.
2. Abort the transaction.
3. Optionally retry the operation. If your application retries deadlocked operations, the new attempt must be made using a new transaction.

If a thread has deadlocked, it may not make any additional container calls using the handle that has deadlocked.

For example:

```
// Environment, Manager and Container opens are omitted for brevity

XmlTransaction txn;

// retry_count is a counter used to identify how many times we've
// retried this operation. To avoid the potential
// for endless looping, we won't retry more than
// MAX_DEADLOCK_RETRIES times.

while (retry_count < MAX_DEADLOCK_RETRIES) {
    try {
        txn = myManager.createTransaction();

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

        // Get the first input stream.
        XmlInputStream *theStream =
            myManager.createLocalFileInputStream("/export/file.xml");

        // Put the first document
        myContainer.putDocument(txn,         // the transaction object
                                "file.xml",  // The document's name
                                theStream,   // The actual document.
                                theContext,  // The update context
                                             // (required).
                                0);
```

```
            txn.commit();
            return (EXIT_SUCCESS);
        } catch (XmlException &e) {
            if (e.getDbErrno() == DB_LOCK_DEADLOCK) {
                try {
                    // Abort the transaction and increment the
                    // retry counter
                    txn.abort();
                    // Increment the retry count
                    retry_count++;
                    // If we've retried too many times, log it
                    // and exit
                    if (retry_count >= MAX_DEADLOCK_RETRIES) {
                        std::cerr << "Exceeded retry limit. Giving up."
                                  << std::endl;
                        return (EXIT_FAILURE);
                    }
                } catch (DbException &ae) {
                    envp->err(ae.getErrorCode(), "txn abort failed.");
                    return (EXIT_FAILURE);
                }
            } else {
                try {
                    // For a generic error, log it and abort.
                    std::cerr << "Error in transaction: "
                              << e.what() << "\n"
                              << "Aborting." << std::endl;
                    txn.abort();
                } catch (DbException &ae) {
                    envp->err(ae.get_errno(), "txn abort failed.");
                    return (EXIT_FAILURE);
                }
            }
        }
    }
```

Isolation

Isolation guarantees are an important aspect of transactional protection. Transactions ensure the data your transaction is working with will not be changed by some other transaction. Moreover, the modifications made by a transaction will never be viewable outside of that transaction until the changes have been committed.

That said, there are different degrees of isolation, and you can choose to relax your isolation guarantees to one degree or another depending on your application's requirements. The primary reason why you might want to do this is because of performance; the more isolation you ask your transactions to provide, the more locking that your application must do. With more locking comes a greater chance of blocking, which in turn causes your threads to pause while waiting for a lock. Therefore, by relaxing your isolation guarantees, you can *potentially* improve your

application's throughput. Whether you actually see any improvement depends, of course, on the nature of your application's data and transactions.

Supported Degrees of Isolation

BDB XML supports the following levels of isolation:

Degree	ANSI Term	Definition
1	READ UNCOMMITTED	Uncommitted reads means that one transaction will never overwrite another transaction's dirty data. Dirty data is data that a transaction has modified but not yet committed to the underlying data store. However, uncommitted reads allows a transaction to see data dirtied by another transaction. In addition, a transaction may read data dirtied by another transaction, but which subsequently is aborted by that other transaction. In this latter case, the reading transaction may be reading data that never really existed in the container.
2	READ COMMITTED	Committed read isolation means that degree 1 is observed, except that dirty data is never read. In addition, this isolation level guarantees that data will never change so long as it is addressed by the cursor, but the data may change before the reading cursor is closed. In the case of a transaction, data at the current cursor position will not change, but once the cursor moves, the previous referenced data can change. This means that readers release read locks before the cursor is closed, and therefore, before the transaction completes. Note that this level of isolation causes the cursor to operate in exactly the same way as it does in the absence of a transaction.
3	SERIALIZABLE	Committed read is observed, plus the data read by a transaction, T, will never be dirtied by another transaction before T completes. This means that both read and write locks are not released until the transaction completes. In addition, no transactions will see phantoms. Phantoms are records returned as a result of a search, but which were not seen by the same transaction when the identical search criteria was previously used. This is BDB XML's default isolation guarantee.

By default, BDB XML transactions and transactional cursors offer serializable isolation. You can optionally reduce your isolation level by configuring BDB XML to use uncommitted read isolation. See Reading Uncommitted Data (page 45) for more information. You can also configure BDB XML to use committed read isolation. See Committed Reads (page 45) for more information.

Finally, in addition to BDB XML's normal degrees of isolation, you can also use *snapshot isolation*. This allows you to avoid the read locks that serializable isolation requires. See Using Snapshot Isolation (page 47) for details.

Reading Uncommitted Data

Berkeley DB allows you to configure your application to read data that has been modified but not yet committed by another transaction; that is, dirty data. When you do this, you may see a performance benefit by allowing your application to not have to block waiting for write locks. On the other hand, the data that your application is reading may change before the transaction has completed.

That said, configuring BDB XML to read uncommitted data can result in internal inconsistencies which lead to random errors. For best results, you should avoid configuring your BDB XML transactions to read uncommitted data.

Committed Reads

Committed Reads control the behavior of a Berkeley DB mechanism called a *cursor*. Cursors are not something you would normally be using directly with your BDB XML application, unless you are using Berkeley DB databases alongside of your BDB XML containers. For that reason, this section is potentially not of great interest to you. Still, we present it here for the sake of completeness.

For a thorough description of cursors, see the *Getting Started with Berkeley DB* guide.

You can configure your transaction so that the data being read by a transactional cursor is consistent so long as it is being addressed by the cursor. However, once the cursor is done reading the record (that is, reading records from the page that it currently has locked), the cursor releases its lock on that record or page. This means that the data the cursor has read and released may change before the cursor's transaction has completed.

For example, suppose you have two transactions, `Ta` and `Tb`. Suppose further that `Ta` has a cursor that reads `record R`, but does not modify it. Normally, `Tb` would then be unable to write `record R` because `Ta` would be holding a read lock on it. But when you configure your transaction for committed reads, `Tb` *can* modify `record R` before `Ta` completes, so long as the reading cursor is no longer addressing the record or page.

When you configure your application for this level of isolation, you may see better performance throughput because there are fewer read locks being held by your transactions. Read committed isolation is most useful when you have a cursor that is reading and/or writing records in a single direction, and that does not ever have to go back to re-read those same records. In this case, you can allow BDB XML to release read locks as it goes, rather than hold them for the life of the transaction.

To configure your application to use committed reads, do one of the following:

- Create your transaction such that it allows committed reads. You do this by specifying `DB_READ_COMMITTED` when you open the transaction.
- Specify `DB_READ_COMMITTED` when you open the cursor.

For example, the following creates a transaction that allows committed reads:

```
#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_THREAD     |  // Free-thread the env handle
                          DB_INIT_TXN;     // Initialize transactions

    DB_ENV *myEnv = NULL;
    XmlManager *myManager = NULL;
    char *envHome = "/export1/testEnv";
    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);
    myManager = new XmlManager(myEnv, 0);

    try {
        // Notice that we do not have to specify any options
        // to allow committed reads (this is as opposed
        // to uncommitted reads where we DO have to specify
        // options).

        XmlContainerConfig cconfig;
        cconfig.setAllowCreate(true);    // If the container does not
                                         // exist, create it.
        cconfig.setTransactional(true); // Enable transactions.
```

```
        std::string containerName = "myContainer.dbxml";
        XmlContainer myContainer =
            myManager->openContainer(containerName, cconfig);

    } catch(XmlException &e) {
        std::cerr << "Error opening container: " << std::endl;
        std::cerr << e.what() << std::endl;
        return (EXIT_FAILURE);
    } catch(std::exception &e) {
        std::cerr << "Error opening container: " << std::endl;
        std::cerr << e.what() << std::endl;
        return (EXIT_FAILURE);
    }

    // File to open
    std::string file1 = "doc1.xml";

    // Transaction handle
    // Open the transaction and enable committed reads.  All
    // queries performed with this transaction handle will
    // use read committed isolation.
    XmlTransaction txn = myManager.createTransaction(DB_READ_COMMITTED);

    // From here, you perform your container reads and writes as normal,
    // committing and aborting the transactions as is necessary,  as well as
    // testing for deadlock exceptions as normal (omitted for brevity).

    ...
```

Using Snapshot Isolation

By default BDB XML uses serializable isolation. An important side effect of this isolation level is that read operations obtain read locks on database pages, and then hold those locks until the read operation is completed.

You can avoid this by using snapshot isolation. Snapshot isolation uses *multiversion concurrency control* to guarantee repeatable reads. What this means is that every time a writer would take a read lock on a page, instead a copy of the page is made and the writer operates on that page copy. This frees other writers from blocking due to a read lock held on the page.

Snapshot isolation is strongly recommended for read-only threads when writer threads are also running, as this will eliminate read-write contention and greatly improve transaction throughput for your writer threads. However, in order for snapshot isolation to work for your reader-only threads, you must of course use transactions for your BDB XML reads.

Snapshot Isolation Cost

Snapshot isolation does not come without a cost. Because pages are being duplicated before being operated upon, the cache will fill up faster. This means that you might need a larger cache in order to hold the entire working set in memory.

If the cache becomes full of page copies before old copies can be discarded, additional I/O will occur as pages are written to temporary "freezer" files on disk. This can substantially reduce throughput, and should be avoided if possible by configuring a large cache and keeping snapshot isolation transactions short.

You can estimate how large your cache should be by taking a checkpoint, followed by a call to the `DB_ENV->log_archive()` method. The amount of cache required is approximately double the size of the remaining log files (that is, the log files that cannot be archived).

Snapshot Isolation Transactional Requirements

In addition to an increased cache size, you may also need to increase the maximum number of transactions that your application supports. (See Configuring the Transaction Subsystem (page 27) for details on how to set this.) In the worst case scenario, you might need to configure your application for one more transaction for every page in the cache. This is because transactions are retained until the last page they created is evicted from the cache.

When to Use Snapshot Isolation

Snapshot isolation is best used when all or most of the following conditions are true:

- You can have a large cache relative to your working data set size.
- You require repeatable reads.
- You will be using transactions that routinely work on the entire database, or more commonly, there is data in your database that will be very frequently written by more than one transaction.
- If your application uses a single write thread and multiple readers, then snapshot isolation can help performance. However, if your application uses multiple write threads, then snapshot isolation can result in additional deadlocks that may harm your application's performance.

