Berkeley DB Transaction Processing for C++

Makers of Berkeley DB

Legal Notice

Published 12/7/2005

Table of Contents

Preface

This document describes how to use transactions with your Berkeley DB 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 and multi-process DB applications and the requirements they have for deadlock detection.

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

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

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."

Variable or non-literal text is presented in *italics*. For example: "Go to your *DB_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 DB application:

- Berkeley DB for C++ Getting Started Guide
  [http://www.sleepycat.com/docs/gsg/CXX/BerkeleyDB-Core-Cxx-GSG.pdf]
- Berkeley DB Programmer's Tutorial and Reference Guide
  [http://www.sleepycat.com/docs/ref/toc.html]
- C++ API Reference [http://www.sleepycat.com/docs/api_cxx/frame.html]

Chapter 1. Introduction

This book provides a thorough introduction and discussion on transactions as used with Berkeley DB (DB). 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 *Getting Started with Berkeley DB* before reading this book.

Transaction Benefits

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

- **A**tomicity

  Multiple database 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 databases. Further, in the event that you abort a transaction, all write operations performed during the transaction are discarded. In this event, your database 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 DB transactions can span one or more database handles.

- **C**onsistency

  Your databases 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 database 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 DB. If, for example, your application requires every record to include an employee ID, and your code faithfully adds that ID to the records it's database records, then DB will never violate that consistency requirement. The ID will remain in the database records until such a time as your application chooses to delete it.

- **I**solation

  While a transaction is in progress, your databases 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. See Isolation (page 40) for more information.

- **D**urability

  Once committed to your databases, 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 19) 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 databases 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 databases will be otherwise unaffected.

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 you 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 databases.

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 optional for non-transactional applications, but they are required for transactional applications.

  Environment usage is described in detail in Transaction Basics (page 16).

- 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 DB Files (page 50) for more information.

- DbTxn 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 DbEnv 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 20) for information on how to use auto commit.

- Database open requirements.

  In addition to using environments and initializing the correct subsystems, your application must transaction protect the database opens, and any secondary index associations, if subsequent operations on the databases are to be transaction protected. The database open and secondary index association are commonly transaction protected using auto commit.

- 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 29) 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 DB 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 28) for more information.

Multi-threaded and Multi-process Applications

DB 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, DB provides mechanisms that allow you to detect and respond to deadlocks (but strictly speaking, this is not limited to just transactional applications).

Isolation means that database 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 database operations performed by different threads.

Note that DB 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 28).

Recoverability

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

In order to provide the transactional durability guarantee, DB uses a write-ahead logging system. Every operation performed on your databases is described in a log before it is performed on your databases. 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. DB 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 database is in a consistent state. DB 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 DB Files (page 50).

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.sleepycat.com/docs/ref/transapp/tune.html] and Transaction throughput [http://www.sleepycat.com/docs/ref/transapp/throughput.html] sections of the *Berkeley DB Programmer's Tutorial and 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 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 `DbEnv::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 DB 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 `DbEnv::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 28) for more information.

  You initialize the locking subsystem by passing the `DB_INIT_LOCK` flag to the `DbEnv::open()` method.
- Initialize the logging subsystem. While this is enabled by default for transactional applications, Sleepycat recommends 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 DB Files (page 50) for more information.

  You initialize the logging subsystem by passing the `DB_INIT_LOG` flag to the `DbEnv::open()` method.
- Transaction-enable your databases. You do this by encapsulating the database open in a transaction. Note that the common practice is for auto commit to be used to transaction-protect the database open.

Environments

For simple DB applications, environments are optional. However, in order to transaction protect your database operations, you must use an environment.

An *environment*, represents an encapsulation of one or more databases 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 DB databases are backed by files on disk. In addition to these files, transactional DB applications create logs that are also by default stored on disk (they can optionally be backed using shared memory). Finally, transactional DB applications also create and use shared-memory regions that are also typically backed by the filesystem. But like databases 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

File Naming

In order to operate, your DB application must be able to locate its database files, log files, and region files. If these are stored in the filesystem, then you must tell DB 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 DB 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 `DbEnv::open()` method, then that location is always used for the environment home.
- If a home directory is not supplied to `DbEnv::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 `DbEnv::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 DB files are created relative to the environment home directory. For example, suppose your environment home is in `/export/myAppHome`. Also suppose you name your database `data/myDatabase.db`. Then in this case, the database is placed in: `/export/myAppHome/data/myDatabase.db`.

That said, DB always defers to absolute pathnames. This means that if you provide an absolute filename when you name your database, 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 ('\').

Sleepycat strongly discourages you from using 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 you 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, DB will place all its files in or relative to the environment home directory. You can also cause a specific database 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 database, 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 database 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 DB's files to be placed in specific locations using the following mechanisms:

File Type	To Override
database files	You can cause database files to be created in a directory other than the environment home by using the `DbEnv::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 database 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 `DbEnv::set_data_dir()` method.

File Type	To Override
Log files	You can cause log files to be created in a directory other than the environment home directory by using the `DbEnv::set_lg_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 database 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 `DbEnv::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. Note that many of these methods are identical to the error handling methods available for the Db class. They are:

- `set_error_stream()`

  Sets the C++ `ostream` to be used for displaying error messages issued by the DB library.

- `set_errcall()`

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

- `set_errfile()`

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

- `set_errpfx()`

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

- `err()`

  Issues an error message based upon a DB 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 DB error message associated with the supplied error code, and a trailing newline.

- `errx()`

  Behaves identically to `err()` except that you do not provide the DB error code and so the DB 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 Berkeley DB for C++ Getting Started Guide. [http://www.sleepycat.com/docs/gsg/CXX/BerkeleyDB-Core-Cxx-GSG.pdf]

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 `DbEnv::open()` method.

(For an example of an entirely in-memory transactional application, see In-Memory Transaction Example (page 80).)

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 `DbEnv::open()` method.

When region files are backed by system memory, DB 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, DB 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. Additionally, VxWorks systems use system memory. In these cases, an initial segment ID must be specified by the application to ensure that applications do not overwrite each other's environments, so that the number of segments created does not grow without bounds. See the `DbEnv::set_shm_key()` method for more information.

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

For example, suppose you write a DB 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 the DB_USE_ENVIRON or DB_USE_ENVIRON_ROOT flags are set for the environment, 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 databases 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 the DB_USE_ENVIRON or DB_USE_ENVIRON_ROOT flags 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, DB always creates database 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 DB.

- 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 DbEnv::set_tmp_dir() method to specify a temporary directory with known permissions.

Opening a Transactional Environment and Database

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). Frequently, but not always, you will also initialize the locking subsystem. For example:

```
#include "db_cxx.h"

...

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

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

    try {

        myEnv.open(envHome.c_str(), env_flags, 0);

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

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

    return (EXIT_SUCCESS);
}
```

You then create and open your database(s) as you would for a non-transactional system. The only difference is that you must pass the environment handle to the `DbEnv::open()` method, and you must open the database within a transaction. Typically auto commit is

used for this purpose. To do so, pass DB_AUTO_COMMIT to the database open command. Also, make sure you close all your databases before you close your environment. For example:

```
#include "db_cxx.h"

...

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";

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

    try {

        myEnv.open(envHome.c_str(), env_flags, 0);
        dbp = new Db(&myEnv, 0);
        dbp->open(dbp,        // Pointer to the database
                  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

    } catch(DbException &e) {
        std::cerr << "Error opening database and environment: "
                  << file_name << ", "
                  << envHome << std::endl;
        std::cerr << e.what() << std::endl;
    }

    try {
        dbp->close(dbp, 0);
        myEnv.close(0);
    } catch(DbException &e) {
        std::cerr << "Error closing database and environment: "
                  << file_name << ", "
```

```
                    << 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 databases, you can use them to protect your database 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 `DbEnv::txn_begin()` method.

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

If, for any reason, you want to abandon the transaction, you abort it using `DbTxn::abort()`.

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 databases and environment.

If you only want to transaction protect a single database 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 20) for more information.

For example, the following example opens a transactional-enabled environment and database, 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 database is left in a state as if no operations had ever been attempted in the first place.

```
#include "db_cxx.h"

...

