Oracle Berkeley DB

Getting Started with Berkeley DB for C++

Release 4.8

Legal Notice

Published 8/14/2009

Table of Contents

Preface

Welcome to Berkeley DB (DB). This document introduces DB, version 4.8. It is intended to provide a rapid introduction to the DB API set and related concepts. The goal of this document is to provide you with an efficient mechanism with which you can evaluate DB against your project's technical requirements. As such, this document is intended for C++ developers and senior software architects who are looking for an in-process data management solution. No prior experience with Berkeley DB is expected or required.

Conventions Used in this Book

The following typographical conventions are used within in this manual:

Class names are represented in `monospaced font`, as are `method names`. For example: "`Db::open()` is a `Db` 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 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 DB application:

- Getting Started with Transaction Processing for C++ [http://www.oracle.com/technology/documentation/berkeley-db/db/gsg_txn/CXX/BerkeleyDB-Core-Cxx-Txn.pdf]
- Berkeley DB Getting Started with Replicated Applications for C++ [http://www.oracle.com/technology/documentation/berkeley-db/db/gsg_db_rep/CXX/Replication_CXX_GSG.pdf]
- Berkeley DB Programmer's Reference Guide [http://www.oracle.com/technology/documentation/berkeley-db/db/programmer_reference/BDB_Prog_Reference.pdf]
- Berkeley DB C++ API [http://www.oracle.com/technology/documentation/berkeley-db/db/api_reference/CXX/BDB-CXX_APIReference.pdf]

Chapter 1. Introduction to Berkeley DB

Welcome to Berkeley DB (DB). DB is a general-purpose embedded database engine that is capable of providing a wealth of data management services. It is designed from the ground up for high-throughput applications requiring in-process, bullet-proof management of mission-critical data. DB can gracefully scale from managing a few bytes to terabytes of data. For the most part, DB is limited only by your system's available physical resources.

You use DB through a series of programming APIs which give you the ability to read and write your data, manage your database(s), and perform other more advanced activities such as managing transactions.

Because DB is an embedded database engine, it is extremely fast. You compile and link it into your application in the same way as you would any third-party library. This means that DB runs in the same process space as does your application, allowing you to avoid the high cost of interprocess communications incurred by stand-alone database servers.

To further improve performance, DB offers an in-memory cache designed to provide rapid access to your most frequently used data. Once configured, cache usage is transparent. It requires very little attention on the part of the application developer.

Beyond raw speed, DB is also extremely configurable. It provides several different ways of organizing your data in its databases. Known as *access methods*, each such data organization mechanism provides different characteristics that are appropriate for different data management profiles. (Note that this manual focuses almost entirely on the BTree access method as this is the access method used by the vast majority of DB applications).

To further improve its configurability, DB offers many different subsystems, each of which can be used to extend DB's capabilities. For example, many applications require write-protection of their data so as to ensure that data is never left in an inconsistent state for any reason (such as software bugs or hardware failures). For those applications, a transaction subsystem can be enabled and used to transactional-protect database writes.

The list of operating systems on which DB is available is too long to detail here. Suffice to say that it is available on all major commercial operating systems, as well as on many embedded platforms.

Finally, DB is available in a wealth of programming languages. DB is officially supported in C, C++, and Java, but the library is also available in many other languages, especially scripting languages such as Perl and Python.

Before going any further, it is important to mention that DB is not a relational database (although you could use it to build a relational database). Out of the box, DB does not provide higher-level features such as triggers, or a high-level query language such as SQL. Instead, DB provides just those minimal APIs required to store and retrieve your data as efficiently as possible.

About This Manual

This manual introduces DB. As such, this book does not examine intermediate or advanced features such as threaded library usage or transactional usage. Instead, this manual provides a step-by-step introduction to DB's basic concepts and library usage.

Specifically, this manual introduces DB environments, databases, database records, and storage and retrieval of database records. This book also introduces cursors and their usage, and it describes secondary databases.

For the most part, this manual focuses on the BTree access method. A chapter is given at the end of this manual that describes some of the concepts involving BTree usage, such as duplicate record management and comparison routines.

Examples are given throughout this book that are designed to illustrate API usage. At the end of each chapter, a complete example is given that is designed to reinforce the concepts covered in that chapter. In addition to being presented in this book, these final programs are also available in the DB software distribution. You can find them in

```
DB_INSTALL/examples_cxx/getting_started
```

where *DB_INSTALL* is the location where you placed your DB distribution.

This book uses the C++ programming languages for its examples. Note that versions of this book exist for the C and Java languages as well.

Berkeley DB Concepts

Before continuing, it is useful to describe some of the larger concepts that you will encounter when building a DB application.

Conceptually, DB databases contain *records*. Logically each record represents a single entry in the database. Each such record contains two pieces of information: a key and a data. This manual will on occasion describe a *a record's key* or a *record's data* when it is necessary to speak to one or the other portion of a database record.

Because of the key/data pairing used for DB databases, they are sometimes thought of as a two-column table. However, data (and sometimes keys, depending on the access method) can hold arbitrarily complex data. Frequently, C structures and other such mechanisms are stored in the record. This effectively turns a 2-column table into a table with *n* columns, where *n-1* of those columns are provided by the structure's fields.

Note that a DB database is very much like a table in a relational database system in that most DB applications use more than one database (just as most relational databases use more than one table).

Unlike relational systems, however, a DB database contains a single collection of records organized according to a given access method (BTree, Queue, Hash, and so forth). In a relational database system, the underlying access method is generally hidden from you.

In any case, frequently DB applications are designed so that a single database stores a specific type of data (just as in a relational database system, a single table holds entries containing a specific set of fields). Because most applications are required to manage multiple kinds of data, a DB application will often use multiple databases.

For example, consider an accounting application. This kind of an application may manage data based on bank accounts, checking accounts, stocks, bonds, loans, and so forth. An accounting application will also have to manage information about people, banking institutions, customer accounts, and so on. In a traditional relational database, all of these different kinds of information would be stored and managed using a (probably very) complex series of tables. In a DB application, all of this information would instead be divided out and managed using multiple databases.

DB applications can efficiently use multiple databases using an optional mechanism called an *environment*. For more information, see Environments (page 6).

You interact with most DB APIs using special structures that contain pointers to functions. These callbacks are called *methods* because they look so much like a method on a C++ class. The variable that you use to access these methods is often referred to as a *handle*. For example, to use a database you will obtain a handle to that database.

Retrieving a record from a database is sometimes called *getting the record* because the method that you use to retrieve the records is called `get()`. Similarly, storing database records is sometimes called *putting the record* because you use the `put()` method to do this.

When you store, or put, a record to a database using its handle, the record is stored according to whatever sort order is in use by the database. Sorting is mostly performed based on the key, but sometimes the data is considered too. If you put a record using a key that already exists in the database, then the existing record is replaced with the new data. However, if the database supports duplicate records (that is, records with identical keys but different data), then that new record is stored as a duplicate record and any existing records are not overwritten.

If a database supports duplicate records, then you can use a database handle to retrieve only the first record in a set of duplicate records.

In addition to using a database handle, you can also read and write data using a special mechanism called a *cursor*. Cursors are essentially iterators that you can use to walk over the records in a database. You can use cursors to iterate over a database from the first record to the last, and from the last to the first. You can also use cursors to seek to a record. In the event that a database supports duplicate records, cursors are the only way you can access all the records in a set of duplicates.

Finally, DB provides a special kind of a database called a *secondary database*. Secondary databases serve as an index into normal databases (called primary database to distinguish them from secondaries). Secondary databases are interesting because DB records can hold complex data types, but seeking to a given record is performed only based on that record's key. If you wanted to be able to seek to a record based on some piece of information that is not the key, then you enable this through the use of secondary databases.

Access Methods

While this manual will focus primarily on the BTree access method, it is still useful to briefly describe all of the access methods that DB makes available.

Note that an access method can be selected only when the database is created. Once selected, actual API usage is generally identical across all access methods. That is, while some exceptions exist, mechanically you interact with the library in the same way regardless of which access method you have selected.

The access method that you should choose is gated first by what you want to use as a key, and then secondly by the performance that you see for a given access method.

The following are the available access methods:

Access Method	Description
BTree	Data is stored in a sorted, balanced tree structure. Both the key and the data for BTree records can be arbitrarily complex. That is, they can contain single values such as an integer or a string, or complex types such as a structure. Also, although not the default behavior, it is possible for two records to use keys that compare as equals. When this occurs, the records are considered to be duplicates of one another.
Hash	Data is stored in an extended linear hash table. Like BTree, the key and the data used for Hash records can be of arbitrarily complex data. Also, like BTree, duplicate records are optionally supported.
Queue	Data is stored in a queue as fixed-length records. Each record uses a logical record number as its key. This access method is designed for fast inserts at the tail of the queue, and it has a special operation that deletes and returns a record from the head of the queue. This access method is unusual in that it provides record level locking. This can provide beneficial performance improvements in applications requiring concurrent access to the queue.
Recno	Data is stored in either fixed or variable-length records. Like Queue, Recno records use logical record numbers as keys.

Selecting Access Methods

To select an access method, you should first consider what you want to use as a key for you database records. If you want to use arbitrary data (even strings), then you should use either BTree or Hash. If you want to use logical record numbers (essentially integers) then you should use Queue or Recno.

Once you have made this decision, you must choose between either BTree or Hash, or Queue or Recno. This decision is described next.

Choosing between BTree and Hash

For small working datasets that fit entirely in memory, there is no difference between BTree and Hash. Both will perform just as well as the other. In this situation, you might just as well use BTree, if for no other reason than the majority of DB applications use BTree.

Note that the main concern here is your working dataset, not your entire dataset. Many applications maintain large amounts of information but only need to access some small portion of that data with any frequency. So what you want to consider is the data that you will routinely use, not the sum total of all the data managed by your application.

However, as your working dataset grows to the point where you cannot fit it all into memory, then you need to take more care when choosing your access method. Specifically, choose:

- BTree if your keys have some locality of reference. That is, if they sort well and you can expect that a query for a given key will likely be followed by a query for one of its neighbors.
- Hash if your dataset is extremely large. For any given access method, DB must maintain a certain amount of internal information. However, the amount of information that DB must maintain for BTree is much greater than for Hash. The result is that as your dataset grows, this internal information can dominate the cache to the point where there is relatively little space left for application data. As a result, BTree can be forced to perform disk I/O much more frequently than would Hash given the same amount of data.

  Moreover, if your dataset becomes so large that DB will almost certainly have to perform disk I/O to satisfy a random request, then Hash will definitely out perform BTree because it has fewer internal records to search through than does BTree.

Choosing between Queue and Recno

Queue or Recno are used when the application wants to use logical record numbers for the primary database key. Logical record numbers are essentially integers that uniquely identify the database record. They can be either mutable or fixed, where a mutable record number is one that might change as database records are stored or deleted. Fixed logical record numbers never change regardless of what database operations are performed.

When deciding between Queue and Recno, choose:

- Queue if your application requires high degrees of concurrency. Queue provides record-level locking (as opposed to the page-level locking that the other access methods use), and this can result in significantly faster throughput for highly concurrent applications.

  Note, however, that Queue provides support only for fixed length records. So if the size of the data that you want to store varies widely from record to record, you should probably choose an access method other than Queue.
- Recno if you want mutable record numbers. Queue is only capable of providing fixed record numbers. Also, Recno provides support for databases whose permanent storage is a flat text file. This is useful for applications looking for fast, temporary storage while the data is being read or modified.

Database Limits and Portability

Berkeley DB provides support for managing everything from very small databases that fit entirely in memory, to extremely large databases holding millions of records and terabytes of data. DB databases can store up to 256 terabytes of data. Individual record keys or record data can store up to 4 gigabytes of data.