How to use Snapshot Isolation

You use snapshot isolation by:

- Opening the container with multiversion support. You can configure this either when you open your environment or when you open your container. Use the `DB_MULTIVERSION` flag to configure this support. Use the `XmlContainerConfig::setMultiversion()` option to configure this support when you open your container. To configure multiversion support when you open your environment, use the `DB_MULTIVERSION` flag on the environment open.
- Configure your transaction to use snapshot isolation.

To do this, pass the DB_TXN_SNAPSHOT flag when you create the transaction.

The simplest way to take advantage of snapshot isolation is for queries: keep update transactions using full read/write locking and use snapshot isolation on read-only transactions or cursors. This should minimize blocking of snapshot isolation transactions and will avoid deadlock errors.

If the application has update transactions which read many items and only update a small set (for example, scanning until a desired record is found, then modifying it), throughput may be improved by running some updates at snapshot isolation as well. But doing this means that you must manage deadlock errors. See Resolving Deadlocks (page 42) for details.

The following code fragment turns on snapshot isolation for a transaction:

```
#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
                          DB_MULTIVERSION; // Support snapshot isolation.

    DB_ENV *myEnv = NULL;
    XmlManager *myManager = NULL;
    char *envHome = "/export1/testEnv";
    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);
    myManager = new XmlManager(myEnv, 0);

    try {
        // Note that no special options are required here for snapshot
        // isolation. This is because it is already enabled at the
        // environment level.
        XmlContainerConfig cconfig;
        cconfig.setAllowCreate(true);   // If the container does not
```

```
                                        // exits, create it.
        cconfig.setTransactional(true); // Enable transactions

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

    ...

    // Transaction handle
    XmlTransaction txn = myManager.createTransaction(DB_TXN_SNAPSHOT);

    // Remainder of program omitted for brevity.

    ...
```

Read/Modify/Write

If you are retrieving a document from the container for the purpose of modifying or deleting it, you should declare a read-modify-write cycle at the time that you read the document. Doing so causes BDB XML to obtain write locks (instead of a read locks) at the time of the read. This helps to prevent deadlocks by preventing another transaction from acquiring a read lock on the same record while the read-modify-write cycle is in progress.

Note that declaring a read-modify-write cycle may actually increase the amount of blocking that your application sees, because readers immediately obtain write locks and write locks cannot be shared. For this reason, you should use read-modify-write cycles only if you are seeing a large amount of deadlocking occurring in your application.

In order to declare a read/modify/write cycle when you perform a read operation, pass the `DB_RMW` flag to the `XmlQueryExpression::execute()` or `XmlManager::query()` method.

For example:

```
XmlTransaction txn;
try {
    txn = myManager.createTransaction();

    // 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. Declare a read/modify/write cycle
    XmlResults results = myManager.query(txn, myQuery, context, DB_RMW);
```

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

    txn.commit();
} catch (XmlException &e) {
    // Perform exception handling as is normal
}
return (EXIT_SUCCESS);
```

No Wait on Blocks

Normally when a BDB XML transaction is blocked on a lock request, it must wait until the requested lock becomes available before its thread-of-control can proceed. However, it is possible to configure a transaction handle such that it will report a deadlock rather than wait for the block to clear.

You do this on a transaction by transaction basis by specifying `DB_TXN_NOWAIT` to the `XmlManager::createTransaction()` method.

For example:

```
XmlTransaction txn = NULL;
try {
    txn = myManager.createTransaction(DB_TXN_NOWAIT);
    ...
} catch (XmlException &e) {
        // Deadlock detection and exception handling omitted
        // for brevity
```

Explicit Transactions on Reads

Concurrent BDB XML applications should take care to use explicit transactions for read-only activity in the container, as well as write-only and read/write container activity. You *should not* perform reads without the benefit of an explicit transaction handle in BDB XML if you are using multiple threads to access the container, and you are using transactions in the first place.

The reason for this is that read operations take locks and can still deadlock. Therefore, non-transactional reads that occur concurrently with transactional operations (read or write) will lead to hangs and/or unexpected exceptions.

In addition, your threads that perform container access should always perform deadlock handling, even if the thread only ever performs read-only access. The exception to this rule for read-only threads is if you are using snapshot isolation. In that case, your application should not see deadlocks.

For information on handling deadlocks, see The Locking Subsystem (page 37). For information on using snapshot isolation, see Using Snapshot Isolation (page 47).

Chapter 5. Managing BDB XML Files

BDB XML is capable of storing several types of files on disk:

- Data files, which contain the actual data in your container.
- Log files, which contain information required to recover your database in the event of a system or application failure.
- Region files, which contain information necessary for the overall operation of your application.

Of these, you must manage your data and log files by ensuring that they are backed up. You should also pay attention to the amount of disk space your log files are consuming, and periodically remove any unneeded files. Finally, you can optionally tune your logging subsystem to best suit your application's needs and requirements. These topics are discussed in this chapter.

Checkpoints

Before we can discuss BDB XML file management, we need to describe checkpoints. When containers are modified (that is, a transaction is committed), the modifications are recorded in BDB XML's logs, but they are *not* necessarily reflected in the actual container files on disk.

This means that as time goes on, increasingly more data is contained in your log files that is not contained in your data files. As a result, you must keep more log files around than you might actually need. Also, any recovery run from your log files will take increasingly longer amounts of time, because there is more data in the log files that must be reflected back into the data files during the recovery process.

You can reduce these problems by periodically running a checkpoint against your environment. The checkpoint:

- Flushes dirty pages from the in-memory cache. This means that data modifications found in your in-memory cache are written to the container files on disk. Note that a checkpoint also causes data dirtied by an uncommitted transaction to also be written to your container files on disk. In this latter case, BDB XML's normal recovery is used to remove any such modifications that were subsequently abandoned by your application using a transaction abort.

  Normal recovery is describe in Recovery Procedures (page 58).
- Writes a checkpoint record.
- Flushes the log. This causes all log data that has not yet been written to disk to be written.
- Writes a list of open containers.

There are several ways to run a checkpoint. One way is to use the **db_checkpoint** command line utility. (Note, however, that this command line utility cannot be used if your environment was opened using `DB_PRIVATE`.)

You can also run a thread that periodically checkpoints your environment for you by calling the `DB_ENV->txn_checkpoint()` method.

Note that you can prevent a checkpoint from occurring unless more than a specified amount of log data has been written since the last checkpoint. You can also prevent the checkpoint from running unless more than a specified amount of time has occurred since the last checkpoint. These conditions are particularly interesting if you have multiple threads or processes running checkpoints.

For configuration information, see the DB_ENV->txn_checkpoint() API reference page. [http://www.oracle.com/technology/documentation/berkeley-db/db/api_c/txn_checkpoint.html]

Note that running checkpoints can be quite expensive. BDB XML must flush every dirty page to the backing container files. On the other hand, if you do not run checkpoints often enough, your recovery time can be unnecessarily long and you may be using more disk space than you really need. Also, you cannot remove log files until a checkpoint is run. Therefore, deciding how frequently to run a checkpoint is one of the most common tuning activity for BDB XML applications.

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

using namespace DbXml;

void *checkpoint_thread(void *);

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

    DB_ENV *myEnv = 0;
    XmlManager *myManager = NULL;
    char *envHome = "/export1/testEnv";
    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);
```

```
    myManager = new XmlManager(myEnv, 0);

    try {
        // Start a checkpoint thread.
        pthread_t ptid;
        int ret;
        if ((ret = pthread_create(
            &ptid, NULL, checkpoint_thread, (void *)myEnv)) != 0) {
                fprintf(stderr,
                    "txnapp: failed spawning checkpoint thread: %s\n",
                    strerror(errno));

                if (myManager != NULL) {
                    delete myManager;
                }
                myEnv->close(myEnv, 0);
                exit (1);
        }

        // All other threads and application shutdown code
        // omitted for brevity.

        ...
}

void *
checkpoint_thread(void *arg) {
    DB_ENV *dbenv = arg;

    // Checkpoint once a minute.
    for (;; sleep(60)) {
        try {
            dbenv->txn_checkpoint(dbenv, 0, 0, 0));
        } catch(DbException &e) {
            dbenv->err(dbenv, e.get_errno(), "checkpoint thread");
            exit (e.get_errno());
        }
    }

    // NOTREACHED
}
```

Backup Procedures

Durability is an important part of your transactional guarantees. It means that once a transaction has been successfully committed, your application will always see the results of that transaction.

Of course, no software algorithm can guarantee durability in the face of physical data loss. Hard drives can fail, and if you have not copied your data to locations other than your primary

disk drives, then you will lose data when those drives fail. Therefore, in order to truly obtain a durability guarantee, you need to ensure that any data stored on disk is backed up to secondary or alternative storage, such as secondary disk drives, or offline tapes.

There are three different types of backups that you can perform with BDB XML containers and log files. They are:

- Offline backups

  This type of backup is perhaps the easiest to perform as it involves simply copying database and log files to an offline storage area. It also gives you a snapshot of the database at a fixed, known point in time. However, you cannot perform this type of a backup while you are performing writes to the database.

- Hot backups

  This type of backup gives you a snapshot of your database. Since your application can be writing to the database at the time that the snapshot is being taken, you do not necessarily know what the exact state of the database is for that given snapshot.

- Incremental backups

  This type of backup refreshes a previously performed backup.

Once you have performed a backup, you can perform *catastrophic recovery* to restore your containers from the backup. See Catastrophic Recovery (page 60) for more information.

Note that you can also maintain a hot failover. See Using Hot Failovers (page 65) for more information.

About Unix Copy Utilities

If you are copying database files you must copy databases atomically, in multiples of the database page size. In other words, the reads made by the copy program must not be interleaved with writes by other threads of control, and the copy program must read the databases in multiples of the underlying database page size. Generally, this is not a problem because operating systems already make this guarantee and system utilities normally read in power-of-2 sized chunks, which are larger than the largest possible Berkeley DB database page size.

On some platforms (most notably, some releases of Solaris), the copy utility (`cp`) was implemented using the `mmap()` system call rather than the `read()` system call. Because `mmap()` did not make the same guarantee of read atomicity as did `read()`, the `cp` utility could create corrupted copies of the containers.

Also, some platforms have implementations of the `tar` utility that performs 10KB block reads by default. Even when an output block size is specified, the utility will still not read the underlying containers in multiples of the specified block size. Again, the result can be a corrupted backup.

To fix these problems, use the `dd` utility instead of `cp` or `tar`. When you use `dd`, make sure you specify a block size that is equal to, or an even multiple of, your container page size. Finally,

if you plan to use a system utility to copy container files, you may want to use a system call trace utility (for example, `ktrace` or `truss`) to make sure you are not using a I/O size that is smaller than your container page size. You can also use these utilities to make sure the system utility is not using a system call other than `read()`.