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";
    const char *keystr ="thekey";
    const char *datastr = "thedata";

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

```
    DbEnv myEnv(0);

    try {

        myEnv.open(envHome.c_str(), env_flags, 0);
        dbp = new Db(&myEnv, 0);

        // Open the database. Note that we are using auto commit for
        // the open, so the database is able to support transactions.
        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

        Dbt key, data;
        key.set_data(keystr);
        key.set_size((strlen(keystr) + 1) * sizeof(char));
        key.set_data(datastr);
        key.set_size((strlen(datastr) + 1) * sizeof(char));

        DbTxn *txn = NULL;
        myEnv.txn_begin(NULL, &txn, 0);
        try {
            db->put(txn, &key, &data, 0);
            txn->commit(0);
        } catch (DbException &e) {
            std::cerr << "Error in transaction: "
                      << e.what() << std::endl;
            txn->abort();
        }

    } catch(DbException &e) {
        std::cerr << "Error opening database and environment: "
                  << file_name << ", "
                  << envHome << std::endl;
        std::cerr << e.what() << std::endl;
    }

    try {
        if (dbp != NULL)
            dbp->close(0);
        myEnv.close(0);
    } catch(DbException &e) {
        std::cerr << "Error closing database and environment: "
                  << file_name << ", "
                  << 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 DB is doing with the logging subsystem. Logging causes all database 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 databases in the event of a system or application failure, so by performing logging, DB ensures the integrity of your data.

Moreover, DB performs *write-ahead* logging. This means that information is written to the logs *before* the actual database 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 database 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:

- 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 19) 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 66).
- A commit record is written to the log files. This indicates that the modifications made by the transaction are now permanent.
- 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 42) for more information).

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

Notice that committing a transaction does not necessarily cause data modified in your memory cache to be written to the files backing your databases 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 50).

- 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 database 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 database 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 `DbEnv::set_flags()` method. This causes DB 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 DB'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 DbTxn::commit() method.

- Specify DB_TXN_WRITE_NOSYNC using the DbEnv::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 66) 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.

Note that aborting a transaction may result in disk I/O if your logs are backed by the filesystem. It is possible that during the course of your transaction, logging data and/or database pages were written to backing files on disk. For this reason, DB notes that the abort occurred in its log files so that at a minimum the database can be brought into a consistent state at recovery time.

Also, 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 DbTxn::abort().

Auto Commit

While transactions are frequently used to provide atomicity to multiple database operations, it is sometimes necessary to perform a single database 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 databases so that they support transactions. See Enabling Transactions (page 6) for details.

   Note that frequently auto commit is used for the environment or database open. To use auto commit for either your environment or database open, specify `DB_AUTO_COMMIT` to the `DbEnv::set_flags()` or `Db::open()` method. If you specify auto commit for the environment open, then you do not need to also specify auto commit for the database open.

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

Note that auto commit is not available for cursors. You must always open your cursor using a transaction if you want the cursor's operations to be transactional protected. See Transactional Cursors (page 23) for details on using transactional cursors.

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

```
#include "db_cxx.h"

...

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";
    const char *keystr ="thekey";
    const char *datastr = "thedata";

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

    try {

        myEnv.open(envHome.c_str(), env_flags, 0);
        dbp = new Db(&myEnv, 0);

        // Open the database. Note that we are using auto commit for
        // the open, so the database is able to support transactions.
        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

        Dbt key, data;
        key.set_data(keystr);
        key.set_size((strlen(keystr) + 1) * sizeof(char));
        key.set_data(datastr);
        key.set_size((strlen(datastr) + 1) * sizeof(char));

        // Perform the write. Because the database was opened to support
        // auto commit, this write is performed using auto commit.
        db->put(NULL, &key, &data, 0);

    } catch(DbException &e) {
        std::cerr << "Error opening database and environment: "
                  << file_name << ", "
                  << envHome << std::endl;
        std::cerr << e.what() << std::endl;
    }

    try {
        if (dbp != NULL)
            dbp->close(0);
        myEnv.close(0);
    } catch(DbException &e) {
        std::cerr << "Error closing database and environment: "
                  << file_name << ", "
                  << 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 database 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 database 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, simply pass the parent transaction's handle when you created the nested transaction's handle. For example:

```
    // parent transaction
    DbTxn *parentTxn, *childTxn;
    ret = myEnv.txn_begin(NULL, &parentTxn, 0);
    // child transaction
    ret = myEnv.txn_begin(parent_txn, &childTxn, 0);
```

Transactional Cursors

You can transaction-protect your cursor operations by specifying a transaction handle at the time that you create your cursor. Beyond that, you do not ever provide a transaction handle directly to a cursor method.

Note that if you transaction-protect a cursor, then you must make sure that the cursor is closed before you either commit or abort the transaction. For example:

```
#include "db_cxx.h"

...

int main(void)
{
    // Environment and database opens omitted
    ...
```

```
    DbTxn *txn = NULL;
    Dbc *cursorp = NULL;

    try {

        Dbt key, data;
        key.set_data(keystr);
        key.set_size((strlen(keystr) + 1) * sizeof(char));
        key.set_data(datastr);
        key.set_size((strlen(datastr) + 1) * sizeof(char));

        DbTxn *txn = NULL;
        myEnv.txn_begin(NULL, &txn, 0);
        try {
            // Get our cursor. Note that we pass the transaction handle here.
            db.cursor(txn, &cursorp, 0);

            // Perform our operations. Note that we do not pass a transaction
            // handle here.
            char *replacementString = "new string";
            while (cursor->get(&key, &data, DB_NEXT) == 0) {
                data.set_data(void *)replacementString);
                data.set_size((strlen(replacementString) + 1) * sizeof(char));
                cursor->put(&key, &data, DB_CURRENT);
            }

            // We're done. Commit the transaction.
            cursor->close();
            txn->commit(0);
        } catch (DbException &e) {
            std::cerr << "Error in transaction: "
                      << e.what() << std::endl;
            cursor->close();
            txn->abort();
        }

    } catch(DbException &e) {
        std::cerr << "Error opening database and environment: "
                  << file_name << ", "
                  << envHome << std::endl;
        std::cerr << e.what() << std::endl;
    }


    return (EXIT_SUCCESS);
}
```

Secondary Indices with Transaction Applications

You can use transactions with your secondary indices so long as you open the secondary index so that it supports transactions (that is, you wrap the database open in a transaction, or use auto commit, in the same way as when you open a primary transactional database). In addition, you must make sure that when you associate the secondary index with the primary database, the association is performed using a transaction. The easiest thing to do here is to simply specify `DB_AUTO_COMMIT` when you perform the association.

All other aspects of using secondary indices with transactions are identical to using secondary indices without transactions. In addition, transaction-protecting cursors opened against secondary indices is performed in exactly the same way as when you use transactional cursors against a primary database. See Transactional Cursors (page 23) for details.

Note that when you use transactions to protect your database writes, your secondary indices are protected from corruption because updates to the primary and the secondaries are performed in a single atomic transaction.

For example:

```
#include <db_cxx.h>

...

// Environment and primary database open omitted
...

Db my_index(&envp, 0);    // Secondary

// Open the secondary
my_index.open(NULL,              // Transaction pointer
              "my_secondary.db", // On-disk file that holds the database.
              NULL,              // Optional logical database name
              DB_BTREE,          // Database access method
              DB_AUTO_COMMIT,    // Open flags.
              0);                // File mode (using defaults)

// Now associate the primary and the secondary
my_database.associate(NULL,             // Txn id
                      &my_index,        // Associated secondary database
                      get_sales_rep,    // Callback used for key extraction.
                                        // This is described in the Getting
                                        // Started guide.
                      DB_AUTO_COMMIT);  // Flags
```

Configuring the Transaction Subsystem

Most of the configuration activities that you need to perform for your transactional DB application will involve the locking and logging subsystems. See Concurrency (page 28) and Managing DB Files (page 50) for details.

However, there are a couple of things that you can do to configure your transaction subsystem directly. These things are:

- Configure the maximum number of simultaneous transactions needed by your application. In general, you should not need to do this unless you use deeply nest transactions or have many threads all of which have active transactions.

  By default, your application can support 20 active transactions.

  You can set the maximum number of simultaneous transactions supported by your application using the `DbEnv::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.