DB's databases store data in a binary format that is portable across platforms, even of differing endian-ness. Be aware, however, that portability aside, some performance issues can crop up in the event that you are using little endian architecture. See Setting Comparison Functions (page 73) for more information.

Also, DB's databases and data structures are designed for concurrent access — they are thread-safe, and they share well across multiple processes. That said, in order to allow multiple processes to share databases and the cache, DB makes use of mechanisms that do not work well on network-shared drives (NFS or Windows networks shares, for example). For this reason, you cannot place your DB databases and environments on network-mounted drives.

Environments

This manual is meant as an introduction to the Berkeley DB library. Consequently, it describes how to build a very simple, single-threaded application and so this manual omits a great many powerful aspects of the DB database engine that are not required by simple applications. One of these is important enough that it warrants a brief overview here: environments.

While environments are frequently not used by applications running in embedded environments where every byte counts, they will be used by virtually any other DB application requiring anything other than the bare minimum functionality.

An *environment* is essentially an encapsulation of one or more databases. You open an environment and then you open databases in that environment. When you do so, the databases are created/located in a location relative to the environment's home directory.

Environments offer a great many features that a stand-alone DB database cannot offer:

- Multi-database files.

  It is possible in DB to contain multiple databases in a single physical file on disk. This is desirable for those application that open more than a few handful of databases. However, in order to have more than one database contained in a single physical file, your application *must* use an environment.

- Multi-thread and multi-process support

  When you use an environment, resources such as the in-memory cache and locks can be shared by all of the databases opened in the environment. The environment allows you to enable subsystems that are designed to allow multiple threads and/or processes to access DB databases. For example, you use an environment to enable the concurrent data store (CDS), the locking subsystem, and/or the shared memory buffer pool.

- Transactional processing

  DB offers a transactional subsystem that allows for full ACID-protection of your database writes. You use environments to enable the transactional subsystem, and then subsequently to obtain transaction IDs.

- High availability (replication) support

  DB offers a replication subsystem that enables single-master database replication with multiple read-only copies of the replicated data. You use environments to enable and then manage this subsystem.

- Logging subsystem

  DB offers write-ahead logging for applications that want to obtain a high-degree of recoverability in the face of an application or system crash. Once enabled, the logging subsystem allows the application to perform two kinds of recovery ("normal" and "catastrophic") through the use of the information contained in the log files.

For more information on these topics, see the *Berkeley DB Getting Started with Transaction Processing* guide and the *Berkeley DB Getting Started with Replicated Applications* guide.

Exception Handling

Before continuing, it is useful to spend a few moments on exception handling in DB with the C++ API.

By default, most DB methods throw `DbException` in the event of a serious error. However, be aware that `DbException` does not inherit from `std::exception` so your `try` blocks should catch both types of exceptions. For example:

```
#include <db_cxx.h>
    ...
try
{
    // DB and other code goes here
}
catch(DbException &e)
{
  // DB error handling goes here
}
catch(std::exception &e)
{
    // All other error handling goes here
}
```

You can obtain the DB error number for a `DbException` by using `DbException::get_errno()`. You can also obtain the informational message associated with that error number using `DbException::what()`.

If for some reason you do not want to manage `DbException` objects in your `try` blocks, you can configure DB to suppress them by setting `DB_CXX_NO_EXCEPTIONS` for your database and environment handles. In this event, you must manage your DB error conditions using the integer value returned by all DB methods. Be aware that this manual assumes that you want to manage your error conditions using `DbException` objects.

Error Returns

In addition to exceptions, the DB interfaces always return a value of 0 on success. If the operation does not succeed for any reason, the return value will be non-zero.

If a system error occurred (for example, DB ran out of disk space, or permission to access a file was denied, or an illegal argument was specified to one of the interfaces), DB returns an `errno` value. All of the possible values of `errno` are greater than 0.

If the operation did not fail due to a system error, but was not successful either, DB returns a special error value. For example, if you tried to retrieve data from the database and the record for which you are searching does not exist, DB would return `DB_NOTFOUND`, a special error value that means the requested key does not appear in the database. All of the possible special error values are less than 0.

Getting and Using DB

You can obtain DB by visiting the Berkeley DB download page: http://www.oracle.com/technology/software/products/berkeley-db/db/index.html.

To install DB, untar or unzip the distribution to the directory of your choice. You will then need to build the product binaries. For information on building DB, see *DB_INSTALL*`/docs/index.html`, where *DB_INSTALL* is the directory where you unpacked DB. On that page, you will find links to platform-specific build instructions.

That page also contains links to more documentation for DB. In particular, you will find links for the *Berkeley DB Programmer's Reference Guide* as well as the API reference documentation.

Chapter 2. Databases

In Berkeley DB, a database is a collection of *records*. Records, in turn, consist of key/data pairings.

Conceptually, you can think of a database as containing a two-column table where column 1 contains a key and column 2 contains data. Both the key and the data are managed using `Dbt` class instances (see Database Records (page 20) for details on this class). So, fundamentally, using a DB database involves putting, getting, and deleting database records, which in turns involves efficiently managing information encapsulated by `Dbt` objects. The next several chapters of this book are dedicated to those activities.

Opening Databases

You open a database by instantiating a `Db` object and then calling its `open()` method.

Note that by default, DB does not create databases if they do not already exist. To override this behavior, specify the `DB_CREATE` flag on the `open()` method.

The following code fragment illustrates a database open:

```
#include <db_cxx.h>

...

Db db(NULL, 0);              // Instantiate the Db object

u_int32_t oFlags = DB_CREATE; // Open flags;

try {
    // Open the database
    db.open(NULL,                // Transaction pointer
            "my_db.db",          // Database file name
            NULL,                // Optional logical database name
            DB_BTREE,            // Database access method
            oFlags,              // Open flags
            0);                  // File mode (using defaults)
// DbException is not subclassed from std::exception, so
// need to catch both of these.
} catch(DbException &e) {
    // Error handling code goes here
} catch(std::exception &e) {
    // Error handling code goes here
}
```

Closing Databases

Once you are done using the database, you must close it. You use the `Db::close()` method to do this.

Closing a database causes it to become unusable until it is opened again. Note that you should make sure that any open cursors are closed before closing your database. Active cursors during a database close can cause unexpected results, especially if any of those cursors are writing to the database. You should always make sure that all your database accesses have completed before closing your database.

Cursors are described in Using Cursors (page 33) later in this manual.

Be aware that when you close the last open handle for a database, then by default its cache is flushed to disk. This means that any information that has been modified in the cache is guaranteed to be written to disk when the last handle is closed. You can manually perform this operation using the `Db::sync()` method, but for normal shutdown operations it is not necessary. For more information about syncing your cache, see Data Persistence (page 23).

The following code fragment illustrates a database close:

```
#include <db_cxx.h>

...

Db db(NULL, 0);

 // Database open and access operations happen here.

try {
    // Close the database
    db.close(0);
// DbException is not subclassed from std::exception, so
// need to catch both of these.
} catch(DbException &e) {
    // Error handling code goes here
} catch(std::exception &e) {
    // Error handling code goes here
}
```

Database Open Flags

The following are the flags that you may want to use at database open time. Note that this list is not exhaustive — it includes only those flags likely to be of interest for introductory, single-threaded database applications. For a complete list of the flags available to you, see the *Berkeley DB C++ API guide*.

To specify more than one flag on the call to `Db::open()`, you must bitwise inclusively OR them together:

```
u_int32_t open_flags = DB_CREATE | DB_EXCL;
```

- `DB_CREATE`

  If the database does not currently exist, create it. By default, the database open fails if the database does not already exist.

- `DB_EXCL`

  Exclusive database creation. Causes the database open to fail if the database already exists. This flag is only meaningful when used with `DB_CREATE`.

- `DB_RDONLY`

  Open the database for read operations only. Causes any subsequent database write operations to fail.

- `DB_TRUNCATE`

  Physically truncate (empty) the on-disk file that contains the database. Causes DB to delete all databases physically contained in that file.

Administrative Methods

The following `Db` methods may be useful to you when managing DB databases:

- `Db::get_open_flags()`

  Returns the current open flags. It is an error to use this method on an unopened database.

```
#include <db_cxx.h>
...
Db db(NULL, 0);
u_int32_t open_flags;

// Database open and subsequent operations omitted for clarity

db.get_open_flags(&open_flags);
```

- `Db::remove()`

Removes the specified database. If no value is given for the *database* parameter, then the entire file referenced by this method is removed.

Never remove a database that has handles opened for it. Never remove a file that contains databases with opened handles.

```
#include <db_cxx.h>
...
Db db(NULL, 0);

// Database handle creation omitted for clarity

db.remove("mydb.db",             // Database file to remove
          NULL,                  // Database to remove. This is
                                 // NULL so the entire file is
                                 // removed.
         0);                     // Flags. None used.
```

- `Db::rename()`

Renames the specified database. If no value is given for the *database* parameter, then the entire file referenced by this method is renamed.

Never rename a database that has handles opened for it. Never rename a file that contains databases with opened handles.

```
#include <db_cxx.h>
...
Db db(NULL, 0);

// Database handle creation omitted for clarity

db.rename("mydb.db",             // Database file to rename
          NULL,                  // Database to rename. This is
                                 // NULL so the entire file is
                                 // renamed.
         "newdb.db",             // New database file name
         0);                     // Flags. None used.
```

Error Reporting Functions

To simplify error reporting and handling, the `Db` class offers several useful methods.

- `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 for any error messages issued by the DB library.

- `err()`

  Issues an error message. 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_errpfx()`), an optional `printf`-style formatted message, the error message, and a trailing newline.

- `errx()`

  Behaves identically to `err()` except that the DB message text associated with the supplied error value is not appended to the error string.

In addition, you can use the `db_strerror()` function to directly return the error string that corresponds to a particular error number.

For example, to send all error messages for a given database handle to a callback for handling, first create your callback. Do something like this:

```
/*
 * Function called to handle any database error messages
 * issued by DB.
 */
void
my_error_handler(const char *error_prefix, char *msg)
{
  /*
```

```
    * Put your code to handle the error prefix and error
    * message here. Note that one or both of these parameters
    * may be NULL depending on how the error message is issued
    * and how the DB handle is configured.
    */
}
```

And then register the callback as follows:

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

Db db(NULL, 0);
std::string dbFileName("my_db.db");

try
{
    // Set up error handling for this database
    db.set_errcall(my_error_handler);
    db.set_errpfx("my_example_program");
```

And to issue an error message:

```
    // Open the database
    db.open(NULL, dbFileName.c_str(), NULL, DB_BTREE, DB_CREATE, 0);
}
    // Must catch both DbException and std::exception
    catch(DbException &e)
    {
        db.err(e.get_errno(), "Database open failed %s",
            dbFileName.c_str());
        throw e;
    }
    catch(std::exception &e)
    {
        // No DB error number available, so use errx
        db.errx("Error opening database: %s", e.what());
        throw e;
    }
```

Managing Databases in Environments

In Environments (page 6), we introduced environments. While environments are not used in the example built in this book, they are so commonly used for a wide class of DB applications that it is necessary to show their basic usage, if only from a completeness perspective.

To use an environment, you must first open it. At open time, you must identify the directory in which it resides. This directory must exist prior to the open attempt. You can also identify open properties, such as whether the environment can be created if it does not already exist.

You will also need to initialize the in-memory cache when you open your environment.

For example, to create an environment handle and open an environment:

```
#include <db_cxx.h>
...
u_int32_t env_flags = DB_CREATE |     // If the environment does not
                                      // exist, create it.
                      DB_INIT_MPOOL; // Initialize the in-memory cache.