Offline Backups

To create an offline backup:

1. Commit or abort all on-going transactions.
2. Pause all database writes.
3. Force a checkpoint. See Checkpoints (page 53) for details.
4. Copy all your container files to the backup location.

   However, be aware that backing up just the modified databases only works if you have all of your log files. If you have been removing log files for any reason then using can result in an unrecoverable backup because you might not be notified of a database file that was modified.
5. Copy the *last* log file to your backup location. Your log files are named `log.`*xxxxxxxxxx*, where *xxxxxxxxxx* is a sequential number. The last log file is the file with the highest number.

Hot Backup

To create a hot backup, you do not have to stop database operations. Transactions may be on-going and you can be writing to your database at the time of the backup. However, this means that you do not know exactly what the state of your database is at the time of the backup.

You can use the **db_hotbackup** command line utility to create a hot backup for you. This utility will (optionally) run a checkpoint and the copy all necessary files to a target directory.

Alternatively, you can manually create a hot backup as follows:

1. Copy all your container files to the backup location.
2. Copy all logs to your backup location.

It is important to copy your container files *and then* your logs. In this way, you can complete or roll back any container operations that were only partially completed when you copied the databases.

Incremental Backups

Once you have created a full backup (that is, either a offline or hot backup), you can create incremental backups. To do this, simply copy all of your currently existing log files to your backup location.

Incremental backups do not require you to run a checkpoint or to cease container write operations.

When you are working with incremental backups, remember that the greater the number of log files contained in your backup, the longer recovery will take. You should run full backups on some interval, and then do incremental backups on a shorter interval. How frequently you need to run a full backup is determined by the rate at which your containers change and how sensitive your application is to lengthy recoveries (should one be required).

You can also shorten recovery time by running recovery against the backup as you take each incremental backup. Running recovery as you go means that there will be less work for BDB XML to do if you should ever need to restore your environment from the backup.

Recovery Procedures

BDB XML supports two types of recovery:

- Normal recovery, which is run when your environment is opened upon application startup, examines only those log records needed to bring the containers to a consistent state since the last checkpoint. Normal recovery starts with any logs used by any transactions active at the time of the last checkpoint, and examines all logs from then to the current logs.
- Catastrophic recovery, which is performed in the same way that normal recovery is except that it examines all available log files. You use catastrophic recovery to restore your containers from a previously created backup.

Of these two, normal recovery should be considered a routine matter; in fact you should run normal recovery whenever you start up your application.

Catastrophic recovery is run whenever you have lost or corrupted your container files and you want to restore from a backup. You also run catastrophic recovery when you create a hot backup (see Using Hot Failovers (page 65) for more information).

Normal Recovery

Normal recovery examines the contents of your environment's log files, and uses this information to ensure that your container files are consistent relative to the information contained in the log files.

Normal recovery also recreates your environment's region files. This has the desired effect of clearing any unreleased locks that your application may have held at the time of an unclean application shutdown.

Normal recovery is run only against those log files created since the time of your last checkpoint. For this reason, your recovery time is dependent on how much data has been written since the last checkpoint, and therefore on how much log file information there is to examine. If you run checkpoints infrequently, then normal recovery can take a relatively long time.

You should run normal recovery every time you perform application startup.

To run normal recovery:

- Make sure all your environment handles are closed.
- Normal recovery *must be* single-threaded.
- Provide the DB_RECOVER flag when you open your environment.

You can also run recovery by pausing or shutting down your application and using the **db_recover** command line utility.

For example:

```
#include "DbXml.hpp"

...

void *checkpoint_thread(void *);

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
                          DB_THREAD     |  // Free-thread the env handle
                          DB_RECOVER;      // Run normal recovery

    DB_ENV *myEnv = 0;
    XmlManager *myManager = NULL;
    char *envHome = "/export1/testEnv";
    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);
    myManager = new XmlManager(myEnv, 0);

    ...
```

```
    // All other operations are identical from here. Notice, however,
    // that we have not created any other threads of control before
    // recovery is complete. You want to run recovery for
    // the first thread in your application that opens an environment,
    // but not for any subsequent threads.
```

Catastrophic Recovery

Use catastrophic recovery when you are recovering your containers from a previously created backup. Note that to restore your containers from a previous backup, you should copy the backup to a new environment directory, and then run catastrophic recovery. Failure to do so can lead to the internal database structures being out of sync with your log files.

Catastrophic recovery must be run single-threaded.

To run catastrophic recovery:

- Shutdown all container operations.
- Restore the backup to an empty directory.
- Provide the `DB_RECOVER_FATAL` flag when you open your environment. This environment open must be single-threaded.

You can also run recovery by pausing or shutting down your application and using the **db_recover** command line utility with the the `-c` option.

Note that catastrophic recovery examines every available log file — not just those log files created since the last checkpoint as is the case for normal recovery. For this reason, catastrophic recovery is likely to take longer than does normal recovery.

For example:

```
#include "DbXml.hpp"

...

void *checkpoint_thread(void *);

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
                          DB_THREAD     |  // Free-thread the env handle
                          DB_RECOVER_FATAL;   // Run catastrophic recovery
```

```
DB_ENV *myEnv = 0;
XmlManager *myManager = NULL;
char *envHome = "/export1/testEnv";
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);
myManager = new XmlManager(myEnv, 0);

...
```

Designing Your Application for Recovery

When building your BDB XML application, you should consider how you will run recovery. If you are building a single threaded, single process application, it is fairly simple to run recovery when your application first opens its environment. In this case, you need only decide if you want to run recovery every time you open your application (recommended) or only some of the time, presumably triggered by a start up option controlled by your application's user.

However, for multi-threaded and multi-process applications, you need to carefully consider how you will design your application's startup code so as to run recovery only when it makes sense to do so.

Recovery for Multi-Threaded Applications

If your application uses only one environment handle, then handling recovery for a multi-threaded application is no more difficult than for a single threaded application. You simply open the environment in the application's main thread, and then pass that handle to each of the threads that will be performing BDB XML operations. We illustrate this with our final example in this book (see Transaction Example (page 73) for more information).

Alternatively, you can have each worker thread open its own environment handle. However, in this case, designing for recovery is a bit more complicated.

Generally, when a thread performing container operations fails or hangs, it is frequently best to simply restart the application and run recovery upon application startup as normal. However, not all applications can afford to restart because a single thread has misbehaved.

If you are attempting to continue operations in the face of a misbehaving thread, then at a minimum recovery must be run if a thread performing container operations fails or hangs.

Remember that recovery clears the environment of all outstanding locks, including any that might be outstanding from an aborted thread. If these locks are not cleared, other threads performing database operations can back up behind the locks obtained but never cleared by the failed thread. The result will be an application that hangs indefinitely.

To run recovery under these circumstances:

1. Suspend or shutdown all other threads performing database operations.

2. Discarding any open environment handles. Note that attempting to gracefully close these handles may be asking for trouble; the close can fail if the environment is already in need of recovery. For this reason, it is best and easiest to simply discard the handle.

3. Open new handles, running recovery as you open them. See Normal Recovery (page 58) for more information.

4. Restart all your container threads.

A traditional way to handle this activity is to spawn a watcher thread that is responsible for making sure all is well with your threads, and performing the above actions if not.

However, in the case where each worker thread opens and maintains its own environment handle, recovery is complicated for two reasons:

1. For some applications and workloads, it might be worthwhile to give your database threads the ability to gracefully finalize any on-going transactions. If this is the case, your code must be capable of signaling each thread to halt BDB XML activities and close its environment. If you simply run recovery against the environment, your container threads will detect this and fail in the midst of performing their container operations.

2. Your code must be capable of ensuring only one thread runs recovery before allowing all other threads to open their respective environment handles. Recovery should be single threaded because when recovery is run against an environment, it is deleted and then recreated. This will cause all other processes and threads to "fail" when they attempt operations against the newly recovered environment. If all threads run recovery when they start up, then it is likely that some threads will fail because the environment that they are using has been recovered. This will cause the thread to have to re-execute its own recovery path. At best, this is inefficient and at worst it could cause your application to fall into an endless recovery pattern.

Recovery in Multi-Process Applications

Frequently, BDB XML applications use multiple processes to interact with the containers. For example, you may have a long-running process, such as some kind of server, and then a series of administrative tools that you use to inspect and administer the underlying containers. Or, in some web-based architectures, different services are run as independent processes that are managed by the server.

In any case, recovery for a multi-process environment is complicated for two reasons:

1. In the event that recovery must be run, you might want to notify processes interacting with the environment that recovery is about to occur and give them a chance to gracefully terminate. Whether it is worthwhile for you to do this is entirely dependent upon the nature of your application. Some long-running applications with multiple processes performing meaningful work might want to do this. Other applications with processes performing container operations that are likely to be harmed by error conditions in other processes will likely find it to be not worth the effort. For this latter group, the chances of performing a graceful shutdown may be low anyway.

2. Unlike single process scenarios, it can quickly become wasteful for every process interacting with the containers to run recovery when it starts up. This is partly because recovery *does* take some amount of time to run, but mostly you want to avoid a situation where your server must reopen all its environment handles just because you fire up a command line container administrative utility that always runs recovery.

BDB XML offers you two methods by which you can manage recovery for multi-process BDB XML applications. Each has different strengths and weaknesses, and they are described in the next sections.

Effects of Multi-Process Recovery

Before continuing, it is worth noting that the following sections describe recovery processes than can result in one process running recovery while other processes are currently actively performing container operations.

When this happens, the current container operation will abnormally fail, indicating a DB_RUNRECOVERY condition. This means that your application should immediately abandon any container operations that it may have on-going, discard any environment handles it has opened, and obtain and open new handles.

The net effect of this is that any writes performed by unresolved transactions will be lost. For persistent applications (servers, for example), the services it provides will also be unavailable for the amount of time that it takes to complete a recovery and for all participating processes to reopen their environment handles.

Process Registration

One way to handle multi-process recovery is for every process to "register" its environment. In doing so, the process gains the ability to see if any other applications are using the environment and, if so, whether they have suffered an abnormal termination. If an abnormal termination is detected, the process runs recovery; otherwise, it does not.

Note that using process registration also ensures that recovery is serialized across applications. That is, only one process at a time has a chance to run recovery. Generally this means that the first process to start up will run recovery, and all other processes will silently not run recovery because it is not needed.