- Configure the timeout value for your transactions. This value represents the longest period of time a transaction can be active. Note, however, that transaction timeouts are checked only when DB examines its lock tables for blocked locks (see Locks, Blocks, and Deadlocks (page 29) for more information). Therefore, a transaction's timeout can have expired, but the application will not be notified until DB has a reason to examine its lock tables.

  Be aware that some transactions may be inappropriately timed out before the transaction has a chance to complete. You should therefore use this mechanism only if you know your application might have unacceptably long transactions and you want to make sure your application will not stall during their execution. (This might happen if, for example, your transaction blocks or requests too much data.)

  To set the maximum timeout value for your transactions, use the `DbEnv::set_timeout()` method. This method configures the entire environment; not just the handle used to set the configuration. Further, this value may be set at any time during the application's lifetime.

  This value can also be set using the `DB_CONFIG` file's `set_txn_timeout` parameter.

For example:

```
#include "db_cxx.h"

...
```

```
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

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

    try {

        // Configure a maximum transaction timeout of 1 second.
        myEnv.set_timeout(1000000, DB_SET_TXN_TIMEOUT);
        // Configure 40 maximum transactions.
        myEnv.set_tx_max(40);
        myEnv.open(envHome.c_str(), env_flags, 0);

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


        ...
```

Chapter 4. Concurrency

DB offers a great deal of support for multi-threaded and multi-process applications even when transactions are not in use. Many of DB's handles are thread-safe, or can be made thread-safe by providing the appropriate flag at handle creation time, and DB provides a flexible locking subsystem for managing databases in a concurrent application. Further, DB 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 DB operations.

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

  Also, DB 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 DB supports both exclusive and non-exclusive locks. See Locks (page 29) 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 DB objects, see Which DB Handles are Free-Threaded (page 29).

- *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 31) 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 may 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 34).

Which DB Handles are Free-Threaded

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

- `DbEnv`

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

- `Db`

  Free-threaded so long as the `DB_THREAD` flag is provided to the database `open()` method, or if the database is opened using a free-threaded environment handle.

- `Dbc`

  Cursors are not free-threaded. However, they can be used by multiple threads of control so long as the application serializes access to the handle.

- `DbTxn`

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

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 DB 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 DB 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 DB 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 Db handle.

2. The lock.

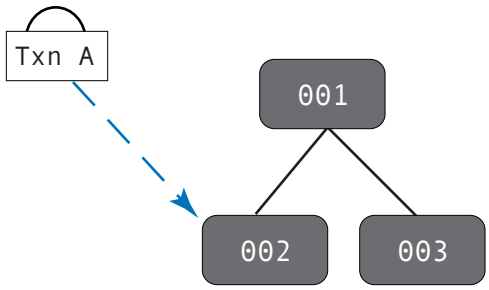
   This is the actual data structure that locks the object. In DB, a locked object structure in the lock manager is representative of the object that is locked.

3. The locked object.

   The thing that your application actually wants to lock. In a DB application, the locked object is usually a database page, which in turn contains multiple database entries (key and data). However, for Queue databases, individual database records are locked.

You can configure how many total lockers, locks, and locked objects your application is allowed to support. See Configuring the Locking Subsystem (page 35) 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

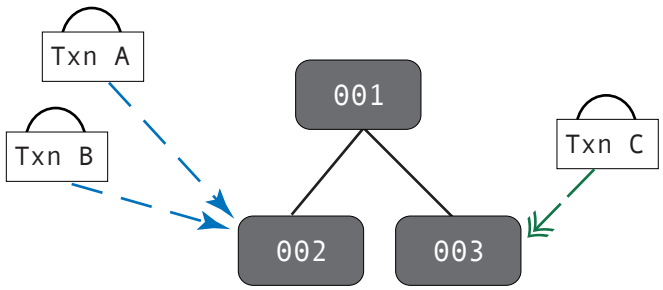
DB applications support both exclusive and non-exclusive locks. *Exclusive locks* are granted when a locker wants to write to an object. For this reason, exclusive locks are also sometimes called *write locks*.

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.

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.

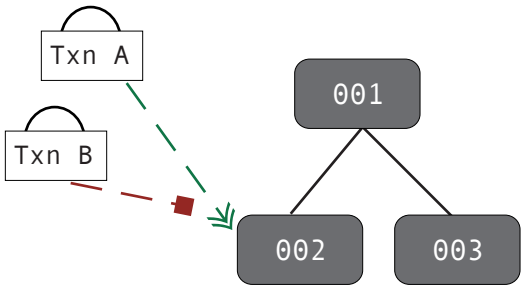
2. All non-transaction operations hold write locks until such a time as the write is completed. So if you are using a database handle to write to a database, the underlying write lock is held until that write is completed.

Blocks

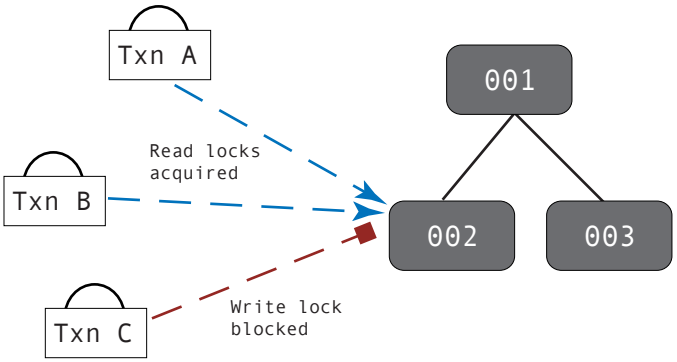
Simply put, a thread of control is blocked when it attempts to obtain a lock, but that attempt is denied because some other thread of control holds a conflicting lock. Once blocked, the thread of control is temporarily unable to make any forward progress until the requested lock is obtained or the operation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



The previous description describes DB's default behavior when it cannot obtain a lock. It is possible to configure DB transactions so that they will not block. Instead, if a lock is unavailable, the application is immediately notified of a deadlock situation. See No Wait on Blocks (page 47) for more information.

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 a database record, 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 DB 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 cursor handle is closed. 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 cursor is closed).

  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 40).

- 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.

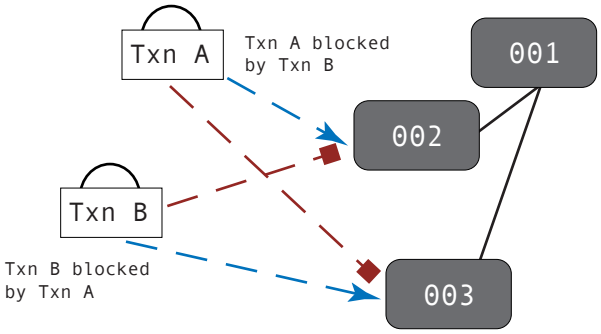
It is possible to configure DB's transactions so that they never wait on blocked lock requests. Instead, if they are blocked on a lock request, they will notify the application of a deadlock (see the next section).

You configure this behavior on a transaction by transaction basis. See No Wait on Blocks (page 47) 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 abandoned the attempted operation in order to resolve the deadlock. DB's locking subsystem offers a deadlock notification mechanism. See Configuring Deadlock Detection (page 37) 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 33)). Beyond that, you can also do the following in order to avoid deadlocks:

- 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).
- If you are using BTrees in which you are constantly adding and then deleting data, turn Btree reverse split off. See Reverse BTree Splits (page 48) for more information.
- 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 DB 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 47).

The Locking Subsystem

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

This section describes the locking subsystem as it is used to protect DB resources. In particular, issues on configuration are examined here. For information on using the locking subsystem to manage non-DB 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 `DbEnv::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 `DbEnv::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 DbEnv::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 DbEnv::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 30).

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

```
#include "db_cxx.h"

...

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

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

    try {

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

        myEnv.open(envHome.c_str(), env_flags, 0);

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

```
        std::cerr << e.what() << std::endl;
        return (EXIT_FAILURE);
    }

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

    return (EXIT_SUCCESS);
}
```

Configuring Deadlock Detection

In order for DB 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 DB to internally detect deadlocks as they occur.

   To do this, you use `DbEnv::set_lk_detect()`. This method causes DB to walk its internal lock table looking for a deadlock whenever a lock request is blocked. This method also identifies how DB decides which lock requests are rejected when deadlocks are detected. For example, DB 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 DB 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 database operations beyond deadlock detection.

   To externally perform lock detection, you can use either the `DbEnv::lock_detect()` method, or use the **db_deadlock** command line utility. This method (or command) causes DB to walk the lock table looking for deadlocks.

   Note that like `DbEnv::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 `DbEnv::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 DB 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 DB checks the lock table for deadlocks every time a lock request is blocked:

```
#include "db_cxx.h"

...