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;
    exit( -1 );
} catch(std::exception &e) {
    std::cerr << "Error opening database environment: "
              << envHome << std::endl;
    std::cerr << e.what() << std::endl;
    exit( -1 );
}
```

Once an environment is opened, you can open databases in it. Note that by default databases are stored in the environment's home directory, or relative to that directory if you provide any sort of a path in the database's file name:

```
#include <db_cxx.h>
...
u_int32_t env_flags = DB_CREATE |  // If the environment does not
                                   // exist, create it.
                      DB_INIT_MPOOL; // Initialize the in-memory cache.
std::string envHome("/export1/testEnv");

u_int32_t db_flags = DB_CREATE;   // If the database does not
                                  // exist, create it.
```

```
std::string dbName("mydb.db");
DbEnv myEnv(0);
Db *myDb;

try {
    myEnv.open(envHome.c_str(), env_flags, 0);
    myDb = new Db(&myEnv, 0);
    myDb->open(NULL,
               dbName.c_str(),
               NULL,
               DB_BTREE,
               db_flags,
               0);
} catch(DbException &e) {
    std::cerr << "Error opening database environment: "
              << envHome
              << " and database "
              << dbName << std::endl;
    std::cerr << e.what() << std::endl;
    exit( -1 );
} catch(std::exception &e) {
    std::cerr << "Error opening database environment: "
              << envHome
              << " and database "
              << dbName << std::endl;
    std::cerr << e.what() << std::endl;
    exit( -1 );
}
```

When you are done with an environment, you must close it. Before you close an environment, make sure you close any opened databases.

```
try {
    if (myDb != NULL) {
        myDb->close(0);
    }
    myEnv.close(0);

} catch(DbException &e) {
    std::cerr << "Error closing database environment: "
              << envHome
              << " or database "
              << dbName << std::endl;
    std::cerr << e.what() << std::endl;
    exit( -1 );
} catch(std::exception &e) {
    std::cerr << "Error closing database environment: "
              << envHome
              << " or database "
```

```
                << dbName << std::endl;
    std::cerr << e.what() << std::endl;
    exit( -1 );
}
```

Database Example

Throughout this book we will build a couple of applications that load and retrieve inventory data from DB databases. While we are not yet ready to begin reading from or writing to our databases, we can at least create the class that we will use to manage our databases.

Note that subsequent examples in this book will build on this code to perform the more interesting work of writing to and reading from the databases.

Note that you can find the complete implementation of these functions in:

```
DB_INSTALL/examples_cxx/getting_started
```

where *DB_INSTALL* is the location where you placed your DB distribution.

Example 2.1. MyDb Class

To manage our database open and close activities, we encapsulate them in the `MyDb` class. There are several good reasons to do this, the most important being that we can ensure our databases are closed by putting that activity in the `MyDb` class destructor.

To begin, we create our class definition:

```
// File: MyDb.hpp
#include <db_cxx.h>

class MyDb
{
public:
    // Constructor requires a path to the database,
    // and a database name.
    MyDb(std::string &path, std::string &dbName);

    // Our destructor just calls our private close method.
    ~MyDb() { close(); }

    inline Db &getDb() {return db_;}

private:
    Db db_;
    std::string dbFileName_;
    u_int32_t cFlags_;

    // Make sure the default constructor is private
    // We don't want it used.
```

```
    MyDb() : db_(NULL, 0) {}

    // We put our database close activity here.
    // This is called from our destructor. In
    // a more complicated example, we might want
    // to make this method public, but a private
    // method is more appropriate for this example.
    void close();
};
```

Next we need the implementation for the constructor:

```
// File: MyDb.cpp
#include "MyDb.hpp"

// Class constructor. Requires a path to the location
// where the database is located, and a database name
MyDb::MyDb(std::string &path, std::string &dbName)
    : db_(NULL, 0),               // Instantiate Db object
      dbFileName_(path + dbName), // Database file name
      cFlags_(DB_CREATE)          // If the database doesn't yet exist,
                                  // allow it to be created.
{
    try
    {
        // Redirect debugging information to std::cerr
        db_.set_error_stream(&std::cerr);

        // Open the database
        db_.open(NULL, dbFileName_.c_str(), NULL, DB_BTREE, cFlags_, 0);
    }
    // DbException is not a subclass of std::exception, so we
    // need to catch them both.
    catch(DbException &e)
    {
        std::cerr << "Error opening database: " << dbFileName_ << "\n";
        std::cerr << e.what() << std::endl;
    }
    catch(std::exception &e)
    {
        std::cerr << "Error opening database: " << dbFileName_ << "\n";
        std::cerr << e.what() << std::endl;
    }
}
```

And then we need the implementation for the `close()` method:

```
// Private member used to close a database. Called from the class
// destructor.
void
MyDb::close()
{
    // Close the db
    try
    {
        db_.close(0);
        std::cout << "Database " << dbFileName_
                  << " is closed." << std::endl;
    }
    catch(DbException &e)
    {
        std::cerr << "Error closing database: " << dbFileName_ << "\n";
        std::cerr << e.what() << std::endl;
    }
    catch(std::exception &e)
    {
        std::cerr << "Error closing database: " << dbFileName_ << "\n";
        std::cerr << e.what() << std::endl;
    }
}
```

Chapter 3. Database Records

DB records contain two parts — a key and some data. Both the key and its corresponding data are encapsulated in `Dbt` class objects. Therefore, to access a DB record, you need two such objects, one for the key and one for the data.

`Dbt` objects provide a `void *` data member that you use to point to your data, and another member that identifies the data length. They can therefore be used to store anything from simple primitive data to complex class objects so long as the information you want to store resides in a single contiguous block of memory.

This chapter describes `Dbt` usage. It also introduces storing and retrieving key/value pairs from a database.

Using Database Records

Each database record is comprised of two `Dbt` objects — one for the key and another for the data.

```
#include <db_cxx.h>
#include <string.h>

...

float money = 122.45;
char *description = "Grocery bill.";

Dbt key(&money, sizeof(float));
Dbt data(description, strlen(description)+1);
```

Note that in the following example we do not allow DB to assign the memory for the retrieval of the money value. The reason why is that some systems may require float values to have a specific alignment, and the memory as returned by DB may not be properly aligned (the same problem may exist for structures on some systems). We tell DB to use our memory instead of its own by specifying the `DB_DBT_USERMEM` flag. Be aware that when we do this, we must also identify how much user memory is available through the use of the `ulen` field.

```
#include <db_cxx.h>
#include <string.h>

...

Dbt key, data;
float money;
char *description;

key.set_data(&money);
key.set_ulen(sizeof(float));
key.set_flags(DB_DBT_USERMEM);
```

```
// Database retrieval code goes here

// Money is set into the memory that we supplied.
description = (char *)data.get_data();
```

Reading and Writing Database Records

When reading and writing database records, be aware that there are some slight differences in behavior depending on whether your database supports duplicate records. Two or more database records are considered to be duplicates of one another if they share the same key. The collection of records sharing the same key are called a *duplicates set*. In DB, a given key is stored only once for a single duplicates set.

By default, DB databases do not support duplicate records. Where duplicate records are supported, cursors (see below) are typically used to access all of the records in the duplicates set.

DB provides two basic mechanisms for the storage and retrieval of database key/data pairs:

- The `Db::put()` and `Db::get()` methods provide the easiest access for all non-duplicate records in the database. These methods are described in this section.
- Cursors provide several methods for putting and getting database records. Cursors and their database access methods are described in Using Cursors (page 33).

Writing Records to the Database

Records are stored in the database using whatever organization is required by the access method that you have selected. In some cases (such as BTree), records are stored in a sort order that you may want to define (see Setting Comparison Functions (page 73) for more information).

In any case, the mechanics of putting and getting database records do not change once you have selected your access method, configured your sorting routines (if any), and opened your database. From your code's perspective, a simple database put and get is largely the same no matter what access method you are using.

You use `Db::put()` to put, or write, a database record. This method requires you to provide the record's key and data in the form of a pair of `Dbt` objects. You can also provide one or more flags that control DB's behavior for the database write.

Of the flags available to this method, `DB_NOOVERWRITE` may be interesting to you. This flag disallows overwriting (replacing) an existing record in the database. If the provided key already exists in the database, then this method returns `DB_KEYEXIST` even if the database supports duplicates.

For example:

```
#include <db_cxx.h>
#include <string.h>
```

```
...

char *description = "Grocery bill.";
float money = 122.45;

Db my_database(NULL, 0);
// Database open omitted for clarity

Dbt key(&money, sizeof(float));
Dbt data(description, strlen(description) + 1);

int ret = my_database.put(NULL, &key, &data, DB_NOOVERWRITE);
if (ret == DB_KEYEXIST) {
    my_database.err(ret, "Put failed because key %f already exists", money);
}
```

Getting Records from the Database

You can use the `Db::get()` method to retrieve database records. Note that if your database supports duplicate records, then by default this method will only return the first record in a duplicate set. For this reason, if your database supports duplicates, the common solution is to use a cursor to retrieve records from it. Cursors are described in Using Cursors (page 33).

(You can also retrieve a set of duplicate records using a bulk get. To do this, you use the `DB_MULTIPLE` flag on the call to `Db::get()`. For more information, see the DB Programmer's Reference Guide).

By default, `Db::get()` returns the first record found whose key matches the key provide on the call to this method. If your database supports duplicate records, you can change this behavior slightly by supplying the `DB_GET_BOTH` flag. This flag causes `DB::get()` to return the first record that matches the provided key and data.

If the specified key and/or data does not exist in the database, this method returns `DB_NOTFOUND`.

```
#include <db_cxx.h>
#include <string.h>

...

float money;
char description[DESCRIPTION_SIZE + 1];

Db my_database(NULL, 0);
// Database open omitted for clarity

money = 122.45;

Dbt key, data;
```

```
key.set_data(&money);
key.set_size(sizeof(float));

data.set_data(description);
data.set_ulen(DESCRIPTION_SIZE + 1);
data.set_flags(DB_DBT_USERMEM);

my_database.get(NULL, &key, &data, 0);

// Description is set into the memory that we supplied.
```

Note that in this example, the `data.size` field would be automatically set to the size of the retrieved data.

Deleting Records

You can use the `Db::del()` method to delete a record from the database. If your database supports duplicate records, then all records associated with the provided key are deleted. To delete just one record from a list of duplicates, use a cursor. Cursors are described in Using Cursors (page 33).

You can also delete every record in the database by using `Db::truncate()`.

For example:

```
#include <db_cxx.h>

...

Db my_database(NULL, 0);
// Database open omitted for clarity

float money = 122.45;
Dbt key(&money, sizeof(float));

my_database.del(NULL, &key, 0);
```

Data Persistence

When you perform a database modification, your modification is made in the in-memory cache. This means that your data modifications are not necessarily flushed to disk, and so your data may not appear in the database after an application restart.

Note that as a normal part of closing a database, its cache is written to disk. However, in the event of an application or system failure, there is no guarantee that your databases will close cleanly. In this event, it is possible for you to lose data. Under extremely rare circumstances, it is also possible for you to experience database corruption.

Therefore, if you care if your data is durable across system failures, and to guard against the rare possibility of database corruption, you should use transactions to protect your database

modifications. Every time you commit a transaction, DB ensures that the data will not be lost due to application or system failure. Transaction usage is described in the *Berkeley DB Getting Started with Transaction Processing* guide.

If you do not want to use transactions, then the assumption is that your data is of a nature that it need not exist the next time your application starts. You may want this if, for example, you are using DB to cache data relevant only to the current application runtime.

If, however, you are not using transactions for some reason and you still want some guarantee that your database modifications are persistent, then you should periodically call `Db::sync()`. Syncs cause any dirty entries in the in-memory cache and the operating system's file cache to be written to disk. As such, they are quite expensive and you should use them sparingly.

Remember that by default a sync is performed any time a non-transactional database is closed cleanly. (You can override this behavior by specifying `DB_NOSYNC` on the call to `Db::close()`.) That said, you can manually run a sync by calling `Db::sync()`.

If your application or system crashes and you are not using transactions, then you should either discard and recreate your databases, or verify them. You can verify a database using Db::verify(). If your databases do not verify cleanly, use the **db_dump** command to salvage as much of the database as is possible. Use either the `-R` or `-r` command line options to control how aggressive **db_dump** should be when salvaging your databases.

Database Usage Example

In Database Example (page 17) we created a class that opens and closes a database for us. We now make use of that class to load inventory data into two databases that we will use for our inventory system.

Again, remember that you can find the complete implementation for these functions in:

```
DB_INSTALL/examples_cxx/getting_started
```

where *DB_INSTALL* is the location where you placed your DB distribution.

Example 3.1. VENDOR Structure

We want to store data related to an inventory system. There are two types of information that we want to manage: inventory data and related vendor contact information. To manage this information, we could have created a structure for each type of data, but to illustrate storing mixed data without a structure we refrain from creating one for the inventory data.

We now show the definition of the VENDOR structure. Note that the VENDOR structure uses fixed-length fields. This is not necessary and in fact could represent a waste of resources if the number of vendors stored in our database scales to very large numbers. However, for simplicity we use fixed-length fields anyway, especially given that our sample data contains so few vendor records.