To cause your application to register its environment, you specify the DB_REGISTER flag when you open your environment. You may also specify DB_RECOVER. However, it is an error to specify DB_RECOVER_FATAL when using the DB_REGISTER flag. If during the open, BDB XML determines

that recovery must be run, it will automatically run the correct type of recovery for you, so long as you specify normal recovery on your environment open. If you do not specify normal recovery, and you register your environment, then no recovery is run if the registration process identifies a need for it. In this case, the environment open simply fails by returning DB_RUNRECOVERY.

If you do not specify normal recovery when you open your first registered environment in the application, then that application will fail the environment open by returning DB_RUNRECOVERY. This is because the first process to register must create an internal registration file, and recovery is forced when that file is created. To avoid an abnormal termination of the environment open, specify recovery on the environment open for at least the first process starting in your application.

In addition, if you specify DB_ENV_FAILCHK when you register your environment, then a fail check is performed on environment open (fail checks are described in the next section). If, during the fail check process, an abnormal termination is detected for any of the processes involved in the application, BDB XML releases any read locks held by the dead process and performs transaction aborts as necessary. This is done in an attempt to clean up the environment.

In this situation, if a general cleanup of the environment is not possible and normal recovery is not specified on environment open, then the open will abort, returning DB_RUNRECOVERY. However, if this situation occurs and recovery was specified, then the appropriate type of recovery (normal or fatal) is run so as to bring the environment back to a healthy state.

Be aware that there are some limitations/requirements if you want your various processes to coordinate recovery using registration:

1. There can be only one environment handle per environment per process. In the case of multi-threaded processes, the environment handle must be shared across threads.

2. All processes sharing the environment must use registration. If registration is not uniformly used across all participating processes, then you can see inconsistent results in terms of your application's ability to recognize that recovery must be run.

Failure Checking

For very large and robust multi-process applications, the most common way to ensure all the processes are working as intended is to make use of a watchdog process. To assist a watchdog process, BDB XML offers a failure checking mechanism.

When a thread of control fails with open environment handles, the result is that there may be resources left locked or corrupted. Other threads of control may encountered these unavailable resources quickly or not at all, depending on data access patterns.

In any case, the BDB XML failure checking mechanism allows a watchdog to detect if an environment is unusable as a result of a thread of control failure. It should be called periodically (for example, once a minute) from the watchdog process. If the environment is deemed unusable, then the watchdog process is notified that recovery should be run. It is then up to the watchdog to actually run recovery. It is also the watchdog's responsibility to decide what

to do about currently running processes before running recovery. The watchdog could, for example, attempt to gracefully shutdown or kill all relevant processes before running recovery.

Note that failure checking need not be run from a separate process, although conceptually that is how the mechanism is meant to be used. This same mechanism could be used in a multi-threaded application that wants to have a watchdog thread.

To use failure checking you must:

1. Provide an `is_alive()` call back using the `Dbenv::set_isalive()` method. BDB XML uses this method to determine whether a specified process and thread is alive when the failure checking is performed.

2. Possibly provide a `thread_id` callback `ThreadIdentifier` interface implementation that uniquely identifies a process and thread of control. This callback is only necessary if the standard process and thread identification functions for your platform are not sufficient to for use by failure checking. This is rarely necessary and is usually because the thread and/or process ids used by your system cannot fit into an unsigned integer.

   You provide this callback using the `DB_ENV->set_thread_id()` method. See the API reference for this method for more information on when setting a thread id callback might be necessary.

3. Call the `DB_ENV->failchk()` method periodically. You can do this either periodically (once per minute, for example), or whenever a thread of control exits for your application.

   If this method determines that a thread of control exited holding read locks, those locks are automatically released. If the thread of control exited with an unresolved transaction, that transaction is aborted. If any other problems exist beyond these such that the environment must be recovered, the method will return `DB_RUNRECOVERY`.

Using Hot Failovers

You can maintain a backup that can be used for failover purposes. Hot failovers differ from the backup and restore procedures described previously in this chapter in that data used for traditional backups is typically copied to offline storage. Recovery time for a traditional backup is determined by:

- How quickly you can retrieve that storage media. Typically storage media for critical backups is moved to a safe facility in a remote location, so this step can take a relatively long time.

- How fast you can read the backup from the storage media to a local disk drive. If you have very large backups, or if your storage media is very slow, this can be a lengthy process.

- How long it takes you to run catastrophic recovery against the newly restored backup. As described earlier in this chapter, this process can be lengthy because every log file must be examined during the recovery process.

When you use a hot failover, the backup is maintained at a location that is reasonably fast to access. Usually, this is a second disk drive local to the machine. In this situation, recovery time

is very quick because you only have to reopen your environment and database, using the failover environment for the environment open.

Hot failovers obviously do not protect you from truly catastrophic disasters (such as a fire in your machine room) because the backup is still local to the machine. However, you can guard against more mundane problems (such as a broken disk drive) by keeping the backup on a second drive that is managed by an alternate disk controller.

To maintain a hot failover:

1. Copy all the active database files to the failover directory. Use the **db_archive** command line utility with the `-s` option to identify all the active database files.

2. Identify all the inactive log files in your production environment and *move* these to the failover directory. Use the **db_archive** command with no command line options to obtain a list of these log files.

3. Identify the active log files in your production environment, and *copy* these to the failover directory. Use the **db_archive** command with the `-l` option to obtain a list of these log files.

4. Run catastrophic recovery against the failover directory. Use the **db_recover** command with the `-c` option to do this.

5. Optionally copy the backup to an archival location.

Once you have performed this procedure, you can maintain an active hot backup by repeating steps 2 - 5 as often as is required by your application.

If you perform step 1, steps 2-5 must follow in order to ensure consistency of your hot backup.

Rather than use the previous procedure, you can use the **db_hotbackup** command line utility to do the same thing. This utility will (optionally) run a checkpoint and then copy all necessary files to a target directory for you.

To actually perform a failover, simply:

1. Shut down all processes which are running against the original environment.

2. If you have an archival copy of the backup environment, you can optionally try copying the remaining log files from the original environment and running catastrophic recovery against that backup environment. Do this *only* if you have a an archival copy of the backup environment.

   This step can allow you to recover data created or modified in the original environment, but which did not have a chance to be reflected in the hot backup environment.

3. Reopen your environment and containers as normal, but use the backup environment instead of the production environment.

Removing Log Files

By default BDB XML does not delete log files for you. For this reason, BDB XML's log files will eventually grow to consume an unnecessarily large amount of disk space. To guard against this, you should periodically take administrative action to remove log files that are no longer in use by your application.

You can remove a log file if all of the following are true:

- the log file is not involved in an active transaction.
- a checkpoint has been performed *after* the log file was created.
- the log file is not the only log file in the environment.
- the log file that you want to remove has already been included in an offline or hot backup. Failure to observe this last condition can cause your backups to be unusable.

BDB XML provides several mechanisms to remove log files that meet all but the last criteria (BDB XML has no way to know which log files have already been included in a backup). The following mechanisms make it easy to remove unneeded log files, but can result in an unusable backup if the log files are not first saved to your archive location. All of the following mechanisms automatically delete unneeded log files for you:

- Run the **db_archive** command line utility with the `-d` option.
- From within your application, call the `DbEnv::log_archive()` method with the `DB_ARCH_REMOVE` flag.
- Call `DB_ENV->::set_flags()` method with the `DB_LOG_AUTOREMOVE` flag. Note that this flag can be set at any point in the lifetime of your application. Setting this parameter affects all environment handles opened against the environment; not just the handle used to set the flag.

  Note that unlike the other log removal mechanisms identified here, this method actually causes log files to be removed on an on-going basis as they become unnecessary. This is extremely desirable behavior if what you want is to use the absolute minimum amount of disk space possible for your application. This mechanism *will* leave you with the log files that are required to run normal recovery. However, it is highly likely that this mechanism will prevent you from running catastrophic recovery.

  Do NOT use this mechanism if you want to be able to perform catastrophic recovery, or if you want to be able to maintain a hot backup.

In order to safely remove log files and still be able to perform catastrophic recovery, use the **db_archive** command line utility as follows:

1. Run either a normal or hot backup as described in Backup Procedures (page 55). Make sure that all of this data is safely stored to your backup media before continuing.

2. If you have not already done so, perform a checkpoint. See Checkpoints (page 53) for more information.

3. If you are maintaining a hot backup, perform the hot backup procedure as described in Using Hot Failovers (page 65).

4. Run the **db_archive** command line utility with the `-d` option against your production environment.

5. Run the **db_archive** command line utility with the `-d` option against your failover environment, if you are maintaining one.

Configuring the Logging Subsystem

You can configure the following aspects of the logging subsystem:

- Size of the log files.
- Size of the logging subsystem's region. See Configuring the Logging Region Size (page 69).
- Maintain logs entirely in-memory. See Configuring In-Memory Logging (page 69) for more information.
- Size of the log buffer in memory. See Setting the In-Memory Log Buffer Size (page 71).
- On-disk location of your log files. See Identifying Specific File Locations (page 8).

Setting the Log File Size

Whenever a pre-defined amount of data is written to a log file (10 MB by default), BDB XML stops using the current log file and starts writing to a new file. You can change the maximum amount of data contained in each log file by using the `DB_ENV->set_lg_max()` method. Note that this method can be used at any time during an application's lifetime.

Setting the log file size to something larger than its default value is largely a matter of convenience and a reflection of the application's preference in backup media and frequency. However, if you set the log file size too low relative to your application's traffic patterns, you can cause yourself trouble.

From a performance perspective, setting the log file size to a low value can cause your active transactions to pause their writing activities more frequently than would occur with larger log file sizes. Whenever a transaction completes the log buffer is flushed to disk. Normally other transactions can continue to write to the log buffer while this flush is in progress. However, when one log file is being closed and another created, all transactions must cease writing to the log buffer until the switch over is completed.

Beyond performance concerns, using smaller log files can cause you to use more physical files on disk. As a result, your application could run out of log sequence numbers, depending on how busy your application is.

Every log file is identified with a 10 digit number. Moreover, the maximum number of log files that your application is allowed to create in its lifetime is 2,000,000,000.

For example, if your application performs 6,000 transactions per second for 24 hours a day, and you are logging 500 bytes of data per transaction into 10 MB log files, then you will run out of log files in around 221 years:

```
(10 * 2^20 * 2000000000) / (6000 * 500 * 365 * 60 *60 * 24) = 221
```

However, if you were writing 2000 bytes of data per transaction, and using 1 MB log files, then the same formula shows you running out of log files in 5 years time.