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

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

```
    try {

        // 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(envHome.c_str(), env_flags, 0);

        // From here, you open your databases, proceed with your
        // database 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.sleepycat.com/docs/utility/db_deadlock.html]

Resolving Deadlocks

When DB determines that a deadlock has occurred, it will select a thread of control to resolve the deadlock and then throws `DbDeadlockException` in that thread. When this happens, the thread must:

1. Cease all read and write operations.
2. Close all open cursors.
3. Abort the transaction.
4. 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 database calls using the handle that has deadlocked.

For example:

```
// 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.
```

```
// txn is a transaction handle.
// key and data are DBT handles. Their usage is not shown here.
while (retry_count < MAX_DEADLOCK_RETRIES) {
    try {
        envp->txn_begin(NULL, txn, 0);
        dbp->put(txn, &key, &data, 0);
        txn->commit(0);
    } catch (DbDeadlockException &de) {
        try {
            // Abort the transaction and increment the
            // retry counter
            txn->abort();
            retry_count++;
            // If we've retried too many times, log it and exit
            if (retry_count >= MAX_DEADLOCK_RETRIES) {
                    envp->errx("Exceeded retry limit. Giving up.");
                    return (EXIT_FAILURE);
            }
        } catch (DbException &ae) {
            envp->err(ae.get_errno(), "txn abort failed.");
            return (EXIT_FAILURE);
        }
    } catch (DbException &e) {
        try {
            // For a generic error, log it and abort.
            envp->err(e.get_errno(), "Error putting data.");
            txn->abort();
        } catch (DbException &ae) {
            envp->err(ae.get_errno(), "txn abort failed.");
            return (EXIT_FAILURE);
        }
    }
}

return (EXIT_SUCCESS);
```

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

Sleepycat supports the following levels of isolation. They are:

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 database.
2	READ COMMITTED	Committed read isolation means that degree 1 is observed, plus 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 DB's default isolation guarantee.

By default, DB transactions and transactional cursors offer serializable isolation. You can optionally reduce your isolation level by configuring DB to use uncommitted read isolation.

See Reading Uncommitted Data (page 42) for more information. You can also configure DB to use committed read isolation. See Committed Reads (page 43) for more information.

Reading Uncommitted Data

You can 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.

When used with transactions, uncommitted reads means that one transaction can see data modified but not yet committed by another transaction. When used with transactional cursors, uncommitted reads means that any database reader can see data modified by the cursor before the cursor's transaction has committed.

Because of this, uncommitted reads allow a transaction to read data that may subsequently be aborted by another transaction. In this case, the reading transaction will have read data that never really existed in the database.

To configure your application to read uncommitted data:

1. Open your database such that it will allow uncommitted reads. You do this by specifying `DB_READ_UNCOMMITTED` when you open your database.

2. Specify `DB_READ_UNCOMMITTED` when you create the transaction, open the cursor, or read a record from the database.

For example, the following opens the database such that it supports uncommitted reads, and then creates a transaction that causes all reads performed within it to use uncommitted reads. Remember that simply opening the database to support uncommitted reads is not enough; you must also declare your read operations to be performed using uncommitted reads.

```
#include "db_cxx.h"

...

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

    u_int32_t db_flags = DB_CREATE |           // Create the db if it does
                                               // not exist
```

```
                        DB_AUTO_COMMIT |      // Enable auto commit
                        DB_READ_UNCOMMITTED;  // Enable uncommitted reads

    Db *dbp = NULL;
    const char *file_name = "mydb.db";
    const char *keystr ="thekey";
    const char *datastr = "thedata";

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

    try {

        myEnv.open(envHome.c_str(), env_flags, 0);
        dbp = new Db(&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

        DbTxn *txn = NULL;
        myEnv.txn_begin(NULL, &txn, DB_READ_UNCOMMITTED);

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

        ...
```

Committed Reads

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 DB 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 "db_cxx.h"

...

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

    // Notice that we do not have to specify any flags to the database to
    // allow committed reads (this is as opposed to uncommitted reads
    // where we DO have to specify a flag on the database open.
    u_int32_t db_flags = DB_CREATE | DB_AUTO_COMMIT;
    Db *dbp = NULL;
    const char *file_name = "mydb.db";

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

    try {

        myEnv.open(envHome.c_str(), env_flags, 0);
        dbp = new Db(&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

        DbTxn *txn = NULL;

        // Open the transaction and enable committed reads. All cursors
```

```
        // open with this transaction handle will use read committed
        // isolation.
        myEnv.txn_begin(NULL, &txn, DB_READ_COMMITTED);

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

        // Using transactional cursors with concurrent applications is
        // described in more detail in the following section.

        ...
```

Transactional Cursors and Concurrent Applications

When you use transactional cursors with a concurrent application, remember that in the event of a deadlock you must make sure that you close your cursor before you abort and retry your transaction.

Also, remember that when you are using the default isolation level, every time your cursor reads a record it locks that record until the encompassing transaction is resolved. This means that walking your database with a transactional cursor increases the chance of lock contention.

For this reason, if you must routinely walk your database with a transactional cursor, consider using a reduced isolation level such as read committed.

Using Cursors with Uncommitted Data

As described in Reading Uncommitted Data (page 42) above, it is possible to relax your transaction's isolation level such that it can read data modified but not yet committed by another transaction. You can configure this when you create your transaction handle, and when you do so then all cursors opened inside that transaction will automatically use uncommitted reads.

You can also do this when you create a cursor handle from within a serializable transaction. When you do this, only those cursors configured for uncommitted reads uses uncommitted reads.

Either way, you must first configure your database handle to support uncommitted reads before you can configure your transactions or your cursors to use them.

The following example shows how to configure an individual cursor handle to read uncommitted data from within a serializable (full isolation) transaction. For an example of configuring a transaction to perform uncommitted reads in general, see Reading Uncommitted Data (page 42).

```
#include "db_cxx.h"

...
```

```
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 |           // Create the db if it does
                                               // not exist
                         DB_AUTO_COMMIT |      // Enable auto commit
                         DB_READ_UNCOMMITTED;  // Enable uncommitted reads

    Db *dbp = NULL;
    const char *file_name = "mydb.db";

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

    Dbc *cursorp = NULL;

    try {

        myEnv.open(envHome.c_str(), env_flags, 0);
        dbp = new Db(&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

        DbTxn *txn = NULL;
        myEnv.txn_begin(NULL, &txn, 0);
        try {
            // Get our cursor. Note that we pass the transaction
            // handle here. Note also that we pass the
            // DB_READ_UNCOMMITTED flag here so as to cause the
            // cursor to perform uncommitted reads.
            db.cursor(txn, &cursorp, DB_READ_UNCOMMITTED);

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

            ...
```

Read/Modify/Write

If you are retrieving a record from the database for the purpose of modifying or deleting it, you should declare a read-modify-write cycle at the time that you read the record. Doing so causes DB to obtain a write lock (instead of a read lock) 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 database or cursor get method.

For example:

```
// Begin the deadlock retry loop as is normal.
while (retry_count < MAX_DEADLOCK_RETRIES) {
    try {
        envp->txn_begin(NULL, txn, 0);

        ...
        // key and data are DBTs. Their usage is omitted for brevity.
        ...

        // Read the data. Declare the read/modify/write cycle here
        dbp->get(txn, &key, &data, DB_RMW);

        ...
        // Modify the data as is required. (not shown here)
        ...

        // Put the data. Note that you do not have to provide any
        // additional flags here due to the read/modify/write
        // cycle. Simply put the data and perform your deadlock
        // detection as normal.
        dbp->put(txn, &key, &data, 0);
        txn->commit(0);
    } catch (DbDeadlockException &de) {
        // Deadlock detection and exception handling omitted
        // for brevity
        ...
```

No Wait on Blocks

Normally when a DB 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 `DbEnv::txn_begin()` method.