```
// File: gettingStartedCommon.hpp
#define MAXFIELD 20
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;
```

Example 3.2. InventoryData Class

In order to manage our actual inventory data, we create a class that encapsulates the data that we want to store for each inventory record. Beyond simple data encapsulation, this class is also capable of marshaling the inventory data into a single contiguous buffer for the purposes of storing in that data in a DB database.

We also provide two constructors for this class. The default constructor simply initializes all our data members for us. A second constructor is also provided that is capable of populating our data members from a `void *`. This second constructor is not really needed until the next chapter where we show how to read data from the databases, but we include it here for the purpose of completeness anyway.

To simplify things a bit, we include the entire implementation for this class in `gettingStartedCommon.hpp` along with our `VENDOR` structure definition.

To begin, we create the public getter and setter methods that we use with our class' private members. We also show the implementation of the method that we use to initialize all our private members.

```
class InventoryData
{
public:
    inline void setPrice(double price) {price_ = price;}
    inline void setQuantity(long quantity) {quantity_ = quantity;}
    inline void setCategory(std::string &category) {category_ = category;}
    inline void setName(std::string &name) {name_ = name;}
    inline void setVendor(std::string &vendor) {vendor_ = vendor;}
    inline void setSKU(std::string &sku) {sku_ = sku;}

    inline double& getPrice() {return(price_);}
    inline long& getQuantity() {return(quantity_);}
    inline std::string& getCategory() {return(category_);}
    inline std::string& getName() {return(name_);}
    inline std::string& getVendor() {return(vendor_);}
    inline std::string& getSKU() {return(sku_);}

    // Initialize our data members
    void clear()
    {
```

```
        price_ = 0.0;
        quantity_ = 0;
        category_.clear();
        name_.clear();
        vendor_.clear();
        sku_.clear();
    }
```

Next we implement our constructors. The default constructor simply calls the `clear()`. The second constructor takes a `void *` as an argument, which it then uses to initialize the data members. Note, again, that we will not actually use this second constructor in this chapter, but we show it here just to be complete anyway.

```
    // Default constructor
    InventoryData() { clear(); }

    // Constructor from a void *
    // For use with the data returned from a bdb get
    InventoryData(void *buffer)
    {
        char *buf = (char *)buffer;

        price_ = *((double *)buf);
        bufLen_ = sizeof(double);

        quantity_ = *((long *)(buf + bufLen_));
        bufLen_ += sizeof(long);

        name_ = buf + bufLen_;
        bufLen_ += name_.size() + 1;

        sku_ = buf + bufLen_;
        bufLen_ += sku_.size() + 1;

        category_ = buf + bufLen_;
        bufLen_ += category_.size() + 1;

        vendor_ = buf + bufLen_;
        bufLen_ += vendor_.size() + 1;
    }
```

Next we provide a couple of methods for returning the class' buffer and the size of the buffer. These are used for actually storing the class' data in a DB database.

```
    // Marshalls this classes data members into a single
    // contiguous memory location for the purpose of storing
    // the data in a database.
    char *
    getBuffer()
    {
        // Zero out the buffer
        memset(databuf_, 0, 500);
        // Now pack the data into a single contiguous memory location for
        // storage.
        bufLen_ = 0;
        int dataLen = 0;

        dataLen = sizeof(double);
        memcpy(databuf_, &price_, dataLen);
        bufLen_ += dataLen;

        dataLen = sizeof(long);
        memcpy(databuf_ + bufLen_, &quantity_, dataLen);
        bufLen_ += dataLen;

        packString(databuf_, name_);
        packString(databuf_, sku_);
        packString(databuf_, category_);
        packString(databuf_, vendor_);

        return (databuf_);
    }

    // Returns the size of the buffer. Used for storing
    // the buffer in a database.
    inline int getBufferSize() { return (bufLen_); }
```

Our last public method is a utility method that we use to get the class to show itself.

```
     // Utility function used to show the contents of this class
    void
    show() {
        std::cout << "\nName:         " << name_ << std::endl;
        std::cout << "    SKU:        " << sku_ << std::endl;
        std::cout << "    Price:      " << price_ << std::endl;
        std::cout << "    Quantity:   " << quantity_ << std::endl;
        std::cout << "    Category:   " << category_ << std::endl;
        std::cout << "    Vendor:     " << vendor_ << std::endl;
    }
```

Finally, we provide a private method that is used to help us pack data into our buffer, and we declare our private data members.

```
private:

    // Utility function that appends a char * to the end of
    // the buffer.
    void
    packString(char *buffer, std::string &theString)
    {
        int string_size = theString.size() + 1;
        memcpy(buffer+bufLen_, theString.c_str(), string_size);
        bufLen_ += string_size;
    }

    // Data members
    std::string category_, name_, vendor_, sku_;
    double price_;
    long quantity_;
    int bufLen_;
    char databuf_[500];
};
```

Example 3.3. example_database_load

Our initial sample application loads database information from several flat files. To save space, we won't show all the details of this example program. However, as always you can find the complete implementation for this program here:

```
DB_INSTALL/examples_cxx/getting_started
```

where *DB_INSTALL* is the location where you placed your DB distribution.

We begin with the normal include directives and forward declarations:

```
// File: example_database_load.cpp
#include <iostream>
#include <fstream>
#include <cstdlib>

#include "MyDb.hpp"
#include "gettingStartedCommon.hpp"

// Forward declarations
void loadVendorDB(MyDb&, std::string&);
void loadInventoryDB(MyDb&, std::string&);
```

Next we begin our `main()` function with the variable declarations and command line parsing that is normal for most command line applications:

```
// Loads the contents of vendors.txt and inventory.txt into
// Berkeley DB databases.
int
main(int argc, char *argv[])
{
    // Initialize the path to the database files
    std::string basename("./");
    std::string databaseHome("./");

    // Database names
    std::string vDbName("vendordb.db");
    std::string iDbName("inventorydb.db");

    // Parse the command line arguments here and determine
    // the location of the flat text files containing the
    // inventory data here. This step is omitted for clarity.

    //  Identify the full name for our input files, which should
    //  also include some path information.
    std::string inventoryFile = basename + "inventory.txt";
    std::string vendorFile = basename + "vendors.txt";

    try
    {
        // Open all databases.
        MyDb inventoryDB(databaseHome, iDbName);
        MyDb vendorDB(databaseHome, vDbName);

        // Load the vendor database
        loadVendorDB(vendorDB, vendorFile);

        // Load the inventory database
        loadInventoryDB(inventoryDB, inventoryFile);
    } catch(DbException &e) {
        std::cerr << "Error loading databases. " << std::endl;
        std::cerr << e.what() << std::endl;
        return(e.get_errno());
    } catch(std::exception &e) {
        std::cerr << "Error loading databases. " << std::endl;
        std::cerr << e.what() << std::endl;
        return(-1);
    }

    return(0);
} // End main
```

Note that we do not explicitly close our databases here. This is because the databases are encapsulated in `MyDb` class objects, and those objects are on the stack. When they go out of scope, their destructors will cause the database close to occur.

Notice that there is not a lot to this function because we have pushed off all the database activity to other places.

Next we show the implementation of `loadVendorDB()`. We load this data by scanning (line by line) the contents of the `vendors.txt` file into a VENDOR structure. Once we have a line scanned into the structure, we can store that structure into our vendors database.

Note that we use the vendor's name as the key here. In doing so, we assume that the vendor's name is unique in our database. If it was not, we would either have to select a different key, or architect our application such that it could cope with multiple vendor records with the same name.

```
// Loads the contents of the vendors.txt file into a database
void
loadVendorDB(MyDb &vendorDB, std::string &vendorFile)
{
    std::ifstream inFile(vendorFile.c_str(), std::ios::in);
    if ( !inFile )
    {
        std::cerr << "Could not open file '" << vendorFile
                  << "'. Giving up." << std::endl;
        throw std::exception();
    }

    VENDOR my_vendor;
    while (!inFile.eof())
    {
        std::string stringBuf;
        std::getline(inFile, stringBuf);
        memset(&my_vendor, 0, sizeof(VENDOR));

        // Scan the line into the structure.
        // Convenient, but not particularly safe.
        // In a real program, there would be a lot more
        // defensive code here.
        sscanf(stringBuf.c_str(),
          "%20[^#]#%20[^#]#%20[^#]#%3[^#]#%6[^#]#%13[^#]#%20[^#]#%20[^\n]",
          my_vendor.name, my_vendor.street,
          my_vendor.city, my_vendor.state,
          my_vendor.zipcode, my_vendor.phone_number,
          my_vendor.sales_rep, my_vendor.sales_rep_phone);

        Dbt key(my_vendor.name, strlen(my_vendor.name) + 1);
        Dbt data(&my_vendor, sizeof(VENDOR));

        vendorDB.getDb().put(NULL, &key, &data, 0);
    }
    inFile.close();
}
```

Finally, we need to write the `loadInventoryDB()` function. To load the inventory information, we read in each line of the inventory.txt file, obtain each field from it, then we load this data into an `InventoryData` instance.

To help us obtain the various fields from each line of input, we also create a simple helper function that locates the position of the first a field delimiter (a pound (#) sign) from a line of input.

Note that we could have simply decided to store our inventory data in a structure very much like the VENDOR structure that we use above. However, by storing this data in the `InventoryData` class, which identifies the size of the data that it contains, we can use the smallest amount of space possible for the data that we are storing. The result is that our cache can be smaller than it might otherwise be and our database will take less space on disk than if we used a structure with fixed-length fields.

For a trivial dataset such as what we use for these examples, these resource savings are negligible. But if we were storing hundreds of millions of records, then the cost savings may become significant.