All of these time frames are quite long, to be sure, but if you do run out of log files after, say, 5 years of continuous operations, then you must reset your log sequence numbers. To do so:

1. Backup your containers as if to prepare for catastrophic failure. See Backup Procedures (page 55) for more information.
2. Reset the log file's sequence number using the **db_load** utility's `-r` option.
3. Remove all of the log files from your environment. Note that this is the only situation in which all of the log files are removed from an environment; in all other cases, at least a single log file is retained.
4. Restart your application.

Configuring the Logging Region Size

The logging subsystem's default region size is 60 KB. The logging region is used to store filenames, and so you may need to increase its size if a large number of files (that is, if you have a very large number of databases) will be opened and registered with BDB XML's log manager.

You can set the size of your logging region by using the `DB_ENV->set_lg_regionmax()` method. Note that this method can only be called before the first environment handle for your application is opened.

Configuring In-Memory Logging

It is possible to configure your logging subsystem such that logs are maintained entirely in memory. When you do this, you give up your transactional durability guarantee. Without log files, you have no way to run recovery so any system or software failures that you might experience can corrupt your containers.

However, by giving up your durability guarantees, you can greatly improve your application's throughput by avoiding the disk I/O necessary to write logging information to disk. In this case, you still retain your transactional atomicity, consistency, and isolation guarantees.

To configure your logging subsystem to maintain your logs entirely in-memory:

- Make sure your log buffer is capable of holding all log information that can accumulate during the longest running transaction. See Setting the In-Memory Log Buffer Size (page 71) for details.
- Do not run normal recovery when you open your environment. In this configuration, there are no log files available against which you can run recovery. As a result, if you specify recovery when you open your environment, it is ignored.
- Specify `DB_LOG_IN_MEMORY` to the `DB_ENV->log_set_config()` method. Note that you must specify this before your application opens its first environment handle.

For example:

```
#include "DbXml.hpp"

...

int main(void)
{
    // Set the normal flags for a transactional subsystem. Note that
    // we DO NOT specify DB_RECOVER.
    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_THREAD     |  // Free-thread the env handle
                          DB_INIT_TXN;     // Initialize transactions

    DB_ENV *myEnv = 0;
    XmlManager *myManager = NULL;
    char *envHome = "/export1/testEnv";
    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);
    }

    // Indicate that logging is to be performed only in memory.
    // Doing this means that we give up our transactional durability
    // guarantee.
    myEnv.set_flags(DB_LOG_INMEMORY, 1);

    // Configure the size of our log memory buffer. This must be
    // large enough to hold all the logging information likely
```

```
    // to be created for our longest running transaction. The
    // default size for the logging buffer is 1 MB when logging
    // is performed in-memory. For this example, we arbitrarily
    // set the logging buffer to 5 MB.
    myEnv.set_lg_bsize(5 * 1024 * 1024);

    // Open the environment as normal.
    myEnv->open(myEnv, envHome, env_flags, 0);
    myManager = new XmlManager(myEnv, 0);

    // From here, you open containers, create transactions and
    // perform container operations exactly as you would if you
    // were logging to disk. This part is omitted for brevity.
```

Setting the In-Memory Log Buffer Size

When your application is configured for on-disk logging (the default behavior for transactional applications), log information is stored in-memory until the storage space fills up, or a transaction commit forces the log information to be flushed to disk.

It is possible to increase the amount of memory available to your file log buffer. Doing so improves throughput for long-running transactions, or for transactions that produce a large amount of data.

When you have your logging subsystem configured to maintain your log entirely in memory (see Configuring In-Memory Logging (page 69)), it is very important to configure your log buffer size because the log buffer must be capable of holding all log information that can accumulate during the longest running transaction. You must make sure that the in-memory log buffer size is large enough that no transaction will ever span the entire buffer. You must also avoid a state where the in-memory buffer is full and no space can be freed because a transaction that started the first log "file" is still active.

When your logging subsystem is configured for on-disk logging, the default log buffer space is 32 KB. When in-memory logging is configured, the default log buffer space is 1 MB.

You can increase your log buffer space using the `DB_ENV->set_lg_bsize()` method. Note that this method can only be called before the first environment handle for your application is opened.

Chapter 6. Summary and Examples

Throughout this manual we have presented the concepts and mechanisms that you need to provide transactional protection for your application. In this chapter, we summarize these mechanisms, and we provide a complete example of a multi-threaded transactional BDB XML application.

Anatomy of a Transactional Application

Transactional applications are characterized by performing the following activities:

1. Create your environment handle.
2. Open your environment, specifying that the following subsystems be used:
   - Transactional Subsystem (this also initializes the logging subsystem).
   - Memory pool (the in-memory cache).
   - Logging subsystem.
   - Locking subsystem (if your application is multi-process or multi-threaded).

   It is also highly recommended that you run normal recovery upon first environment open. Normal recovery examines only those logs required to ensure your container files are consistent relative to the information found in your log files.
3. Open your manager, passing to it your opened environment.
4. Optionally spawn off any utility threads that you might need. Utility threads can be used to run checkpoints periodically, or to periodically run a deadlock detector if you do not want to use BDB XML's built-in deadlock detector.
5. Open whatever container handles that you need.
6. Spawn off worker threads. How many of these you need and how they split their BDB XML workload is entirely up to your application's requirements. However, any worker threads that perform write operations will do the following:

   a. Begin a transaction.

   b. Perform one or more read and write operations.

   c. Commit the transaction if all goes well.

   d. Abort and retry the operation if a deadlock is detected.

   e. Abort the transaction for most other errors.

If you have read-only threads that are operating concurrently with read-write or write-only threads, then you should also transaction-protect the read operations in these threads.

7. On application shutdown:

   a. Make sure there are no active transactions. Either abort or commit all transactions before shutting down.

   b. Close your environment if you did not allow your manager to adopt it.

Robust BDB XML applications should monitor their worker threads to make sure they have not died unexpectedly. If a thread does terminate abnormally, you must shutdown all your worker threads and then run normal recovery (you will have to reopen your environment to do this). This is the only way to clear any resources (such as a lock or a mutex) that the abnormally exiting worker thread might have been holding at the time that it died.

Failure to perform this recovery can cause your still-functioning worker threads to eventually block forever while waiting for a lock that will never be released.

In addition to these activities, which are all entirely handled by code within your application, there are some administrative activities that you should perform:

- Periodically checkpoint your application. Checkpoints will reduce the time to run recovery in the event that one is required. See Checkpoints (page 53) for details.
- Periodically back up your container and log files. This is required in order to fully obtain the durability guarantee made by BDB XML's transaction ACID support. See Backup Procedures (page 55) for more information.
- You may want to maintain a hot failover if 24x7 processing with rapid restart in the face of a disk hit is important to you. See Using Hot Failovers (page 65) for more information.

Transaction Example

The following code provides a fully functional example of a multi-threaded transactional BDB XML application. For improved portability across platforms, this examples uses pthreads to provide threading support.

The example creates multiple threads, each of which creates a set number of XML documents that it then writes to the container. Each thread creates and writes 10 documents under a single transaction before committing and writing another 10 documents. This activity is repeated 50 times.

From the command line, you can tell the program to vary:

- The number of threads that it should use.
- The number of nodes each XML document will contain.
- Whether the container used by the program is of type Wholedoc or node storage.

- Whether read committed (degree 2) isolation should be used for the container writes.

As we will see in Runtime Analysis (page 90) each of these variables plays a role in the number of deadlocks the program encounters during its run time.

Of course, each writer thread performs deadlock detection as described in this manual. In addition, normal recovery is performed when the environment is opened.

We start with our normal `include` directives and other house keeping necessities:

```
// File TxnGuide.cpp

// We assume an ANSI-compatible compiler
#include "dbxml/DbXml.hpp"
#include <cstdlib>
#include <iostream>
#include <pthread.h>
#include <sstream>

#ifdef _WIN32
extern int getopt(int, char * const *, const char *);
#define PATHD '\\'
#else
#include <unistd.h>
#define PATHD '/'
#endif

using namespace DbXml;
```

Next, we declare a few global variables. `global_thread_num` is used to assist in creating a portable thread ID for each thread in use by the program. `global_num_deadlocks` is a variable that we use to count the total number of deadlocks the program encounters during its runtime. Finally, we declare a couple of pthread mutex variables that we will use to lock these variables when they are in use.

```
// File TxnGuide.cpp

// Printing of pthread_t is implementation-specific, so we
// create our own thread IDs for reporting purposes.
int global_thread_num;
int global_num_deadlocks;
pthread_mutex_t thread_num_lock, thread_num_deadlocks;
```

Next we perform a couple of forward function declarations. `usage()` provides our application's help text and `writerThread` is the function that will run for each thread.

We also declare a structure that we use to contain variables of local interest to our writer threads. We will pass this structure to each of our `writerThread` functions called by `pthread_create()`.

```
// Forward declarations
int usage(void);
void *writerThread(void *);

struct ThreadVars {
    XmlContainer container;
    bool useReadCommitted;
    int numNodes;
};
```

Next we implement our `usage()`, which describes how to use our application.

```
// Usage function
int
usage()
{
    std::cerr << "\nThis program writes XML documents to a DB XML"
              << "container. The documents are written using any number\n"
              << "of threads that will perform writes "
              << "using 50 transactions. Each transaction writes \n"
              << "10 documents. You can choose to perform the "
              << "writes using default isolation, or using \n"
              << "READ COMMITTED isolation. If READ COMMITTED "
              << "is used, the application will see fewer deadlocks."
              << std::endl;
     std::cerr << "\nNote that you can vary the size of the documents "
               << "written to the container by defining the number of \n"
               << "nodes in the documents. Up to a point, and depending "
               << "on your system's performance, increasing the number \n"
               << "of nodes will increase the number of deadlocks that "
               << "your application will see." << std::endl;
    std::cerr << "Command line options are: " << std::endl;
    std::cerr << " -h <database_home_directory>" << std::endl;
    std::cerr << " [-t <number of threads>]" << std::endl;
    std::cerr << " [-n <number of nodes per document>]" << std::endl;
    std::cerr << " [-w]       (create a Wholedoc container)"   << std::endl;
    std::cerr << " [-2]       (use READ COMMITTED isolation)" << std::endl;
    return (EXIT_FAILURE);
}
```

Now we implement our `main()` function. We start by declaring and initializing the local variables needed by the function. Notice that by default we will not use read committed isolation, we will use 5 threads, and our default container type is a node container.

```
int
main(int argc, char *argv[])
{

    DB_ENV *envp = NULL;
    XmlManager *mgrp = NULL;
```

```
    std::string containerName("txn.dbxml");

    ThreadVars threadInfo;
    threadInfo.useReadCommitted = false;

    // Initialize globals
    global_thread_num = 0;
    global_num_deadlocks = 0;

    int ch, i, dberr;
    int numThreads = 5;
    u_int32_t envFlags;
    XmlContainer::ContainerType containerType =
        XmlContainer::NodeContainer;
    char *dbHomeDir;

    // Application name
    const char *progName = "TxnGuide";
```