For example:

```
    DbTxn *txn = NULL;
    try {
        envp->txn_begin(NULL, &txn, DB_TXN_NOWAIT);

        ...
    } catch (DbException &de) {
        // Deadlock detection and exception handling omitted
        // for brevity
        ...
```

Reverse BTree Splits

If your application is using the Btree access method, and your application is repeatedly deleting then adding records to your database, then you might be able to reduce lock contention by turning off reverse Btree splits.

As pages are emptied in a database, DB attempts to delete empty pages in order to keep the database as small as possible and minimize search time. Moreover, when a page in the database fills up, DB, of course, adds additional pages to make room for more data.

Adding and deleting pages in the database requires that the writing thread lock the parent page. Consequently, as the number of pages in your database diminishes, your application will see increasingly more lock contention; the maximum level of concurrency in a database of two pages is far smaller than that in a database of 100 pages, because there are fewer pages that can be locked.

Therefore, if you prevent the database from being reduced to a minimum number of pages, you can improve your application's concurrency throughput. Note, however, that you should do so only if your application tends to delete and then add the same data. If this is not the case, then preventing reverse Btree splits can harm your database search time.

To turn off reverse Btree splits, provide the `DB_REVSPLITOFF` flag to the `Db::set_flags()` method.

For example:

```
#include "db_cxx.h"

...

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 locking
                          DB_INIT_MPOOL |  // Initialize the cache
                          DB_THREAD     |  // Free-thread the env handle
                          DB_INIT_TXN;     // Initialize transactions

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

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

    try {

        myEnv.open(envHome.c_str(), env_flags, 0);
        dbp = new Db(&myEnv, 0);

        // Turn off BTree reverse split.
        dbp=>set_flags(DB_REVSPLITOFF);

        dbp->open(dbp,        // Pointer to the database
                  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

    } catch(DbException &e) {
        std::cerr << "Error opening database and environment: "
                  << file_name << ", " << envHome << std::endl;
        std::cerr << e.what() << std::endl;
    }

    try {
        dbp->close(dbp, 0);
        myEnv.close(0);
    } catch(DbException &e) {
        std::cerr << "Error closing database and environment: "
                  << file_name << ", " << envHome << std::endl;
        std::cerr << e.what() << std::endl;
        return (EXIT_FAILURE);
    }

    return (EXIT_SUCCESS);
}
```

Chapter 5. Managing DB Files

DB is capable of storing several types of files on disk:

- Data files, which contain the actual data in your database.
- 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.
- Temporary files, which are created only under certain special circumstances. These files never need to be backed up or otherwise managed and so they are not a consideration for the topics described in this chapter. See Security Considerations (page 11) for more information on temporary files.

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 DB file management, we need to describe checkpoints. When databases are modified (that is, a transaction is committed), the modifications are recorded in DB's logs, but they are *not* necessarily reflected in the actual database 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 database files on disk. Note that a checkpoint also causes data dirtied by an uncommitted transaction to also be written to your database files on disk. In this latter case, DB'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 55).

- 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 databases.

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 `DbEnv::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 DbEnv::txn_checkpoint() API reference page. [http://www.sleepycat.com/docs/api_cxx/txn_checkpoint.html]

Note that running checkpoints can be quite expensive. DB must flush every dirty page to the backing database 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 DB applications.

For example, to run a checkpoint from a separate thread of control:

```
#include <pthread.h>
#include "db_cxx.h"

...

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

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

```
    try {

        myEnv.open(envHome.c_str(), env_flags, 0);

        // 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));
                myEnv.close(0);
                exit (1);
        }

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

        ...
}

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

    // Checkpoint once a minute.
    for (;; sleep(60)) {
        try {
            dbenv->txn_checkpoint(0, 0, 0));
        } catch(DbException &e) {
            dbenv->err(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 DB databases 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 databases from the backup. See Catastrophic Recovery (page 57) for more information.

Note that you can also maintain a hot failover. See Using Hot Failovers (page 62) 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 databases.

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 databases 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 database page size.

Finally, if you plan to use a system utility to copy database 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 database 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 50) for details.
4. Copy all your database files to the backup location. Note that you can simply copy all of the database files, or you can determine which database files have been written during the lifetime of the current logs. To do this, use either the `DbEnv::log_archive()` method with the `DB_ARCH_DATA` option, or use the **db_archive** command with the `-s` option.

   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 `log_archive()` 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 hotbackup 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 database files to the backup location. Note that you can simply copy all of the database files, or you can determine which database files have been written during the lifetime of the current logs. To do this, use either the `DbEnv::log_archive()` with the `DB_ARCH_DATA` option, or use the **db_archive** command with the `-s` option.
2. Copy all logs to your backup location.

It is important to copy your database files *and then* your logs. In this way, you can complete or roll back any database 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 database 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 databases change and how sensitive your application is to lengthy recoveries (should one be required).

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

Recovery Procedures

DB 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 databases 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 databases from a previously created backup.

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

Catastrophic recovery is run whenever you have lost or corrupted your database 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 62) for more information).

Normal Recovery

Normal recovery examines the contents of your environment's log files, and uses this information to ensure that your database 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.

Sleepycat recommends that you 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 "db_cxx.h"

...

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

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

    try {

        myEnv.open(envHome.c_str(), env_flags, 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 databases from a previously created backup. Note that to restore your databases 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 database 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 "db_cxx.h"

...

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

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

try {

    myEnv.open(envHome.c_str(), env_flags, 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.
```

Designing Your Application for Recovery

When building your DB 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 DB operations. We illustrate this with our final example in this book (see Transaction Example (page 70) 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 database 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 database 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 55) for more information.

4. Restart all your database 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 DB activities and close its environment. If you simply run recovery against the environment, your database threads will detect this and fail in the midst of performing their database 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, DB applications use multiple processes to interact with the databases. 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 databases. 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 database 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 databases 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 database administrative utility that always runs recovery.

DB offers you two methods by which you can manage recovery for multi-process DB 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 database operations.

When this happens, the current database operation will abnormally fail, indicating a DB_RUNRECOVERY condition. This means that your application should immediately abandon any database 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. Note that you must also specify DB_RECOVER or DB_RECOVER_FATAL for your environment open. If during the open, DB determines that

recovery must be run, this indicates the type of recovery that is run. If you do not specify either type of recovery, 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`.

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

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.

3. You can not use this mechanism with the `failchk()` mechanism described in the next section.

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, Sleepycat 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 Sleepycat 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. DB 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 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 DbEnv::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 DbEnv::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.

Note that this mechanism should not be mixed with the process registration method of multi-process recovery described in the previous section.

Using Hot Failovers

You can maintain a hot 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 databases as normal, but use the backup environment instead of the production environment.

Removing Log Files

By default DB does not delete log files for you. For this reason, DB'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.

DB provides several mechanisms to remove log files that meet all but the last criteria (DB 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 `DbEnv::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 52). 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 50) for more information.
3. If you are maintaining a hot backup, perform the hot backup procedure as described in Using Hot Failovers (page 62).
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 66).
- Maintain logs entirely in-memory. See Configuring In-Memory Logging (page 66) for more information.
- Size of the log buffer in memory. See Setting the In-Memory Log Buffer Size (page 68).
- 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), DB 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 `DbEnv::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 databases as if to prepare for catastrophic failure. See Backup Procedures (page 52) 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's 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 DB's log manager.

You can set the size of your logging region by using the `DbEnv::set_lg_region()` 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 databases.

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 68) 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_INMEMORY` to the `DbEnv::set_flags()` method. Note that you must specify this before your application opens its first environment handle.

For example:

```
#include "db_cxx.h"

...

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

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

    try {

        // 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(envHome.c_str(), env_flags, 0);

    } catch(DbException &e) {
        std::cerr << "Error opening database and environment: "
                  << file_name << ", "
                  << envHome << std::endl;
        std::cerr << e.what() << std::endl;
    }

    // From here, you open databases, create transactions and
    // perform database 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 66)), 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 `DbEnv::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 DB 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 database files are consistent relative to the information found in your log files.
3. 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 DB's built-in deadlock detector.
4. Open whatever database handles that you need.
5. Spawn off worker threads. How many of these you need and how they split their DB workload is entirely up to your application's requirements. However, any worker threads that perform write operations against your databases will do the following:
   a. Begin a transaction.
   b. Perform one or more read and write operations against your databases.
   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.
6. On application shutdown:
   a. Make sure there are no opened cursors.