```
// Used to locate the first pound sign (a field delimiter)
// in the input string.
int
getNextPound(std::string &theString, std::string &substring)
{
    int pos = theString.find("#");
    substring.assign(theString, 0, pos);
    theString.assign(theString, pos + 1, theString.size());
    return (pos);
}


// Loads the contents of the inventory.txt file into a database
void
loadInventoryDB(MyDb &inventoryDB, std::string &inventoryFile)
{
    InventoryData inventoryData;
    std::string substring;
    int nextPound;

    std::ifstream inFile(inventoryFile.c_str(), std::ios::in);
    if (!inFile)
    {
        std::cerr << "Could not open file '" << inventoryFile
                  << "'. Giving up." << std::endl;
        throw std::exception();
    }

    while (!inFile.eof())
    {
        inventoryData.clear();
```

```
        std::string stringBuf;
        std::getline(inFile, stringBuf);

        // Now parse the line
        if (!stringBuf.empty())
        {
            nextPound = getNextPound(stringBuf, substring);
            inventoryData.setName(substring);

            nextPound = getNextPound(stringBuf, substring);
            inventoryData.setSKU(substring);

            nextPound = getNextPound(stringBuf, substring);
            inventoryData.setPrice(strtod(substring.c_str(), 0));

            nextPound = getNextPound(stringBuf, substring);
            inventoryData.setQuantity(strtol(substring.c_str(), 0, 10));

            nextPound = getNextPound(stringBuf, substring);
            inventoryData.setCategory(substring);

            nextPound = getNextPound(stringBuf, substring);
            inventoryData.setVendor(substring);

            void *buff = (void *)inventoryData.getSKU().c_str();
            int size = inventoryData.getSKU().size()+1;
            Dbt key(buff, size);

            buff = inventoryData.getBuffer();
            size = inventoryData.getBufferSize();
            Dbt data(buff, size);

            inventoryDB.getDb().put(NULL, &key, &data, 0);
        }
    }
    inFile.close();
}
```

In the next chapter we provide an example that shows how to read the inventory and vendor databases.

Chapter 4. Using Cursors

Cursors provide a mechanism by which you can iterate over the records in a database. Using cursors, you can get, put, and delete database records. If a database allows duplicate records, then cursors are the easiest way that you can access anything other than the first record for a given key.

This chapter introduces cursors. It explains how to open and close them, how to use them to modify databases, and how to use them with duplicate records.

Opening and Closing Cursors

Cursors are managed using the `Dbc` class. To use a cursor, you must open it using the `Db::cursor()` method.

For example:

```
#include <db_cxx.h>

...

Dbc *cursorp;
Db my_database(NULL, 0);

// Database open omitted for clarity

// Get a cursor
my_database.cursor(NULL, &cursorp, 0);
```

When you are done with the cursor, you should close it. To close a cursor, call the `Dbc::close()` method. Note that closing your database while cursors are still opened within the scope of the DB handle, especially if those cursors are writing to the database, can have unpredictable results. Always close your cursors before closing your database.

```
#include <db_cxx.h>

...

Dbc *cursorp;
Db my_database(NULL, 0);

// Database and cursor open omitted for clarity

if (cursorp != NULL)
    cursorp->close();

my_database.close(0);
```

Getting Records Using the Cursor

To iterate over database records, from the first record to the last, simply open the cursor and then use the `Dbc::get()` method. Note that you need to supply the `DB_NEXT` flag to this method. For example:

```
#include <db_cxx.h>

...

Db my_database(NULL, 0);
Dbc *cursorp;

try {
    // Database open omitted for clarity

    // Get a cursor
    my_database.cursor(NULL, &cursorp, 0);

    Dbt key, data;
    int ret;

    // Iterate over the database, retrieving each record in turn.
    while ((ret = cursorp->get(&key, &data, DB_NEXT)) == 0) {
        // Do interesting things with the Dbts here.
    }
    if (ret != DB_NOTFOUND) {
        // ret should be DB_NOTFOUND upon exiting the loop.
        // Dbc::get() will by default throw an exception if any
        // significant errors occur, so by default this if block
        // can never be reached.
    }
} catch(DbException &e) {
        my_database.err(e.get_errno(), "Error!");
} catch(std::exception &e) {
        my_database.errx("Error! %s", e.what());
}

// Cursors must be closed
if (cursorp != NULL)
    cursorp->close();

my_database.close(0);
```

To iterate over the database from the last record to the first, use DB_PREV instead of DB_NEXT:

```
#include <db_cxx.h>

...

Db my_database(NULL, 0);
Dbc *cursorp;

try {
    // Database open omitted for clarity

    // Get a cursor
    my_database.cursor(NULL, &cursorp, 0);

    Dbt key, data;
    int ret;
    // Iterate over the database, retrieving each record in turn.
    while ((ret = cursorp->get(&key, &data, DB_PREV)) == 0) {
        // Do interesting things with the Dbts here.
    }
    if (ret != DB_NOTFOUND) {
        // ret should be DB_NOTFOUND upon exiting the loop.
        // Dbc::get() will by default throw an exception if any
        // significant errors occur, so by default this if block
        // can never be reached.
    }
} catch(DbException &e) {
        my_database.err(e.get_errno(), "Error!");
} catch(std::exception &e) {
        my_database.errx("Error! %s", e.what());
}

// Cursors must be closed
if (cursorp != NULL)
    cursorp->close();

my_database.close(0);
```

Searching for Records

You can use cursors to search for database records. You can search based on just a key, or you can search based on both the key and the data. You can also perform partial matches if your database supports sorted duplicate sets. In all cases, the key and data parameters of these methods are filled with the key and data values of the database record to which the cursor is positioned as a result of the search.

Also, if the search fails, then cursor's state is left unchanged and DB_NOTFOUND is returned.

To use a cursor to search for a record, use Dbt::get(). When you use this method, you can provide the following flags:

Notice in the following list that the cursor flags use the keyword `SET` when the cursor examines just the key portion of the records (in this case, the cursor is set to the record whose key matches the value provided to the cursor). Moreover, when the cursor uses the keyword `GET`, then the cursor is positioned to both the key *and* the data values provided to the cursor.

Regardless of the keyword you use to get a record with a cursor, the cursor's key and data `Dbt`s are filled with the data retrieved from the record to which the cursor is positioned.

- `DB_SET`

  Moves the cursor to the first record in the database with the specified key.

- `DB_SET_RANGE`

  Identical to `DB_SET` unless you are using the BTree access. In this case, the cursor moves to the first record in the database whose key is greater than or equal to the specified key. This comparison is determined by the comparison function that you provide for the database. If no comparison function is provided, then the default lexicographical sorting is used.

  For example, suppose you have database records that use the following Strings as keys:

  ```
  Alabama
  Alaska
  Arizona
  ```

  Then providing a search key of `Alaska` moves the cursor to the second key noted above. Providing a key of `Al` moves the cursor to the first key (`Alabama`), providing a search key of `Alas` moves the cursor to the second key (`Alaska`), and providing a key of `Ar` moves the cursor to the last key (`Arizona`).

- `DB_GET_BOTH`

  Moves the cursor to the first record in the database that uses the specified key and data.

- `DB_GET_BOTH_RANGE`

  Moves the cursor to the first record in the database whose key matches the specified key and whose data is greater than or equal to the specified data. If the database supports duplicate records, then on matching the key, the cursor is moved to the duplicate record with the smallest data that is greater than or equal to the specified data.

  For example, suppose your database uses BTree and it has database records that use the following key/data pairs:

  ```
  Alabama/Athens
  Alabama/Florence
  Alaska/Anchorage
  Alaska/Fairbanks
  ```

```
Arizona/Avondale
Arizona/Florence
```

then providing:

a search key of ...	and a search data of ...	moves the cursor to ...
Alaska	Fa	Alaska/Fairbanks
Arizona	Fl	Arizona/Florence
Alaska	An	Alaska/Anchorage

For example, assuming a database containing sorted duplicate records of U.S. States/U.S Cities key/data pairs (both as Strings), then the following code fragment can be used to position the cursor to any record in the database and print its key/data values:

```
#include <db_cxx.h>
#include <string.h>

...

Db my_database(NULL, 0);
Dbc *cursorp;

try {
    // database open omitted for clarity

    // Get a cursor
    my_database.cursor(NULL, &cursorp, 0);

    // Search criteria
    char *search_key = "Alaska";
    char *search_data = "Fa";

    // Set up our DBTs
    Dbt key(search_key, strlen(search_key) + 1);
    Dbt data(search_data, strlen(search_data) + 1);

    // Position the cursor to the first record in the database whose
    // key matches the search key and whose data begins with the search
    // data.
    int ret = cursorp->get(&key, &data, DB_GET_BOTH_RANGE);
    if (!ret) {
        // Do something with the data
    }
} catch(DbException &e) {
        my_database.err(e.get_errno(), "Error!");
} catch(std::exception &e) {
        my_database.errx("Error! %s", e.what());
```

```
}

// Close the cursor
if (cursorp != NULL)
    cursorp->close();

// Close the database
my_database.close(0);
```

Working with Duplicate Records

A record is a duplicate of another record if the two records share the same key. For duplicate records, only the data portion of the record is unique.

Duplicate records are supported only for the BTree or Hash access methods. For information on configuring your database to use duplicate records, see Allowing Duplicate Records (page 71).

If your database supports duplicate records, then it can potentially contain multiple records that share the same key. By default, normal database get operations will only return the first such record in a set of duplicate records. Typically, subsequent duplicate records are accessed using a cursor. The following `Dbc::get()` flags are interesting when working with databases that support duplicate records:

- `DB_NEXT`, `DB_PREV`

  Shows the next/previous record in the database, regardless of whether it is a duplicate of the current record. For an example of using these methods, see Getting Records Using the Cursor (page 34).

- `DB_GET_BOTH_RANGE`

  Useful for seeking the cursor to a specific record, regardless of whether it is a duplicate record. See Searching for Records (page 35) for more information.

- `DB_NEXT_NODUP`, `DB_PREV_NODUP`

  Gets the next/previous non-duplicate record in the database. This allows you to skip over all the duplicates in a set of duplicate records. If you call `Dbc::get()` with `DB_PREV_NODUP`, then the cursor is positioned to the last record for the previous key in the database. For example, if you have the following records in your database:

  ```
  Alabama/Athens
  Alabama/Florence
  Alaska/Anchorage
  Alaska/Fairbanks
  Arizona/Avondale
  Arizona/Florence
  ```

  and your cursor is positioned to `Alaska/Fairbanks`, and you then call `Dbc::get()` with `DB_PREV_NODUP`, then the cursor is positioned to Alabama/Florence. Similarly, if you call

Dbc::get() with DB_NEXT_NODUP, then the cursor is positioned to the first record corresponding to the next key in the database.

If there is no next/previous key in the database, then DB_NOTFOUND is returned, and the cursor is left unchanged.

- DB_NEXT_DUP

  Gets the next record that shares the current key. If the cursor is positioned at the last record in the duplicate set and you call Dbc::get() with DB_NEXT_DUP, then DB_NOTFOUND is returned and the cursor is left unchanged.

For example, the following code fragment positions a cursor to a key and displays it and all its duplicates.