Now we parse the command line options. See the `usage()` function above for a description of what each of these options does.

```
    // Parse the command line arguments
#ifdef _WIN32
    dbHomeDir = ".\\";
#else
    dbHomeDir = "./";
#endif
    while ((ch = getopt(argc, argv, "h:n:t:w2")) != EOF)
        switch (ch) {
        case 'h':
            dbHomeDir = optarg;
            break;
        case 'n':
            threadInfo.numNodes = atoi(optarg);
            break;
        case 't':
            numThreads = atoi(optarg);
            break;
        case '2':
            threadInfo.useReadCommitted = true;
            break;
        case 'w':
            containerType = XmlContainer::WholedocContainer;
            break;
        case '?':
        default:
            return (usage());
        }
```

As a final bit of plumbing, we enforce the minimum values passed to the application and issue informative text indicating how the program will run:

```
    // Find out how many nodes we'll write to the container
    threadInfo.numNodes = threadInfo.numNodes < 1 ? 1 :
                          threadInfo.numNodes;

    // Find out how many threads
    numThreads = numThreads < 1 ? 1 : numThreads;

    std::cout << "Number nodes per document:       "
              << threadInfo.numNodes << std::endl;
    std::cout << "Number of writer threads:        " << numThreads
              << std::endl;

    std::string msg = threadInfo.useReadCommitted ?
                        "Read Committed " :
                        "Default";
    std::cout << "Isolation level:                 " << msg
              << std::endl;

    msg = containerType == XmlContainer::WholedocContainer ?
                           "Wholedoc storage" : "Node storage";
    std::cout << "Container type:                  " << msg << "\n\n"
              << std::endl;
```

Now that we know what it is that the program is supposed to do, we can start to do it. We begin by opening our environment, manager and container so that they support transactional processing.

Notice here that if our container already exists, we delete and then recreate it. This allows us to avoid document ID conflict. This also allows us to change the container type from run to run of the program since the container type can only be set at container creation time.

Finally, notice that we set up deadlock detection here, and we choose to resolve deadlocks by picking the thread with the smallest number of write locks. The thread with the smallest number of write locks is the one that has performed the least amount of work. By choosing this thread for the abort/retry cycle, we minimize the amount of rework our application must perform due to a deadlock.

```
    // Env open flags
    envFlags =
      DB_CREATE     |  // Create the environment if it does not exist
      DB_RECOVER    |  // Run normal recovery.
      DB_INIT_LOCK  |  // Initialize the locking subsystem
      DB_INIT_LOG   |  // Initialize the logging subsystem
      DB_INIT_TXN   |  // Initialize the transactional subsystem.
      DB_INIT_MPOOL |  // Initialize the memory pool (in-memory cache)
      DB_THREAD;       // Cause the environment to be free-threaded
```

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

    // Indicate that we want to internally perform deadlock
    // detection.  Also indicate that the transaction with
    // the fewest number of write locks will receive the
    // deadlock notification in the event of a deadlock.
    envp->set_lk_detect(envp, DB_LOCK_MINWRITE);

    envp->open(envp, dbHomeDir, env_flags, 0);

    myManager = new XmlManager(envp, 0);
    // Create and open a DB XML manager.
    mgrp = new XmlManager(envp,
                          DBXML_ADOPT_DBENV); // Close the env when
                                              // the manager closes.
    try {
        // If we had utility threads (for running checkpoints or
        // deadlock detection, for example) we would spawn those
        // here. However, for a simple example such as this,
        // that is not required.

        // If the container already exists, delete it. We don't want
        // naming conflicts if this program is run multiple times.
        if (mgrp->existsContainer(containerName) != 0)
            mgrp->removeContainer(containerName);

        XmlContainerConfig cconfig;
        cconfig.setTransactional(true);  // Container is transactional.
        cconfig.setThreaded(true);
        cconfig.setAllowCreate(true);    // Create the container if it
                                         // does not exist.
        cconfig.setContainerType(containerType);

        // Open the container
        threadInfo.container =
            mgrp->openContainer(<userinput>""</userinput>,
                cconfig); </programlisting>
        // Open the container
        threadInfo.container =
            mgrp->openContainer(containerName,
                cconfig);
```

Next we initialize our mutexes and we start and join our writer threads. This is all standard pthread usage, so we present it here without much comment.

```
        // Initialize a pthread mutex. Used to help provide thread ids.
        (void)pthread_mutex_init(&thread_num_lock, NULL);
        // Initialize a pthread mutex. Used to count the number of
        // deadlocks encountered by the various threads in this example.
        (void)pthread_mutex_init(&thread_num_deadlocks, NULL);

        // Start the writer threads.
        pthread_t writerThreads[numThreads];
        for (i = 0; i < numThreads; i++)
            (void)pthread_create(
                &writerThreads[i], NULL,
                writerThread, (void *)&threadInfo);

        // Join the writers
        for (i = 0; i < numThreads; i++)
            (void)pthread_join(writerThreads[i], NULL);
```

Of course we need to catch and handle any exceptions thrown during our application's runtime.

```
    } catch(XmlException &xe) {
        std::cerr << "Error opening XmlContainer: "
                  << xe.what() << std::endl;
        return (EXIT_FAILURE);
    } catch(std::exception &ee) {
        std::cerr << "Unknown error: "
                  << ee.what() << std::endl;
        return (EXIT_FAILURE);
    }
```

Once all our writer threads complete, we need to clean up a little. Remember that containers automatically close when they go out of scope. Also, our manager adopted the environment, so when the manager closes, it will close the environment for us. And, of course, the manager also closes when the last handle to it either goes out of scope or is deleted.

Consequently, to close our application all we need to do is delete the `XmlManager` object.

```
    try {
        // Close our manager if it was opened.
        if (mgrp != NULL)
            delete mgrp;

        // We don't have to close our container or
        // environment handles. The container closes
        // when it goes out of scope. The environment
        // is closed when the manager is deleted, because
        // we specified DBXML_ADOPT_DBENV on the manager
        // open.
```

```
    } catch(XmlException &xe) {
        std::cerr << progName << "Error closing manager and environment."
                  << std::endl;
        std::cerr << xe.what() << std::endl;
        return (EXIT_FAILURE);
    } catch(std::exception &ee) {
        std::cerr << progName << "Error closing manager and environment."
                  << std::endl;
        std::cerr << ee.what() << std::endl;
        return (EXIT_FAILURE);
    }
```

As a final bit of clean up, we issue a count of the deadlocks seen during this program runtime and then return from the `main()` function.

```
    // Final status message and return.

    std::cout << "I'm all done." << std::endl;
    std::cout << "I saw " << global_num_deadlocks
              << " deadlocks in this program run."
              << std::endl;
    return (EXIT_SUCCESS);
}
```

The writerThread Function

To perform actual work, our application spawns a number of threads, each of which runs the `writerThread()` function. This function:

- Runs 50 transactions.
- Within each transaction, it creates and writes 10 XML documents to the container.
- The size of each document is determined by information provided on the command line.
- Documents can be written using read committed isolation, depending on information provided on the command line.
- In the event of a deadlock, the function will abort the transaction and then retry. Note that the function will only retry a given transaction 30 times before giving up and moving on to the next transaction.
- Upon completing its workload, the function increments the global deadlock counter with the number of deadlocks that it saw before exiting.

To begin, the function sets up the local variables that it needs in order to perform its work. Notice that we create the `XmlUpdateContext` object at the top of this function; there is no need for us to continually recreate that object as the function iterates over its workload.

```
// A function that performs a series of writes to a
// Berkeley DBXML container.
```

```
// The mechanism of transactional commit/abort and
// deadlock detection is illustrated here.
void *
writerThread(void *args)
{
    int j, thread_num;
    int max_retries = 30;   // Max retry on a deadlock
    int num_deadlocks = 0;

    ThreadVars *threadInfo = (ThreadVars *)args;
    XmlContainer container = threadInfo->container;
    XmlManager myManager = container.getManager();
    XmlTransaction txn;
    XmlUpdateContext context = myManager.createUpdateContext();
```

Next we determine our thread ID. Some pthread packages allow us to use a `pthread_t` variable, as returned by `pthread_self()`, for this purpose because for those packages a `pthread_t` is really just an integer type. However, this is not universally true – in some implementations `pthread_t` is a structure, for example – so we use a simple global counter for this purpose instead.

```
    // Get the thread number
    (void)pthread_mutex_lock(&thread_num_lock);
    global_thread_num++;
    thread_num = global_thread_num;
    (void)pthread_mutex_unlock(&thread_num_lock);
```

Having done that, we also initialize our random number generator. We use this to create random data for our XML documents so that they are not all identical.

```
    // Initialize the random number generator
    srand((u_int)pthread_self());
```

Now we get to the main workload loop in our application. Here we begin the `for` loop that will perform the 50 transactions, and we begin the retry `while` loop.

```
    // Perform 50 transactions
    for (int i=0; i<50; i++) {
        bool retry = true;
        int retry_count = 0;
        // while loop is used for deadlock retries
        while (retry) {
            // try block used for deadlock detection and
            // general exception handling
            try {
```

Now that we are inside our `try` block, we can create our transaction. Notice that we set on a transaction by transaction basis whether read committed isolation is used. Lowering the isolation level for the transaction for this workload will result in fewer lock contentions and therefore fewer deadlocks. See Runtime Analysis (page 90) for more information.

```
// Set this transaction to use READ COMMITTED isolation
// if it is indicated by the command line switches.
u_int32_t txnFlags =
    threadInfo->useReadCommitted ? DB_READ_COMMITTED : 0;
txn = myManager.createTransaction(txnFlags);
```

Now we create and write the 10 documents for this transaction. Remember that the size of the document is determined by information provided on the command line. Again, the size of the document has a lot to do with the amount of lock contention the application will see.

Beyond that, this portion of the application is simply basic BDB XML library usage.

```
// Perform the container writes for this transaction.
for (j = 0; j < 10; j++) {

    // Get a document ID
    std::ostringstream docID;
    docID << thread_num << i << j;

    // Build the document
    std::ostringstream theDoc;
    theDoc << "<testDoc>\n";
    for (int i = 0; i < threadInfo->numNodes; i++) {
        int payload = rand() + i;
        theDoc << "<payload>" << payload
               << "</payload>\n";
    }
    theDoc << "</testDoc>";

    // Put the document
    container.putDocument(txn,
                          docID.str(),
                          theDoc.str(),
                          context,
                          0);
}
```

Now that we are all done writing to the container, we can commit the transaction. If all has gone well, we are done with this particular transaction and we can iterate to the next transaction without retrying the current one.