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

c. Close your databases

d. Close your environment.

Robust applications should monitor their database 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 50) for details.
- Periodically back up your database and log files. This is required in order to fully obtain the durability guarantee made by DB's transaction ACID support. See Backup Procedures (page 52) 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 62) for more information.

Transaction Example

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

The example opens an environment and database and then creates 5 threads, each of which writes 500 records to the database. The keys used for these writes are pre-determined strings, while the data is a random value. This means that the actual data is arbitrary and therefore uninteresting; we picked it only because it requires minimum code to implement and therefore will stay out of the way of the main points of this example.

Each thread writes 10 records under a single transaction before committing and writing another 10 (this is repeated 50 times). At the end of each transaction, but before committing, each thread calls a function that uses a cursor to read every record in the database. We do this in order to make some points about database reads in a transactional environment.

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:

```
// File TxnGuide.cpp

// We assume an ANSI-compatible compiler
#include <db_cxx.h>
#include <pthread.h>
#include <iostream>

#ifdef _WIN32
extern int getopt(int, char * const *, const char *);
#else
#include <unistd.h>
#endif
```

We also need a directive that we use to identify how many threads we want our program to create:

```
// Run 5 writers threads at a time.
#define NUMWRITERS 5
```

Next we declare a couple of global variables (used by our threads), and we provide our forward declarations for the functions used by this example.

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

// Forward declarations
int countRecords(Db *, DbTxn *);
int openDb(Db **, const char *, const char *, DbEnv *, u_int32_t);
int usage(void);
void *writerThread(void *);
```

We now implement our usage function, which identifies our only command line parameter:

```
// Usage function
int
usage()
{
    std::cerr << " [-h <database_home_directory>]" << std::endl;
    return (EXIT_FAILURE);
}
```

With that, we have finished up our program's housekeeping, and we can now move on to the main part of our program. As usual, we begin with `main()`. First we declare all our variables, and then we initialize our DB handles.

```
int
main(int argc, char *argv[])
{
    // Initialize our handles
    Db *dbp = NULL;
    DbEnv *envp = NULL;

    pthread_t writerThreads[NUMWRITERS];
    int ch, i;
    u_int32_t envFlags;
    char *dbHomeDir;

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

    // Database file name
    const char *fileName = "mydb.db";
```

Now we need to parse our command line. In this case, all we want is to know where our environment directory is. If the `-h` option is not provided when this example is run, the current working directory is used instead.

```
    // Parse the command line arguments
#ifdef _WIN32
    dbHomeDir = ".\\";
#else
    dbHomeDir = "./";
#endif
    while ((ch = getopt(argc, argv, "h:")) != EOF)
        switch (ch) {
        case 'h':
            dbHomeDir = optarg;
            break;
        case '?':
        default:
            return (usage());
        }
```

Next we create our database handle, and we define our environment open flags. There are a few things to notice here:

- We specify `DB_RECOVER`, which means that normal recovery is run every time we start the application. This is highly desirable and recommended by Sleepycat for most applications.
- We also specify `DB_THREAD`, which means our environment handle will be free-threaded. This is very important because we will be sharing the environment handle across threads.

```
    // 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. This
                       // also turns on logging.
      DB_INIT_MPOOL |  // Initialize the memory pool (in-memory cache)
      DB_THREAD;       // Cause the environment to be free-threaded

    try {
        // Create and open the environment
        envp = new DbEnv(0);
```

Now we configure how we want deadlock detection performed. In our case, we will cause DB to perform deadlock detection by walking its internal lock tables looking for a block every time a lock is requested. Further, in the event of a deadlock, the thread that holds the youngest lock will receive the deadlock notification.

```
        // Indicate that we want db 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(DB_LOCK_MINWRITE);
```

Now we open our environment.

```
        // 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.

        envp->open(dbHomeDir, envFlags, 0);
```

Now we call the function that will open our database for us. This is not very interesting, except that you will notice that we are specifying `DB_DUPSORT`. This is required purely by the data that we are writing to the database, and it is only necessary if you run the application more than once without first deleting the environment.

The implementation of `open_db()` is described later in this section.

```
        // Open the database
        openDb(&dbp, progName, fileName, envp, DB_DUPSORT);
```

Now we create our threads. In this example we are using pthreads for our threading package. A description of threading (beyond how it impacts DB usage) is beyond the scope of this manual. However, the things that we are doing here should be familiar to anyone who has prior experience with any threading package. We are simply initializing a mutex, creating our threads, and then joining our threads, which causes our program to wait until the joined threads have completed before continuing operations in the main thread.

```
        // Initialize a pthread mutex. Used to help provide thread ids.
        (void)pthread_mutex_init(&thread_num_lock, NULL);

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

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

    } catch(DbException &e) {
        std::cerr << "Error opening database environment: "
                  << dbHomeDir << std::endl;
        std::cerr << e.what() << std::endl;
        return (EXIT_FAILURE);
    }
```

Finally, to wrap up `main()`, we close out our database and environment handle, as is normal for any DB application. Notice that this is where our `err` label is placed in our application. If any database operation prior to this point in the program returns an error status, the program simply jumps to this point and closes our handles if necessary before exiting the application completely.

```
    try {
        // Close our database handle if it was opened.
        if (dbp != NULL)
            dbp->close(0);

        // Close our environment if it was opened.
        if (envp != NULL)
            envp->close(0);
    } catch(DbException &e) {
        std::cerr << "Error closing database and environment."
                  << std::endl;
        std::cerr << e.what() << std::endl;
        return (EXIT_FAILURE);
    }

    // Final status message and return.

    std::cout << "I'm all done." << std::endl;
    return (EXIT_SUCCESS);
}
```

Now that we have completed `main()`, we need to implement the function that our writer threads will actually run. This is where the bulk of our transactional code resides.

We start as usual with variable declarations and initialization.

```
// A function that performs a series of writes to a
// Berkeley DB database. The information written
// to the database is largely nonsensical, but the
// mechanisms of transactional commit/abort and
// deadlock detection are illustrated here.
void *
writerThread(void *args)
{
    int j, thread_num;
    int max_retries = 20;   // Max retry on a deadlock
    char *key_strings[] = {"key 1", "key 2", "key 3", "key 4",
                           "key 5", "key 6", "key 7", "key 8",
                           "key 9", "key 10"};

    Db *dbp = (Db *)args;
    DbEnv *envp = dbp->get_env();
```

Now we want a thread number for reporting purposes. It is possible to use the `pthread_t` value directly for this purpose, but how that is done unfortunately differs depending on the pthread implementation you are using. So instead we use a mutex-protected global variable to obtain a simple integer for our reporting purposes.

Note that we are also use this thread id for initializing a random number generator, which we do here. We use this random number generator for data generation.

```
    // 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);

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

Now we begin the loop that we use to write data to the database. Notice that in this top loop, we begin a new transaction. We will actually use 50 transactions per writer thread, although we will only ever have one active transaction per thread at a time. Within each transaction, we will perform 10 database writes.

By combining multiple writes together under a single transaction, we increase the likelihood that a deadlock will occur. Normally, you want to reduce the potential for a deadlock and in this case the way to do that is to perform a single write per transaction. To avoid deadlocks, we could be using auto commit to write to our database for this workload.

However, we want to show deadlock handling and by performing multiple writes per transaction we can actually observe deadlocks occurring. We also want to underscore the idea that you can combing multiple database operations together in a single atomic unit of work in order to improve the efficiency of your writes.

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

                // Begin our transaction. We group multiple writes in
                // this thread under a single transaction so as to
                // (1) show that you can atomically perform multiple
                // writes at a time, and (2) to increase the chances
                // of a deadlock occurring so that we can observe our
                // deadlock detection at work.

                // Normally we would want to avoid the potential for
                // deadlocks, so for this workload the correct thing
                // would be to perform our puts with auto commit. But
                // that would excessively simplify our example, so we
                // do the "wrong" thing here instead.
                txn = NULL;
                envp->txn_begin(NULL, &txn, 0);
```

Now we begin the inner loop that we use to actually perform the write.