```
#include <db_cxx.h>
#include <string.h>

...

char *search_key = "Al";

Db my_database(NULL, 0);
Dbc *cursorp;

try {
    // database open omitted for clarity

    // Get a cursor
    my_database.cursor(NULL, &cursorp, 0);

    // Set up our DBTs
    Dbt key(search_key, strlen(search_key) + 1);
    Dbt data;

    // Position the cursor to the first record in the database whose
    // key and data begin with the correct strings.
    int ret = cursorp->get(&key, &data, DB_SET);
    while (ret != DB_NOTFOUND) {
        std::cout << "key: " << (char *)key.get_data()
                  << "data: " << (char *)data.get_data()<< std::endl;
        ret = cursorp->get(&key, &data, DB_NEXT_DUP);
    }
} catch(DbException &e) {
        my_database.err(e.get_errno(), "Error!");
} catch(std::exception &e) {
        my_database.errx("Error! %s", e.what());
}

// Close the cursor
```

```
if (cursorp != NULL)
    cursorp->close();

// Close the database
my_database.close(0);
```

Putting Records Using Cursors

You can use cursors to put records into the database. DB's behavior when putting records into the database differs depending on the flags that you use when writing the record, on the access method that you are using, and on whether your database supports sorted duplicates.

Note that when putting records to the database using a cursor, the cursor is positioned at the record you inserted.

You use `Dbc::put()` to put (write) records to the database. You can use the following flags with this method:

- `DB_NODUPDATA`

  If the provided key already exists in the database, then this method returns `DB_KEYEXIST`.

  If the key does not exist, then the order that the record is put into the database is determined by the insertion order in use by the database. If a comparison function has been provided to the database, the record is inserted in its sorted location. Otherwise (assuming BTree), lexicographical sorting is used, with shorter items collating before longer items.

  This flag can only be used for the BTree and Hash access methods, and only if the database has been configured to support sorted duplicate data items (`DB_DUPSORT` was specified at database creation time).

  This flag cannot be used with the Queue or Recno access methods.

  For more information on duplicate records, see Allowing Duplicate Records (page 71).

- `DB_KEYFIRST`

  For databases that do not support duplicates, this method behaves exactly the same as if a default insertion was performed. If the database supports duplicate records, and a duplicate sort function has been specified, the inserted data item is added in its sorted location. If the key already exists in the database and no duplicate sort function has been specified, the inserted data item is added as the first of the data items for that key.

- `DB_KEYLAST`

  Behaves exactly as if `DB_KEYFIRST` was used, except that if the key already exists in the database and no duplicate sort function has been specified, the inserted data item is added as the last of the data items for that key.

For example:

```
#include <db_cxx.h>
#include <string.h>

...

char *key1str = "My first string";
char *data1str = "My first data";
char *key2str = "A second string";
char *data2str = "My second data";
char *data3str = "My third data";


Db my_database(NULL, 0);
Dbc *cursorp;

try {
    // Set up our DBTs
    Dbt key1(key1str, strlen(key1str) + 1);
    Dbt data1(data1str, strlen(data1str) + 1);

    Dbt key2(key2str, strlen(key2str) + 1);
    Dbt data2(data2str, strlen(data2str) + 1);
    Dbt data3(data3str, strlen(data3str) + 1);

    // Database open omitted

    // Get the cursor
    my_database.cursor(NULL, &cursorp, 0);

    // Assuming an empty database, this first put places
    // "My first string"/"My first data" in the first
    // position in the database
    int ret = cursorp->put(&key1, &data1, DB_KEYFIRST);

    // This put places "A second string"/"My second data" in the
    // the database according to its key sorts against the key
    // used for the currently existing database record. Most likely
    // this record would appear first in the database.
    ret = cursorp->put(&key2, &data2,
            DB_KEYFIRST); /* Added according to sort order */

    // If duplicates are not allowed, the currently existing record that
    // uses "key2" is overwritten with the data provided on this put.
    // That is, the record "A second string"/"My second data" becomes
    // "A second string"/"My third data"
    //
    // If duplicates are allowed, then "My third data" is placed in the
    // duplicates list according to how it sorts against "My second data".
    ret = cursorp->put(&key2, &data3,
```

```
            DB_KEYFIRST); // If duplicates are not allowed, record
                          // is overwritten with new data. Otherwise,
                          // the record is added to the beginning of
                          // the duplicates list.
} catch(DbException &e) {
        my_database.err(e.get_errno(), "Error!");
} catch(std::exception &e) {
        my_database.errx("Error! %s", e.what());
}

// Cursors must be closed
if (cursorp != NULL)
    cursorp->close();

my_database.close(0);
```

Deleting Records Using Cursors

To delete a record using a cursor, simply position the cursor to the record that you want to delete and then call `Dbc::del()`.

For example:

```
#include <db_cxx.h>
#include <string.h>

...

char *key1str = "My first string";
Db my_database(NULL, 0);
Dbc *cursorp;

try {
    // Database open omitted

    // Get the cursor
    my_database.cursor(NULL, &cursorp, 0);

    // Set up our DBTs
    Dbt key(key1str, strlen(key1str) + 1);
    Dbt data;

    // Iterate over the database, deleting each record in turn.
    int ret;
    while ((ret = cursorp->get(&key, &data,
                                DB_SET)) == 0) {
        cursorp->del(0);
    }
```

```
} catch(DbException &e) {
    my_database.err(e.get_errno(), "Error!");
} catch(std::exception &e) {
    my_database.errx("Error! %s", e.what());
}

// Cursors must be closed
if (cursorp != NULL)
    cursorp->close();

my_database.close(0);
```

Replacing Records Using Cursors

You replace the data for a database record by using `Dbc::put()` with the `DB_CURRENT` flag.

```
#include <db_cxx.h>
#include <string.h>

...

Db my_database(NULL, 0);
Dbc *cursorp;

int ret;
char *key1str = "My first string";
char *replacement_data = "replace me";

try {
    // Database open omitted

    // Get the cursor
    my_database.cursor(NULL, &cursorp, 0);

    // Set up our DBTs
    Dbt key(key1str, strlen(key1str) + 1);
    Dbt data;

    // Position the cursor */
    ret = cursorp->get(&key, &data, DB_SET);
    if (ret == 0) {
        data.set_data(replacement_data);
        data.set_size(strlen(replacement_data) + 1);
        cursorp->put(&key, &data, DB_CURRENT);
    }
} catch(DbException &e) {
        my_database.err(e.get_errno(), "Error!");
} catch(std::exception &e) {
        my_database.errx("Error! %s", e.what());
```

```
}

// Cursors must be closed
if (cursorp != NULL)
    cursorp->close();

my_database.close(0);
```

Note that you cannot change a record's key using this method; the key parameter is always ignored when you replace a record.

When replacing the data portion of a record, if you are replacing a record that is a member of a sorted duplicates set, then the replacement will be successful only if the new record sorts identically to the old record. This means that if you are replacing a record that is a member of a sorted duplicates set, and if you are using the default lexicographic sort, then the replacement will fail due to violating the sort order. However, if you provide a custom sort routine that, for example, sorts based on just a few bytes out of the data item, then potentially you can perform a direct replacement and still not violate the restrictions described here.

Under these circumstances, if you want to replace the data contained by a duplicate record, and you are not using a custom sort routine, then delete the record and create a new record with the desired key and data.

Cursor Example

In Database Usage Example (page 24) we wrote an application that loaded two databases with vendor and inventory information. In this example, we will write an application to display all of the items in the inventory database. As a part of showing any given inventory item, we will look up the vendor who can provide the item and show the vendor's contact information.

Specifically, the `example_database_read` application does the following:

1. Opens the the inventory and vendor databases that were created by our `example_database_load` application. See example_database_load (page 28) for information on how that application creates the databases and writes data to them.
2. Obtains a cursor from the inventory database.
3. Steps through the inventory database, displaying each record as it goes.
4. Gets the name of the vendor for that inventory item from the inventory record.
5. Uses the vendor name to look up the vendor record in the vendor database.
6. Displays the vendor record.

Remember that you can find the complete implementation of this application in:

```
DB_INSTALL/examples_cxx/getting_started
```

where *DB_INSTALL* is the location where you placed your DB distribution.

Example 4.1. example_database_read

To begin, we include the necessary header files and perform our forward declarations. We also write our `usage()` function.

```
// File: example_database_read.cpp
#include <iostream>
#include <fstream>
#include <cstdlib>

#include "MyDb.hpp"
#include "gettingStartedCommon.hpp"

// Forward declarations
int show_all_records(MyDb &inventoryDB, MyDb &vendorDB);
int show_vendor(MyDb &vendorDB, const char *vendor);
```

Next we write our `main()` function. Note that it is somewhat unnecessarily complicated here because we will be extending it in the next chapter to perform inventory item lookups.

```
// Displays all inventory items and the associated vendor record.
int
main (int argc, char *argv[])
{
    // Initialize the path to the database files
    std::string databaseHome("./");

    // Database names
    std::string vDbName("vendordb.db");
    std::string iDbName("inventorydb.db");

    // Parse the command line arguments
    // Omitted for brevity

    try
    {
        // Open all databases.
        MyDb inventoryDB(databaseHome, iDbName);
        MyDb vendorDB(databaseHome, vDbName);

        show_all_records(inventoryDB, vendorDB);
    } catch(DbException &e) {
        std::cerr << "Error reading databases. " << std::endl;
        std::cerr << e.what() << std::endl;
        return(e.get_errno());
    } catch(std::exception &e) {
        std::cerr << "Error reading databases. " << std::endl;
        std::cerr << e.what() << std::endl;
        return(-1);
    }
```

```
    return(0);
} // End main
```

Next we need to write the `show_all_records()` function. This function displays all of the inventory records found in the inventory database. Once it shows the inventory record, it retrieves the vendor's name from that record and uses it to look up and display the appropriate vendor record:

```
// Shows all the records in the inventory database.
// For each inventory record shown, the appropriate
// vendor record is also displayed.
int
show_all_records(MyDb &inventoryDB, MyDb &vendorDB)
{
    // Get a cursor to the inventory db
    Dbc *cursorp;
    try {
        inventoryDB.getDb().cursor(NULL, &cursorp, 0);

        // Iterate over the inventory database, from the first record
        // to the last, displaying each in turn
        Dbt key, data;
        int ret;
        while ((ret = cursorp->get(&key, &data, DB_NEXT)) == 0 )
        {
            InventoryData inventoryItem(data.get_data());
            inventoryItem.show();

            show_vendor(vendorDB, inventoryItem.getVendor().c_str());
        }
    } catch(DbException &e) {
        inventoryDB.getDb().err(e.get_errno(), "Error in show_all_records");
        cursorp->close();
        throw e;
    } catch(std::exception &e) {
        cursorp->close();
        throw e;
    }

    cursorp->close();
    return (0);
}
```

Note that the `InventoryData` class that we use here is described in InventoryData Class (page 25).

Having displayed the inventory record, we now want to display the vendor record corresponding to this record. In this case we do not need to use a cursor to display the vendor record. Using a cursor here complicates our code slightly for no good gain. Instead, we simply perform a `get()` directly against the vendor database.

```
// Shows a vendor record. Each vendor record is an instance of
// a vendor structure. See loadVendorDB() in
// example_database_load for how this structure was originally
// put into the database.
int
show_vendor(MyDb &vendorDB, const char *vendor)
{
    Dbt data;
    VENDOR my_vendor;

    try {
        // Set the search key to the vendor's name
        // vendor is explicitly cast to char * to stop a compiler
        // complaint.
        Dbt key((char *)vendor, strlen(vendor) + 1);

        // Make sure we use the memory we set aside for the VENDOR
        // structure rather than the memory that DB allocates.
        // Some systems may require structures to be aligned in memory
        // in a specific way, and DB may not get it right.

        data.set_data(&my_vendor);
        data.set_ulen(sizeof(VENDOR));
        data.set_flags(DB_DBT_USERMEM);

        // Get the record
        vendorDB.getDb().get(NULL, &key, &data, 0);
        std::cout << "        " << my_vendor.street << "\n"
                  << "        " << my_vendor.city << ", "
                  << my_vendor.state << "\n"
                  << "        " << my_vendor.zipcode << "\n"
                  << "        " << my_vendor.phone_number << "\n"
                  << "        Contact: " << my_vendor.sales_rep << "\n"
                  << "                 " << my_vendor.sales_rep_phone
                  << std::endl;