```
// commit
txn.commit(0);
retry = false;
```

However, if an exception has been thrown, we must decide what to do about it. Our first concern, and the most likely cause of an exception given our workload, is that we have encountered a deadlock. So we begin by catching `XmlException` and testing to see if we have a deadlock situation.

If we do see a deadlock, we immediately abort the transaction which releases our locks, thereby allowing the other deadlock thread to make forward progress. We then must decide if we can retry the transaction; this is gated by the number of retry attempts we have made so far.

If we have caught an `XmlException` and it is *not* a deadlock situation, then we simply abort and give up on the current transaction. The function will then loop to the next transaction where, hopefully, we will not encounter any further unexpected exceptions.

```
            } catch (XmlException &xxe) {
                if (xxe.getDbErrno() == DB_LOCK_DEADLOCK) {
                    // First thing that we MUST do is abort the transaction.
                    txn.abort();

                    // Now we decide if we want to retry the operation.
                    // If we have retried less than max_retries,
                    // increment the retry count and goto retry.
                    if (retry_count < max_retries) {
                    //    std::cout << "############### Writer " << thread_num
                    //          << ": Got DB_LOCK_DEADLOCK. "
                    //          << "Retrying write operation."
                    //          << std::endl;
                        retry_count++;
                        retry = true;
                    } else {
                        // Otherwise, just give up.
                        std::cerr << "Writer " << thread_num
                              << ": Got DB_LOCK_DEADLOCK and I'm out of "
                              << "retries. Giving up." << std::endl;
                        retry = false;
                    }
                    num_deadlocks++;
                } else {
                    std::cerr << "Caught an XmlException : "
                             << xxe.what() << std::endl;
                    txn.abort();
                    retry = false;
                }
```

We also catch `std::exception` just for the sake of completeness. As is the case with a general `XmlException` event, here we abort and do not attempt to retry the current transaction.

```
           } catch (std::exception &ee) {
                std::cerr << "Unknown exception: " << ee.what()
                          << std::endl;
                txn.abort();
                retry = false;
          }
        }
    }
```

Finally, we increment our global deadlock counter before exiting the function. This is used for reporting purposes when the application itself exits.

```
    (void)pthread_mutex_lock(&thread_num_deadlocks);
    global_num_deadlocks += num_deadlocks;
    (void)pthread_mutex_unlock(&thread_num_deadlocks);
    return (0);
}
```

This completes our transactional example. If you would like to experiment with this code, you can find the example in the following location in your BDB XML distribution:

```
BDBXML_INSTALL/dbxml/examples/cxx/txn
```

In addition, please see Runtime Analysis (page 90) for an analysis on the performance characteristic illustrated by this program.

In-Memory Transaction Example

Some applications use XML documents in a transient manner. That is, they create and store XML documents as a part of their run time, but there is no need for the documents to persist between application restarts. For these class of applications, overall throughput can be improved by abandoning the transactional durability guarantee. To do this, you keep your environment, containers, and logs entirely in-memory so as to avoid the performance impact of unneeded disk I/O.

To do this:

- Refrain from specifying a home directory when you open your environment. The exception to this is if you are using the `DB_CONFIG` configuration file — in that case you must identify the environment's home directory so that the configuration file can be found.
- Configure your environment to back your regions from system memory instead of the filesystem.
- Configure your logging subsystem such that log files are kept entirely in-memory.
- Increase the size of your in-memory log buffer so that it is large enough to hold the largest set of concurrent write operations.
- Increase the size of your in-memory cache so that it can hold your entire data set. You do not want your cache to page to disk.
- Specify an empty string when you open your container. Note that for in-memory operations, you are limited to just one container.

As an example, this section takes the transaction example provided in Transaction Example (page 73) and it updates that example so that the environment, container, log files, and regions are all kept entirely in-memory.

To begin, we simplify the beginning of our example a bit. Because we no longer need an environment home directory, we can remove all the code that we used to determine path delimiters.

```
// File TxnGuideInMemory.cpp

// We assume an ANSI-compatible compiler
#include "dbxml/DbXml.hpp"
#include <cstdlib>
#include <iostream>
#include <pthread.h>
#include <sstream>

#ifdef _WIN32
extern int getopt(int, char * const *, const char *);
#endif

using namespace DbXml;

// Printing of pthread_t is implementation-specific, so we
// create our own thread IDs for reporting purposes.
int global_thread_num;
int global_num_deadlocks;
pthread_mutex_t thread_num_lock, thread_num_deadlocks;

// Forward declarations
int usage(void);
void *writerThread(void *);

struct ThreadVars {
    XmlContainer container;
    bool useReadCommitted;
    int numNodes;
};
```

Next, we modify the `usage()` function so that it no longer mentions the `-h` option which was used to specify the environment home directory.

```
// Usage function
int
usage()
{
    std::cerr << "\nThis program writes XML documents to a DB XML"
              << "container. The documents are written using any number\n"
              << "of threads that will perform writes "
              << "using 50 transactions. Each transaction writes \n"
              << "10 documents. You can choose to perform the "
              << "writes using default isolation, or using \n"
              << "READ COMMITTED isolation. If READ COMMITTED "
              << "is used, the application will see fewer deadlocks."
```

```
              << std::endl;
     std::cerr << "\nNote that you can vary the size of the documents "
               << "written to the container by defining the number of \n"
               << "nodes in the documents. Up to a point, and depending "
               << "on your system's performance, increasing the number \n"
               << "of nodes will increase the number of deadlocks that "
               << "your application will see." << std::endl;
    std::cerr << "Command line options are: " << std::endl;
    std::cerr << " [-t <number of threads>]" << std::endl;
    std::cerr << " [-n <number of nodes per document>]" << std::endl;
    std::cerr << " [-w]       (create a Wholedoc container)"   << std::endl;
    std::cerr << " [-2]       (use READ COMMITTED isolation)" << std::endl;
    return (EXIT_FAILURE);
}
```

We are also able to eliminate the `containerName` and `dbHomeDir` variables from our `main()`.

```
int
main(int argc, char *argv[])
{

    DB_ENV *envp = NULL;
    XmlManager *mgrp = NULL;

    ThreadVars threadInfo;
    threadInfo.useReadCommitted = false;

    // Initialize globals
    global_thread_num = 0;
    global_num_deadlocks = 0;

    int ch, i, dberr;
    int numThreads = 5;
    u_int32_t envFlags;
    XmlContainer::ContainerType containerType =
        XmlContainer::NodeContainer;

    // Application name
    const char *progName = "TxnGuide-inmem";
```

Parsing the command line arguments is somewhat simpler now too. We no longer care about the difference in file path delimiters between a windows and a unix system, and we no longer support the `-h` option.

```
    // Parse the command line arguments
    while ((ch = getopt(argc, argv, "n:t:w2")) != EOF)
        switch (ch) {
        case 'n':
            threadInfo.numNodes = atoi(optarg);
            break;
```

```
        case 't':
            numThreads = atoi(optarg);
            break;
        case '2':
            threadInfo.useReadCommitted = true;
            break;
        case 'w':
            containerType = XmlContainer::WholedocContainer;
            break;
        case '?':
        default:
            return (usage());
        }
```

Until now we have only eliminated things from the program. This is to be expected; after all, we need to collect less information in order to operate and so our code should be slightly simpler.

But now we need to start adding information to tell the Berkeley DB library that it must keep information in-memory only. We start by making the environment private; this causes all the region files to be kept in memory. (Additional code is in **bold**.)

Note that we also remove the DB_RECOVER flag from the environment open flags. Because our containers, logs, and regions are maintained in-memory, there can never be anything to recover.

```
    // Find out how many nodes we'll write to the container
    threadInfo.numNodes = threadInfo.numNodes < 1 ? 1 :
                          threadInfo.numNodes;

    // Find out how many threads
    numThreads = numThreads < 1 ? 1 : numThreads;

    std::cout << "Number nodes per document:       "
              << threadInfo.numNodes << std::endl;
    std::cout << "Number of writer threads:        " << numThreads
              << std::endl;

    std::string msg = threadInfo.useReadCommitted ?
                      "Read Committed " :
                      "Default";
    std::cout << "Isolation level:                 " << msg
              << std::endl;

    msg = containerType == XmlContainer::WholedocContainer ?
                           "Wholedoc storage" : "Node storage";
    std::cout << "Container type:                  " << msg << "\n\n"
              << std::endl;

    // Env open flags
    envFlags =
```

```
    DB_CREATE     |  // Create the environment if it does not exist
    // Removed DB_RECOVER flag
    DB_INIT_LOCK  |  // Initialize the locking subsystem
    DB_INIT_LOG   |  // Initialize the logging subsystem
    DB_INIT_TXN   |  // Initialize the transactional subsystem.
    DB_INIT_MPOOL |  // Initialize the memory pool (in-memory cache)
    DB_PRIVATE    |  // Region files are not backed by the filesystem.
                     // Instead, they are backed by heap memory.
    DB_THREAD;       // Cause the environment to be free-threaded
```

Now we configure our environment to keep the log files in memory, increase the log buffer size to 10 MB, and increase our in-memory cache to 10 MB. These values should be more than enough for our application's workload.

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

// Specify in-memory logging
envp->set_flags(envp, DB_LOG_INMEMORY, 1);

// Specify the size of the in-memory log buffer.
envp->set_lg_bsize(envp, 10 * 1024 * 1024);

// Specify the size of the in-memory cache
envp->set_cachesize(envp, 0, 10 * 1024 * 1024, 1);
```

Next, we open the environment and setup our lock detection. This is identical to how the example previously worked, except that we do not provide a location for the environment's home directory.

```
// Indicate that we want to internally perform deadlock
// detection.  Also indicate that the transaction with
// the fewest number of write locks will receive the
// deadlock notification in the event of a deadlock.
envp->set_lk_detect(envp, DB_LOCK_MINWRITE);

envp->open(envp, NULL, env_flags, 0);

myManager = new XmlManager(envp, 0);
// Create and open a DB XML manager.
mgrp = new XmlManager(envp,
                      DBXML_ADOPT_DBENV); // Close the env when
```

```
                        // the manager closes.
    try {
```

When we open our container, we provide an empty string for the container name. This causes the container to be kept entirely in memory.

```
        XmlContainerConfig cconfig;
        cconfig.setTransactional(true);  // Container is transactional.
        cconfig.setThreaded(true);
        cconfig.setAllowCreate(true);    // Create the container if it
                                         // does not exist.
        cconfig.setContainerType(containerType);

        // Open the container
        threadInfo.container =
            mgrp->openContainer("",
                cconfig);
```