```
                // Perform the database write for this transaction.
                for (j = 0; j < 10; j++) {
                    Dbt key, value;
                    key.set_data(key_strings[j]);
                    key.set_size((strlen(key_strings[j]) + 1) *
                        sizeof(char));

                    int payload = rand() + i;
                    value.set_data(&payload);
                    value.set_size(sizeof(int));

                    // Perform the database put
                    dbp->put(txn, &key, &value, 0);
                }
```

Having completed the inner database write loop, we could simply commit the transaction and continue on to the next block of 10 writes. However, we want to first illustrate a few points about transactional processing so instead we call our `countRecords()` function before calling the transaction commit. `countRecords()` uses a cursor to read every record in the database and return a count of the number of records that it found.

```
                // countRecords runs a cursor over the entire database.
                // We do this to illustrate issues of deadlocking
                std::cout << thread_num <<  " : Found "
                          <<  countRecords(dbp, NULL)
                          << " records in the database." << std::endl;

                std::cout << thread_num <<  " : committing txn : " << i
                          << std::endl;

                // commit
                try {
                    txn->commit(0);
                    retry = false;
                    txn = NULL;
                } catch (DbException &e) {
                    std::cout << "Error on txn commit: "
                              << e.what() << std::endl;
                }
```

Finally, we finish our try block. Notice how we examine the exceptions to determine whether we need to abort (or abort/retry in the case of a deadlock) our current transaction.

```
            } catch (DbDeadlockException &de) {
                // First thing we MUST do is abort the transaction.
                if (txn != NULL)
                    (void)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.\n"
                              << "Retrying write operation."
                              << std::endl;
                    retry_count++;
                    retry = true;
                 } else {
                    // Otherwise, just give up.
                    std::cerr << "Writer " << thread_num
                              << ": Got DeadLockException and out of "
                              << "retries. Giving up." << std::endl;
                    retry = false;
                 }
            } catch (DbException &e) {
                std::cerr << "db put failed" << std::endl;
                std::cerr << e.what() << std::endl;
                if (txn != NULL)
```

```
                    txn->abort();
                retry = false;
            } catch (std::exception &ee) {
             std::cerr << "Unknown exception: " << ee.what() << std::endl;
             return (0);
            }
        }
    }
    return (0);
}
```

We want to back up for a moment and take a look at the call to `countRecords()`. If you look at the `countRecords()` function prototype at the beginning of this example, you will see that the function's second parameter takes a transaction handle. However, our usage of the function here does not pass a transaction handle through to the function.

Because `countRecords()` reads every record in the database, if used incorrectly the thread will self-deadlock. The writer thread has just written 500 records to the database, but because the transaction used for that write has not yet been committed, each of those 500 records are still locked by the thread's transaction. If we then simply run a non-transactional cursor over the database from within the same thread that has locked those 500 records, the cursor will block when it tries to read one of those transactional protected records. The thread immediately stops operation at that point while the cursor waits for the read lock it has requested. Because that read lock will never be released (the thread can never make any forward progress), this represents a self-deadlock for the the thread.

There are three ways to prevent this self-deadlock:

1. We can move the call to `countRecords()` to a point after the thread's transaction has committed.

2. We can allow `countRecords()` to operate under the same transaction as all of the writes were performed (this is what the transaction parameter for the function is for).

3. We can reduce our isolation guarantee for the application by allowing uncommitted reads.

For this example, we choose to use option 3 (uncommitted reads) to avoid the deadlock. This means that we have to open our database such that it supports uncommitted reads, and we have to open our cursor handle so that it knows to perform uncommitted reads.

Note that in In-Memory Transaction Example (page 80), we simply perform the cursor operation using the same transaction as is used for the thread's writes.

The following is the `countRecords()` implementation. There is not anything particularly interesting about this function other than specifying uncommitted reads when we open the cursor handle, but we include the function here anyway for the sake of completeness.

```
// This simply counts the number of records contained in the
// database and returns the result.
//
// Note that this method exists only for illustrative purposes.
// A more straight-forward way to count the number of records in
// a database is to use the Database.getStats() method.
int
countRecords(Db *dbp, DbTxn *txn)
{

    Dbc *cursorp = NULL;
    int count = 0;

    try {
        // Get the cursor
        dbp->cursor(txn, &cursorp, DB_READ_UNCOMMITTED);

        Dbt key, value;
        while (cursorp->get(&key, &value, DB_NEXT) == 0) {
            count++;
        }
    } catch (DbDeadlockException &de) {
        std::cerr << "countRecords: got deadlock" << std::endl;
        cursorp->close();
        throw de;
    } catch (DbException &e) {
        std::cerr << "countRecords error:" << std::endl;
        std::cerr << e.what() << std::endl;
    }

    if (cursorp != NULL) {
        try {
            cursorp->close();
        } catch (DbException &e) {
            std::cerr << "countRecords: cursor close failed:" << std::endl;
            std::cerr << e.what() << std::endl;
        }
    }

    return (count);
}
```

Finally, we provide the implementation of our `openDb()` function. This function should hold no surprises for you. Note, however, that we do specify uncommitted reads when we open the database. If we did not do this, then our `countRecords()` function would cause our thread to self-deadlock because the cursor could not be opened to support uncommitted reads (that flag on the cursor open would, in fact, be silently ignored by DB).

```
// Open a Berkeley DB database
int
openDb(Db **dbpp, const char *progname, const char *fileName,
  DbEnv *envp, u_int32_t extraFlags)
{
    int ret;
    u_int32_t openFlags;

    try {
        Db *dbp = new Db(envp, 0);

        // Point to the new'd Db
        *dbpp = dbp;

        if (extraFlags != 0)
            ret = dbp->set_flags(extraFlags);

        // Now open the database
        openFlags = DB_CREATE              | // Allow database creation
                    DB_READ_UNCOMMITTED    | // Allow uncommitted reads
                    DB_AUTO_COMMIT;          // Allow auto commit

        dbp->open(NULL,       // Txn pointer
                  fileName,   // File name
                  NULL,       // Logical db name
                  DB_BTREE,   // Database type (using btree)
                  openFlags,  // Open flags
                  0);         // File mode. Using defaults
    } catch (DbException &e) {
        std::cerr << progname << "open_db: db open failed:" << std::endl;
        std::cerr << e.what() << std::endl;
        return (EXIT_FAILURE);
    }

    return (EXIT_SUCCESS);
}
```

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 DB distribution:

```
DB_INSTALL/examples_cxx/txn_guide
```

In-Memory Transaction Example

DB is sometimes used for applications that simply need to cache data retrieved from some other location (such as a remote database server). DB is also often used in embedded systems.

In both cases, applications may want to use transactions for atomicity, consistency, and isolation guarantees, but they may also want to forgo the durability guarantee entirely. In doing so, they can keep their DB environment and databases 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.
- 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.
- Do not specify a file name when you open your database(s).

As an example, this section takes the transaction example provided in Transaction Example (page 70) and it updates that example so that the environment, database, log files, and regions are all kept entirely in-memory.

For illustration purposes, we also modify this example so that uncommitted reads are no longer used to enable the `countRecords()` function. Instead, we simply provide a transaction handle to `countRecords()` so as to avoid the self-deadlock. Be aware that using a transaction handle here rather than uncommitted reads will work just as well as if we had continued to use uncommitted reads. However, the usage of the transaction handle here will probably cause more deadlocks than using read-uncommitted does, because more locking is being performed in this case.

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 and include the `getopt` function. We can also remove our `usage()` function because we no longer require any command line arguments.

```
// File TxnGuideInMemory.cpp

// We assume an ANSI-compatible compiler
#include <db_cxx.h>
#include <pthread.h>
#include <iostream>

// Run 5 writers threads at a time.
#define NUMWRITERS 5

// Printing of pthread_t is implementation-specific, so we
```

```
// create our own thread IDs for reporting purposes.
int global_thread_num;
pthread_mutex_t thread_num_lock;

// Forward declarations
int countRecords(Db *, DbTxn *);
int openDb(Db **, const char *, const char *, DbEnv *, u_int32_t);
int usage(void);
void *writerThread(void *);
```

Next, in our `main()`, we also eliminate some variables that this example no longer needs. In particular, we are able to remove the `dbHomeDir` and `fileName` variables. We also remove all our `getopt` code.