    } catch(DbException &e) {
        vendorDB.getDb().err(e.get_errno(), "Error in show_vendor");
        throw e;
    } catch(std::exception &e) {
        throw e;
    }
    return (0);
}
```

That completes the implementation of `example_database_read()`. In the next chapter, we will extend this application to make use of a secondary database so that we can query the inventory database for a specific inventory item.

Chapter 5. Secondary Databases

Usually you find database records by means of the record's key. However, the key that you use for your record will not always contain the information required to provide you with rapid access to the data that you want to retrieve. For example, suppose your database contains records related to users. The key might be a string that is some unique identifier for the person, such as a user ID. Each record's data, however, would likely contain a complex object containing details about people such as names, addresses, phone numbers, and so forth. While your application may frequently want to query a person by user ID (that is, by the information stored in the key), it may also on occasion want to locate people by, say, their name.

Rather than iterate through all of the records in your database, examining each in turn for a given person's name, you create indexes based on names and then just search that index for the name that you want. You can do this using secondary databases. In DB, the database that contains your data is called a *primary database*. A database that provides an alternative set of keys to access that data is called a *secondary database*. In a secondary database, the keys are your alternative (or secondary) index, and the data corresponds to a primary record's key.

You create a secondary database by creating the database, opening it, and then *associating* the database with the *primary* database (that is, the database for which you are creating the index). As a part of associating the secondary database to the primary, you must provide a callback that is used to create the secondary database keys. Typically this callback creates a key based on data found in the primary database record's key or data.

Once opened, DB manages secondary databases for you. Adding or deleting records in your primary database causes DB to update the secondary as necessary. Further, changing a record's data in the primary database may cause DB to modify a record in the secondary, depending on whether the change forces a modification of a key in the secondary database.

Note that you can not write directly to a secondary database. Any attempt to write to a secondary database results in a non-zero status return. To change the data referenced by a secondary record, modify the primary database instead. The exception to this rule is that delete operations are allowed on the secondary database. See Deleting Secondary Database Records (page 53) for more information.

Secondary database records are updated/created by DB only if the key creator callback function returns `0`. If a value other than `0` is returned, then DB will not add the key to the secondary database, and in the event of a record update it will remove any existing key. Note that the callback can use either `DB_DONOTINDEX` or some error code outside of DB's name space to indicate that the entry should not be indexed.

See Implementing Key Extractors (page 50) for more information.

When you read a record from a secondary database, DB automatically returns the data and optionally the key from the corresponding record in the primary database.

Opening and Closing Secondary Databases

You manage secondary database opens and closes in the same way as you would any normal database. The only difference is that:

- You must associate the secondary to a primary database using `Db::associate()`.
- When closing your databases, it is a good idea to make sure you close your secondaries before closing your primaries. This is particularly true if your database closes are not single threaded.

When you associate a secondary to a primary database, you must provide a callback that is used to generate the secondary's keys. These callbacks are described in the next section.

For example, to open a secondary database and associate it to a primary database:

```
#include <db_cxx.h>

...

Db my_database(NULL, 0); // Primary
Db my_index(NULL, 0);    // Secondary

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

// Setup the secondary to use sorted duplicates.
// This is often desirable for secondary databases.
my_index.set_flags(DB_DUPSORT);

// 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_CREATE,         // 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 next
                                     // section.
                      0);            // Flags
```

Closing the primary and secondary databases is accomplished exactly as you would for any database:

```
// Close the secondary before the primary
my_index.close(0);
my_database.close(0);
```

Implementing Key Extractors

You must provide every secondary database with a class that creates keys from primary records. You identify this class when you associate your secondary database to your primary.

You can create keys using whatever data you want. Typically you will base your key on some information found in a record's data, but you can also use information found in the primary record's key. How you build your keys is entirely dependent upon the nature of the index that you want to maintain.

You implement a key extractor by writing a function that extracts the necessary information from a primary record's key or data. This function must conform to a specific prototype, and it must be provided as a callback to the `associate()` method.

For example, suppose your primary database records contain data that uses the following structure:

```
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;
```

Further suppose that you want to be able to query your primary database based on the name of a sales representative. Then you would write a function that looks like this:

```
#include <db_cxx.h>

...

int
get_sales_rep(Db *sdbp,          // secondary db handle
              const Dbt *pkey,   // primary db record's key
              const Dbt *pdata,  // primary db record's data
              Dbt *skey)         // secondary db record's key
{
    VENDOR *vendor;

    // First, extract the structure contained in the primary's data
    vendor = (VENDOR *)pdata->get_data();
```

```
    // Now set the secondary key's data to be the representative's name
    skey->set_data(vendor->sales_rep);
    skey->set_size(strlen(vendor->sales_rep) + 1);

    // Return 0 to indicate that the record can be created/updated.
    return (0);
}
```

In order to use this function, you provide it on the `associate()` method after the primary and secondary databases have been created and opened:

```
db.associate(NULL,           // TXN id
             &sdb,           // Secondary database
             get_sales_rep,      // Callback used for key creation.
             0);                 // Flags
```

Working with Multiple Keys

Until now we have only discussed indexes as if there is a one-to-one relationship between the secondary key and the primary database record. In fact, it is possible to generate multiple keys for any given record, provided that you take appropriate steps in your key creator to do so.

For example, suppose you had a database that contained information about books. Suppose further that you sometimes want to look up books by author. Because sometimes books have multiple authors, you may want to return multiple secondary keys for every book that you index.

To do this, you write a key extractor that returns a Dbt whose `data` member points to an array of Dbts. Each such member of this array contains a single secondary key. In addition, the Dbt returned by your key extractor must have a size field equal to the number of elements contained in the Dbt array. Also, the flag field for the Dbt returned by the callback must include `DB_DBT_MULTIPLE`. For example:

It is important that the array of secondary keys created by your callback not contain repeats. That is, every element in the array must be unique. If the array does not contain a unique set, then the secondary can get out of sync with the primary.

```
int
my_callback(Db *dbp, const Dbt *pkey, const Dbt *pdata, Dbt *skey)
{
    Dbt *tmpdbt;
    char *tmpdata1, tmpdata2;

    // This example skips the step of extracting the data you
    // want to use for building your secondary keys from the
    // pkey or pdata Dbt.

    // Assume for the purpose of this example that the data
    // is temporarily stored in two variables,
    // tmpdata1 and tmpdata2.
```

```
    // Create an array of Dbts that is large enough for the
    // number of keys that you want to return. In this case,
    // we go with an array of size two.

    tmpdbt = malloc(sizeof(Dbt) * 2);
    memset(tmpdbt, 0, sizeof(Dbt) * 2);

    // Now assign secondary keys to each element of the array.
    tmpdbt[0].set_data(tmpdata1);
    tmpdbt[0].set_size((u_int32_t)strlen(tmpdbt[0].data) + 1);
    tmpdbt[1].set_data(tmpdata2);
    tmpdbt[1].set_size((u_int32_t)strlen(tmpdbt[1].data) + 1);

    // Now we set flags for the returned Dbt. DB_DBT_MULTIPLE is
    // required in order for DB to know that the Dbt references an
    // array. In addition, we set DB_DBT_APPMALLOC because we
    // dynamically allocated memory for the Dbt's data field.
    // DB_DBT_APPMALLOC causes DB to release that memory once it
    // is done with the returned Dbt.
    skey->set_flags(DB_DBT_MULTIPLE | DB_DBT_APPMALLOC);

    // Point the results data field to the arrays of Dbts
    skey->set_data(tmpdbt);

    // Indicate the returned array is of size 2
    skey->size = 2;

    return (0);
}
```

Reading Secondary Databases

Like a primary database, you can read records from your secondary database either by using the `Db::get()` or `Db::pget()` methods, or by using a cursor on the secondary database. The main difference between reading secondary and primary databases is that when you read a secondary database record, the secondary record's data is not returned to you. Instead, the primary key and data corresponding to the secondary key are returned to you.

For example, assuming your secondary database contains keys related to a person's full name:

```
#include <db_cxx.h>
#include <string.h>

...


// The string to search for
char *search_name = "John Doe";
```

```
// Instantiate our Dbt's
Dbt key(search_name, strlen(search_name) + 1);
Dbt pkey, pdata; // Primary key and data


Db my_secondary_database(NULL, 0);
// Primary and secondary database opens omitted for brevity

// Returns the key from the secondary database, and the data from the
// associated primary database entry.
my_secondary_database.get(NULL, &key, &pdata, 0);

// Returns the key from the secondary database, and the key and data
// from the associated primary database entry.
my_secondary_database.pget(NULL, &key, &pkey, &pdata, 0);
```

Note that, just like a primary database, if your secondary database supports duplicate records then `Db::get()` and `Db::pget()` only return the first record found in a matching duplicates set. If you want to see all the records related to a specific secondary key, then use a cursor opened on the secondary database. Cursors are described in Using Cursors (page 33).

Deleting Secondary Database Records

In general, you will not modify a secondary database directly. In order to modify a secondary database, you should modify the primary database and simply allow DB to manage the secondary modifications for you.

However, as a convenience, you can delete secondary database records directly. Doing so causes the associated primary key/data pair to be deleted. This in turn causes DB to delete all secondary database records that reference the primary record.

You can use the `Db::del()` method to delete a secondary database record. Note that if your secondary database contains duplicate records, then deleting a record from the set of duplicates causes all of the duplicates to be deleted as well.

You can delete a secondary database record using the previously described mechanism only if the primary database is opened for write access.

For example:

```
#include <db_cxx.h>
#include <string.h>

...

Db my_database(NULL, 0); // Primary
Db my_index(NULL, 0);    // Secondary

// Open the primary
```

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

// Setup the secondary to use sorted duplicates.
// This is often desireable for secondary databases.
my_index.set_flags(DB_DUPSORT);

// 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_CREATE,         // 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.
                      0);            // Flags

// Name to delete
char *search_name = "John Doe";

// Get a search key
Dbt key(search_name, strlen(search_name) + 1);

// Now delete the secondary record. This causes the associated primary
// record to be deleted. If any other secondary databases have secondary
// records referring to the deleted primary record, then those secondary
// records are also deleted.
my_index.del(NULL, &key, 0);
```

Using Cursors with Secondary Databases

Just like cursors on a primary database, you can use cursors on secondary databases to iterate over the records in a secondary database. Like cursors used with primary databases, you can also use cursors with secondary databases to search for specific records in a database, to seek to the first or last record in the database, to get the next duplicate record, and so forth. For a complete description on cursors and their capabilities, see Using Cursors (page 33).

However, when you use cursors with secondary databases:

- Any data returned is the data contained on the primary database record referenced by the secondary record.
- You cannot use DB_GET_BOTH and related flags with Db::get() and a secondary database. Instead, you must use Db::pget(). Also, in that case the primary and secondary key given on the call to Db::pget() must match the secondary key and associated primary record key in order for that primary record to be returned as a result of the call.

For example, suppose you are using the databases, classes, and key extractors described in Implementing Key Extractors (page 50). Then the following searches for a person's name in the secondary database, and deletes all secondary and primary records that use that name.

```
#include <db_cxx.h>

...

Db my_database(NULL, 0);
Db my_index(NULL, 0);


// Get a cursor on the secondary database
Dbc *cursorp;
my_index.cursor(NULL, &cursorp, 0);


// Name to delete
char *search_name = "John Doe";

// Instantiate Dbts as normal
Dbt key(search_name, strlen(search_name) + 1);
Dbt data;


// Position the cursor
while (cursorp->get(&key, &data, DB_SET) == 0)
    cursorp->del(0);
```

Database Joins

If you have two or more secondary databases associated with a primary database, then you can retrieve primary records based on the intersection of multiple secondary entries. You do this using a join cursor.

Throughout this document we have presented a structure that stores information on grocery vendors. That structure is fairly simple with a limited number of data members, few of which would be interesting from a query perspective. But suppose, instead, that we were storing information on something with many more characteristics that can be queried, such as an automobile. In that case, you may be storing information such as color, number of doors, fuel mileage, automobile type, number of passengers, make, model, and year, to name just a few.

In this case, you would still likely be using some unique value to key your primary entries (in the United States, the automobile's VIN would be ideal for this purpose). You would then create a structure that identifies all the characteristics of the automobiles in your inventory.

To query this data, you might then create multiple secondary databases, one for each of the characteristics that you want to query. For example, you might create a secondary for color, another for number of doors, another for number of passengers, and so forth. Of course, you will need a unique key extractor function for each such secondary database. You do all of this using the concepts and techniques described throughout this chapter.