After that, our `main()` function is unchanged, except that our error messages are changed so as to not reference the environment home directory.

```
        // Initialize a pthread mutex. Used to help provide thread ids.
        (void)pthread_mutex_init(&thread_num_lock, NULL);
        // Initialize a pthread mutex. Used to count the number of
        // deadlocks encountered by the various threads in this example.
        (void)pthread_mutex_init(&thread_num_deadlocks, NULL);

        // Start the writer threads.
        pthread_t writerThreads[numThreads];
        for (i = 0; i < numThreads; i++)
            (void)pthread_create(
                &writerThreads[i], NULL,
                writerThread, (void *)&threadInfo);

        // Join the writers
        for (i = 0; i < numThreads; i++)
            (void)pthread_join(writerThreads[i], NULL);

    } catch(XmlException &xe) {
        std::cerr << "Error opening XmlManager and Container: "
                  << std::endl;
        std::cerr << xe.what() << std::endl;
        return (EXIT_FAILURE);
    } catch(std::exception &ee) {
        std::cerr << "Unknown error: "
                  << ee.what() << std::endl;
        return (EXIT_FAILURE);
    }
```

```
    try {
        // Close our manager if it was opened.
        if (mgrp != NULL)
            delete mgrp;

        // We don't have to close our container or
        // environment handles. The container closes
        // when it goes out of scope. The environment
        // is closed when the manager is deleted, because
        // we specified DBXML_ADOPT_DBENV on the manager
        // open.

    } catch(XmlException &xe) {
        std::cerr << progName << "Error closing manager and environment."
                  << std::endl;
        std::cerr << xe.what() << std::endl;
        return (EXIT_FAILURE);
    } catch(std::exception &ee) {
        std::cerr << progName << "Error closing manager and environment."
                  << std::endl;
        std::cerr << ee.what() << std::endl;
        return (EXIT_FAILURE);
    }

    // Final status message and return.

    std::cout << "I'm all done." << std::endl;
    std::cout << "I saw " << global_num_deadlocks
              << " deadlocks in this program run."
              << std::endl;
    return (EXIT_SUCCESS);
}
```

That completes the updates we must make in order to cause the application to keep its environment, container, and logs entirely in memory. The `writerThread()` is left entirely unchanged.

If you would like to experiment with this code, you can find the example in the following location in your BDB XML distribution:

```
BDBXML_INSTALL/dbxml/examples/cxx/txn
```

Runtime Analysis

The examples presented in this chapter allow you to manipulate certain runtime characteristics that will affect the number of deadlocks the program will encounter. You can modify:

- The number of threads the program will use to write the container.

- The number of nodes that will be created per document written to the container. The key thing here is the size of the documents, as we will see later on in this section.
- Whether default isolation is used for the container writes, or if read committed should be used instead.
- Whether the container uses wholedoc or node storage.

The point of the application is to measure the number of deadlocks encountered for a given program run. By counting the number of deadlocks, we can get a sense of the overall amount of lock contention occurring in our application. Remember that deadlocks represent a race condition that your application lost. In order to occur, two more more threads had to have attempted to lock database pages in such a way that the threads blocked waiting for locks that will never be released (see Locks, Blocks, and Deadlocks (page 31) for a more complete description). So by examining the number of deadlocks that we see, we can indirectly get a sense for the amount of lock contention that the application encountered. Roughly speaking, the more deadlocks seen, the more lock contention that was going on during the application run.

Note that as you modify these constraints, you will see that the program will encounter differing numbers of deadlocks per program run. No two program runs will indicate the same number of deadlocks, but changing these constraint can on average increase or decrease the number of deadlocks reported by the application.

The reason why this application sees deadlocks is because of what BDB XML does under the hood. Recall that BDB XML writes XML documents to underlying Berkeley DB databases. Also, recall the Berkeley DB databases are usually organized in pages; multiple database entries will exist on any given page. Also, Berkeley DB uses page-level locking. The result is that multiple XML documents (or portions of XML documents) can and will be stored on the same database page. When multiple threads attempt to lock a database page, you get lock contention. When multiple database pages are in use and they are locked out of order by the threads of control, you can see deadlocks.

Therefore, the things that will immediately affect the amount of lock contention our application will encounter are:

- Number of threads. If you only ever use a single thread to write to your containers, you will never see any lock contention or deadlocks. On the other hand, increasing the number of writer threads will increase the number of deadlocks that are reported — up to a point. Recall that deadlocks are the result of losing a race condition. As you increase the number of threads in use, your system will slow down due to the overhead from context switching. This system slowdown will result in at least a leveling out of the number of deadlocks, if not an outright reduction in them. Of course, the point at which this occurs depends on the hardware in use.
- XML document size relative to the underlying database page size. The fewer documents that share a database page, the less chance there is for lock contention and therefore deadlocks. For our workload, the worse thing you can do is have lots of little database entries and a very large page size. Using large documents relative to the page size allows the document

to fill up the page, which means that, for this example program anyway, there will only ever be one locker for that page.

Note that selecting whole document versus node storage for the container plays into this equation. Whole document storage causes the XML document to be written using a single database entry. As a result, the entry itself is fairly large and so the underlying page is less likely to be shared by another document (depending on document size, of course). Conversely, node storage stores the document's individual nodes as individual database entries. Depending on the document, this can result in a lot of tiny database entries, which can adversely affect write performance due to increased lock contention. (Of course, the flip side to that is that node storage actually improves container query and read performance, but you will have to take our word for it because our sample application does not model that behavior.)

- Isolation level. Recall that by default, Berkeley DB hangs on to all write locks until the transaction either commits or aborts. It does this so as to provide your threads of control with the maximum isolation protection possible. However, hanging on to write locks like this means that our example application will encounter more lock contention and therefore see more deadlocks.

  If your application can accept a lessened isolation guarantee, and this one can, then you can reduce the isolation so as to reduce the amount of lock contention. In our case, we provide a way to use read committed (degree 2) isolation. Read committed causes the transaction to release the write lock as soon as it is finished writing to the page. Since the write locks are held for a shorter amount of time, there is less risk of lock contention and, again, deadlocks.

  For this workload, using read committed isolation results in a dramatic decrease in the reported number of deadlocks, which means that our application is simply working more efficiently.

Default Program Run

By default, the program makes the following choices:

```
> ./TxnGuide -h myEnvironmentDirectory
Number nodes per document:       1
Number of writer threads:        5
Isolation level:                 Default
Container type:                  Node storage
```

This represents a worse-case situation for the application in all ways but one; it uses small documents that are just one node in size. Running the example three times in a row results in 370, 317, and 382 reported deadlocks for an an average of 356.333 deadlocks. Note that your own test results will likely differ depending on the number and speed of your CPUs and the speed of your hard drive. For the record, these test results were taken using a single CPU PowerPC G3 system with a slow (4200 RPM) laptop hard drive.

Varying the Node Size

With a default node size of 1, we saw an average of 356.333 reported deadlocks over three runs of the program. Now lets try increasing the size of the document to see what it does to the number of reported deadlocks:

```
> ./TxnGuide -h myEnvironmentDirectory -n 10
Number nodes per document:       10
Number of writer threads:        5
Isolation level:                 Default
Container type:                  Node storage
```

This results in 894, 854, and 861 deadlocks for an average of 869.667 reported deadlocks. Clearly the amount of lock contention that we are seeing has increased, but why?

Remember that larger documents should fill up database pages, which should result in less lock contention because there are fewer lockers per database page. However, we are using node storage which means that the additional nodes results in additional small database entries. Given the way our application is writing documents, adding 9 additional nodes per document simply increases the chance of even more documents placing nodes on any given page.

Notice that there is a limit to the amount of lock contention that this application will see by simply adding nodes to the documents it creates. For example, suppose we created documents with 100 nodes:

```
> ./TxnGuide -h myEnvironmentDirectory -n 100
Number nodes per document:       100
Number of writer threads:        5
Isolation level:                 Default
Container type:                  Node storage
```

In this case, we see an average of 316 deadlocks — less, even, than the single node case. Why? First, the documents are now very large so there is a good chance that each document is filling up entire pages, even though we are still using node-level storage. In addition, each thread is now busy creating documents and then writing them to the containers, where they are being deconstructed into individual nodes. All of this is CPU-intensive activity that is likely helping to prevent lock contention because each thread is spending more time on document handling than it does with the smaller document sizes.

Using Wholedoc Storage

In the previous section we saw that specifying a document node size of 10 resulted in an average of 869.667 deadlocks across three program runs. This indicates a fairly high level of lock contention. It also indicates that the program is not operating particularly efficiently.

One way we could improve the write throughput for our application is to use whole document storage instead of node-level storage. This will result in fewer, but larger, database entries. The result should be fewer threads of control fighting for locks on a given page because fewer individual documents will be held on any given page.

Specifying a node size of 10 with whole document storage:

```
> ./TxnGuide -h myEnvironmentDirectory -n 10 -w
Number nodes per document:       10
Number of writer threads:        5
Isolation level:                 Default
Container type:                  Wholedoc storage
```

gives us an average deadlock count of 556 across three program runs. That's certainly a significant improvement over node-level storage, although for many workloads you will pay for it in terms of the higher query times that wholedoc storage will cost you.

Using Read Committed Isolation

Another way we can modestly improve our write performance is by using read committed isolation. This causes our transactions to release write locks immediately, instead of waiting until the transaction is resolved. Using read committed isolation does not gives us the dramatic write performance that does using wholedoc storage (see the previous section) but it is still an improvement.

```
> ./TxnGuide -h myEnvironmentDirectory -n 10 -2
Number nodes per document:       10
Number of writer threads:        5
Isolation level:                 Read Committed
Container type:                  Node storage
```

The average number of deadlocks seen across three runs with these settings is 724, down from 869.667. This is a modest improvement to be sure, but then you do not have to pay the query penalty that wholedoc containers might cost you.

Read Committed with Wholedoc Storage

Finally, the best improvement we can hope to see for this application, using 10 node documents and 5 writer threads, is to use read committed isolation to write to whole document containers.

```
> ./TxnGuide -h myEnvironmentDirectory -n 10 -w -2
Number nodes per document:       10
Number of writer threads:        5
Isolation level:                 Read Committed
Container type:                  Wholedoc storage
```

For three runs of the program with these settings, we observe 228.333 deadlocks — a remarkable improvement over the worst-case 869.667 that we saw for 10 nodes, 5 writer threads!