```
int
main(void)
{
    // Initialize our handles
    Db *dbp = NULL;
    DbEnv *envp = NULL;

    pthread_t writerThreads[NUMWRITERS];
    int i;
    u_int32_t envFlags;

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

Next we create our environment as always. However, we add `DB_PRIVATE` to our environment open flags. This flag causes our environment to back regions using our application's heap memory rather than by using the filesystem. This is the first important step to keeping our DB data entirely in-memory.

We also remove the `DB_RECOVER` flag from the environment open flags. Because our databases, logs, and regions are maintained in-memory, there will never be anything to recover.

Note that we show the additional code here in **bold**.

```
    // Env open flags
    envFlags =
      DB_CREATE     |  // Create the environment if it does not exist
      DB_INIT_LOCK  |  // Initialize the locking subsystem
      DB_INIT_LOG   |  // Initialize the logging subsystem
      DB_INIT_TXN   |  // Initialize the transactional subsystem. This
                       // also turns on logging.
      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
```

```
    try {
        // Create the environment
        envp = new DbEnv(0);
```

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.

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

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

        // Specify the size of the in-memory cache
        envp->set_cachesize(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 db 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(DB_LOCK_MINWRITE);

        // Open the environment
        envp->open(NULL, envFlags, 0);
```

When we call `openDb()`, which is what we use to open our database, we no not provide a database filename for the third parameter. When the filename is `NULL`, the database is not backed by the filesystem.

```
        // 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.

        // Open the database
        openDb(&dbp, progName, NULL,
            envp, DB_DUPSORT);
```

After that, our `main()` function is unchanged, except that when we check for exceptions on the database open, we change the error message string so as to not reference the database filename.

```
        // Initialize a pthread mutex. Used to help provide thread ids.
        (void)pthread_mutex_init(&thread_num_lock, NULL);
```

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

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

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

    try {
        // Close our database handle if it was opened.
        if (dbp != NULL)
            dbp->close(0);

        // Close our environment if it was opened.
        if (envp != NULL)
            envp->close(0);
    } catch(DbException &e) {
        std::cerr << "Error closing database and environment."
                  << std::endl;
        std::cerr << e.what() << std::endl;
        return (EXIT_FAILURE);
    }

    // Final status message and return.

    std::cout << "I'm all done." << std::endl;
    return (EXIT_SUCCESS);
}
```

That completes `main()`. The bulk of our `writerThread()` function implementation is unchanged from the initial transaction example, except that we now pass `countRecords` a transaction handle, rather than configuring our application to perform uncommitted reads. Both mechanisms work well-enough for preventing a self-deadlock. However, the individual count in this example will tend to be lower than the counts seen in the previous transaction example, because `countRecords()` can no longer see records created but not yet committed by other threads.

```
// A function that performs a series of writes to a
// Berkeley DB database. The information written
// to the database is largely nonsensical, but the
// mechanism of transactional commit/abort and
// deadlock detection is illustrated here.
void *
writerThread(void *args)
{
    Db *dbp = (Db *)args;
    DbEnv *envp = dbp->get_env(dbp);

    int j, thread_num;
    int max_retries = 20;   // Max retry on a deadlock
    char *key_strings[] = {"key 1", "key 2", "key 3", "key 4",
                           "key 5", "key 6", "key 7", "key 8",
                           "key 9", "key 10"};

    // 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);

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

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

                // Begin our transaction. We group multiple writes in
                // this thread under a single transaction so as to
                // (1) show that you can atomically perform multiple
                // writes at a time, and (2) to increase the chances
                // of a deadlock occurring so that we can observe our
                // deadlock detection at work.

                // Normally we would want to avoid the potential for
                // deadlocks, so for this workload the correct thing
                // would be to perform our puts with auto commit. But
                // that would excessively simplify our example, so we
                // do the "wrong" thing here instead.
```

```
                txn = NULL;
                envp->txn_begin(NULL, &txn, 0);
                // Perform the database write for this transaction.
                for (j = 0; j < 10; j++) {
                    Dbt key, value;
                    key.set_data(key_strings[j]);
                    key.set_size((strlen(key_strings[j]) + 1) *
                        sizeof(char));

                    int payload = rand() + i;
                    value.set_data(&payload);
                    value.set_size(sizeof(int));

                    // Perform the database put
                    dbp->put(txn, &key, &value, 0);
                }

                // countRecords runs a cursor over the entire database.
                // We do this to illustrate issues of deadlocking
                std::cout << thread_num <<  " : Found "
                          <<  countRecords(dbp, txn)
                          << " records in the database." << std::endl;

                std::cout << thread_num <<  " : committing txn : " << i
                          << std::endl;

                // commit
                try {
                    txn->commit(0);
                    retry = false;
                    txn = NULL;
                } catch (DbException &e) {
                    std::cout << "Error on txn commit: "
                              << e.what() << std::endl;
                }
            } catch (DbDeadlockException &de) {
                // First thing we MUST do is abort the transaction.
                if (txn != NULL)
                    (void)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.\n"
                              << "Retrying write operation."
                              << std::endl;
                    retry_count++;
```

```
                    retry = true;
                } else {
                    // Otherwise, just give up.
                    std::cerr << "Writer " << thread_num
                              << ": Got DeadLockException and out of "
                              << "retries. Giving up." << std::endl;
                    retry = false;
                }
            } catch (DbException &e) {
                std::cerr << "db put failed" << std::endl;
                std::cerr << e.what() << std::endl;
                if (txn != NULL)
                    txn->abort();
                retry = false;
            } catch (std::exception &ee) {
                std::cerr << "Unknown exception: " << ee.what() << std::endl;
                return (0);
            }
        }
    }
    return (0);
}
```

Next we update `countRecords()`. The only difference here is that we no longer specify `DB_READ_UNCOMMITTED` when we open our cursor. Note that even this minor change is not required. If we do not configure our database to support uncommitted reads, `DB_READ_UNCOMMITTED` on the cursor open will be silently ignored. However, we remove the flag anyway from the cursor open so as to avoid confusion.

```
int
countRecords(Db *dbp, DbTxn *txn)
{

    Dbc *cursorp = NULL;
    int count = 0;

    try {
        // Get the cursor
        dbp->cursor(txn, &cursorp, 0);

        Dbt key, value;
        while (cursorp->get(&key, &value, DB_NEXT) == 0) {
            count++;
        }
    } catch (DbDeadlockException &de) {
        std::cerr << "countRecords: got deadlock" << std::endl;
        cursorp->close();
        throw de;
    } catch (DbException &e) {
```

```
        std::cerr << "countRecords error:" << std::endl;
        std::cerr << e.what() << std::endl;
    }

    if (cursorp != NULL) {
        try {
            cursorp->close();
        } catch (DbException &e) {
            std::cerr << "countRecords: cursor close failed:" << std::endl;
            std::cerr << e.what() << std::endl;
        }
    }

    return (count);
}
```

Finally, we update `openDb()`. This involves removing `DB_READ_UNCOMMITTED` from the open flags.

```
// Open a Berkeley DB database
int
openDb(Db **dbpp, const char *progname, const char *fileName,
  DbEnv *envp, u_int32_t extraFlags)
{
    int ret;
    u_int32_t openFlags;

    try {
        Db *dbp = new Db(envp, 0);

        // Point to the new'd Db
        *dbpp = dbp;

        if (extraFlags != 0)
            ret = dbp->set_flags(extraFlags);

        // Now open the database
        openFlags = DB_CREATE   |      // Allow database creation
                    DB_THREAD        |
                    DB_AUTO_COMMIT;    // Allow auto commit

        dbp->open(NULL,       // Txn pointer
                  fileName,   // File name
                  NULL,       // Logical db name
                  DB_BTREE,   // Database type (using btree)
                  openFlags,  // Open flags
                  0);         // File mode. Using defaults
    } catch (DbException &e) {
        std::cerr << progname << ": openDb: db open failed:" << std::endl;
```

```
        std::cerr << e.what() << std::endl;
        return (EXIT_FAILURE);
    }

    return (EXIT_SUCCESS);
}
```

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

```
DB_INSTALL/examples_cxx/txn_guide
```