Once you have created this primary database and all interesting secondaries, what you have is the ability to retrieve automobile records based on a single characteristic. You can, for example, find all the automobiles that are red. Or you can find all the automobiles that have four doors. Or all the automobiles that are minivans.

The next most natural step, then, is to form compound queries, or joins. For example, you might want to find all the automobiles that are red, and that were built by Toyota, and that are minivans. You can do this using a join cursor.

Using Join Cursors

To use a join cursor:

- Open two or more cursors for secondary databases that are associated with the same primary database.
- Position each such cursor to the secondary key value in which you are interested. For example, to build on the previous description, the cursor for the color database is positioned to the `red` records while the cursor for the model database is positioned to the `minivan` records, and the cursor for the make database is positioned to `Toyota`.
- Create an array of cursors, and place in it each of the cursors that are participating in your join query. Note that this array must be null terminated.
- Obtain a join cursor. You do this using the `Db::join()` method. You must pass this method the array of secondary cursors that you opened and positioned in the previous steps.
- Iterate over the set of matching records until the return code is not `0`.
- Close your cursor.
- If you are done with them, close all your cursors.

For example:

```
#include <db_cxx.h>
#include <string.h>

...

// Exception handling omitted
```

```
int ret;

Db automotiveDB(NULL, 0);
Db automotiveColorDB(NULL, 0);
Db automotiveMakeDB(NULL, 0);
Db automotiveTypeDB(NULL, 0);

// Database and secondary database opens omitted for brevity.
// Assume a primary database:
//   automotiveDB
// Assume 3 secondary databases:
//   automotiveColorDB  -- secondary database based on automobile color
//   automotiveMakeDB  -- secondary database based on the manufacturer
//   automotiveTypeDB  -- secondary database based on automobile type

// Position the cursors
Dbc *color_curs;
automotiveColorDB.cursor(NULL, &color_curs, 0);
char *the_color = "red";
Dbt key(the_color, strlen(the_color) + 1);
Dbt data;
if ((ret = color_curs->get(&key, &data, DB_SET)) != 0) {
    // Error handling goes here
}

Dbc *make_curs;
automotiveMakeDB.cursor(NULL, &make_curs, 0);
char *the_make = "Toyota";
key.set_data(the_make);
key.set_size(strlen(the_make) + 1);
if ((ret = make_curs->get(&key, &data, DB_SET)) != 0) {
    // Error handling goes here
}

Dbc *type_curs;
automotiveTypeDB.cursor(NULL, &type_curs, 0);
char *the_type = "minivan";
key.set_data(the_type);
key.set_size(strlen(the_type) + 1);
if ((ret = type_curs->get(&key, &data, DB_SET)) != 0) {
    // Error handling goes here
}

// Set up the cursor array
Dbc *carray[4];
carray[0] = color_curs;
carray[1] = make_curs;
carray[2] = type_curs;
```

```
carray[3] = NULL;

// Create the join
Dbc *join_curs;
if ((ret = automotiveDB.join(carray, &join_curs, 0)) != 0) {
    // Error handling goes here
}

// Iterate using the join cursor
while ((ret = join_curs->get(&key, &data, 0)) == 0) {
    // Do interesting things with the key and data
}

// If we exited the loop because we ran out of records,
// then it has completed successfully.
if (ret == DB_NOTFOUND) {
     // Close all our cursors and databases as is appropriate,  and
     // then exit with a normal exit status (0).
}
```

Secondary Database Example

In previous chapters in this book, we built applications that load and display several DB databases. In this example, we will extend those examples to use secondary databases. Specifically:

- In Database Usage Example (page 24) we built an application that can open and load data into several databases. In Secondary Databases with example_database_load (page 58) we will extend that application to also open a secondary database for the purpose of indexing inventory item names.
- In Cursor Example (page 44) we built an application to display our inventory database (and related vendor information). In Secondary Databases with example_database_read (page 63) we will extend that application to show inventory records based on the index we cause to be loaded using `example_database_load`.

Secondary Databases with example_database_load

In order to update `example_database_load` to maintain an index of inventory item names, all we really need to do is:

1. Create a new database to be used as a secondary database.

2. Associate our new database to the inventory primary database.

We also need a function that can create our secondary keys for us.

Because DB maintains secondary databases for us; once this work is done we need not make any other changes to `example_database_load`.

Remember that you can find the complete implementation of these functions in:

```
DB_INSTALL/examples_cxx/getting_started
```

where *DB_INSTALL* is the location where you placed your DB distribution.

To begin, we go to `gettingStartedCommon.hpp` and we write our secondary key extractor function. This is a fairly trivial function to write because we have already done most of the work when we wrote the `InventoryData` class. Recall that when we wrote that class, we provided a constructor that accepts a pointer to a buffer and unpacks the contents of the buffer for us (see InventoryData Class (page 25) for the implementation). We now make use of that constructor.

```
// File: gettingStartedCommon.hpp
// Forward declarations
class Db;
class Dbt;

// Used to extract an inventory item's name from an
// inventory database record. This function is used to create
// keys for secondary database records.
int
get_item_name(Db *dbp, const Dbt *pkey, const Dbt *pdata, Dbt *skey)
{
    // Obtain the buffer location where the we placed the item's name. In
    // this example, the item's name is located in the primary data. It is
    // the first string in the buffer after the price (a double) and
    // the quantity (a long).
    size_t offset = sizeof(double) + sizeof(long);
    char * itemname = (char *)pdata->get_data() + offset;

    // unused
    (void)pkey;

    // If the offset is beyond the end of the data, then there is a
    // problem with the buffer contained in pdata, or there's a
    // programming error in how the buffer is marshalled/unmarshalled.
    // This should never happen!
    if ((u_int32_t)id.getBufferSize() != pdata->get_size()) {
        dbp->errx("get_item_name: buffer sizes do not match!");
        // When we return non-zero, the index record is not
        // added/updated.
        return (-1);
    }
    // Now set the secondary key's data to be the item name

    skey->set_data(itemname);
    skey->set_size(strlen(itemname) + 1);
```

```
    return (0);
};
```

Having written our key extractor callback, we now need to make a trivial update to our `MyDb` implementation. Because an item name is used by multiple inventory records, we need our secondary database to support sorted duplicates. We therefore must update `MyDb` to handle this detail.

The `MyDb` class definition changes to add a boolean to the constructor (remember that new code is in **bold**):

```
// File: MyDb.hpp
#include <db_cxx.h>

class MyDb
{
public:
    // Constructor requires a path to the database,
    // and a database name.
    MyDb(std::string &path, std::string &dbName,
         bool isSecondary = false);

    // Our destructor just calls our private close method.
    ~MyDb() { close(); }

    inline Db &getDb() {return db_;}

private:
    Db db_;
    std::string dbFileName_;
    u_int32_t cFlags_;

    // Make sure the default constructor is private
    // We don't want it used.
    MyDb() : db_(0, 0) {}

    // We put our database close activity here.
    // This is called from our destructor. In
    // a more complicated example, we might want
    // to make this method public, but a private
    // method is more appropriate for this example.
    void close();
};
```

And the implementation changes slightly to take advantage of the new boolean. Note that to save space, we just show the constructor where the code actually changes:

```
// File: MyDb.cpp
#include "MyDb.hpp"
```

```
// Class constructor. Requires a path to the location
// where the database is located, and a database name
MyDb::MyDb(std::string &path, std::string &dbName,
           bool isSecondary)
    : db_(NULL, 0),               // Instantiate Db object
      dbFileName_(path + dbName), // Database file name
      cFlags_(DB_CREATE)          // If the database doesn't yet exist,
                                  // allow it to be created.
{
    try
    {
        // Redirect debugging information to std::cerr
        db_.set_error_stream(&std::cerr);

        // If this is a secondary database, support
        // sorted duplicates
        if (isSecondary)
            db_.set_flags(DB_DUPSORT);

        // Open the database
        db_.open(NULL, dbFileName_.c_str(), NULL, DB_BTREE, cFlags_, 0);
    }
    // DbException is not a subclass of std::exception, so we
    // need to catch them both.
    catch(DbException &e)
    {
        std::cerr << "Error opening database: " << dbFileName_ << "\n";
        std::cerr << e.what() << std::endl;
    }
    catch(std::exception &e)
    {
        std::cerr << "Error opening database: " << dbFileName_ << "\n";
        std::cerr << e.what() << std::endl;
    }
}
```

That done, we can now update `example_database_load` to open our new secondary database and associate it to the inventory database.

To save space, we do not show the entire implementation for this program here. Instead, we show just the `main()` function, which is where all our modifications occur. To see the rest of the implementation for this command, see example_database_load (page 28).

```
// Loads the contents of vendors.txt and inventory.txt into
// Berkeley DB databases.
int
main(int argc, char *argv[])
{
    // Initialize the path to the database files
```

```
    std::string basename("./");
    std::string databaseHome("./");

    // Database names
    std::string vDbName("vendordb.db");
    std::string iDbName("inventorydb.db");
    std::string itemSDbName("itemname.sdb");

    // Parse the command line arguments here and determine
    // the location of the flat text files containing the
    // inventory data here. This step is omitted for clarity.

    //  Identify the full name for our input files, which should
    //  also include some path information.
    std::string inventoryFile = basename + "inventory.txt";
    std::string vendorFile = basename + "vendors.txt";

    try
    {
        // Open all databases.
        MyDb inventoryDB(databaseHome, iDbName);
        MyDb vendorDB(databaseHome, vDbName);
        MyDb itemnameSDB(databaseHome, itemSDbName, true);

        // Associate the primary and the secondary
        inventoryDB.getDb().associate(NULL,
                                      &(itemnameSDB.getDb()),
                                      get_item_name,
                                      0);

        // Load the vendor database
        loadVendorDB(vendorDB, vendorFile);

        // Load the inventory database
        loadInventoryDB(inventoryDB, inventoryFile);
    } catch(DbException &e) {
        std::cerr << "Error loading databases. " << std::endl;
        std::cerr << e.what() << std::endl;
        return(e.get_errno());
    } catch(std::exception &e) {
        std::cerr << "Error loading databases. " << std::endl;
        std::cerr << e.what() << std::endl;
        return(-1);
    }

    return(0);
} // End main
```

Note that the order in which we instantiate our `MyDb` class instances is important. In general you want to close a secondary database before closing the primary with which it is associated. This is particularly true for multi-threaded or multi-processed applications where the database closes are not single threaded. Even so, it is a good habit to adopt, even for simple applications such as this one. Here, we ensure that the databases are closed in the desired order by opening the secondary database last. This works because our `MyDb` objects are on the stack, and therefore the last one opened is the first one closed.

That completes our update to `example_database_load`. Now when this program is called, it will automatically index inventory items based on their names. We can then query for those items using the new index. We show how to do that in the next section.

Secondary Databases with example_database_read

In Cursor Example (page 44) we wrote an application that displays every inventory item in the Inventory database. In this section, we will update that example to allow us to search for and display an inventory item given a specific name. To do this, we will make use of the secondary database that `example_database_load` now creates.

The update to `example_database_read` is relatively modest. We need to open the new secondary database in exactly the same way was we do for `example_database_load`. We also need to add a command line parameter on which we can specify the item name, and we will need a new function in which we will perform the query and display the results.

To begin, we add a single forward declaration to the application, and update our usage function slightly:

```
// File: example_database_read.cpp
#include <iostream>
#include <fstream>
#include <cstdlib>

#include "MyDb.hpp"
#include "gettingStartedCommon.hpp"

// Forward declarations
int show_all_records(MyDb &inventoryDB, MyDb &vendorDB);
int show_item(MyDb &itemnameSDB, MyDb &vendorDB, std::string &itemName);
int show_vendor(MyDb &vendorDB, const char *vendor);
```

Next, we update `main()` to open the new secondary database and accept the new command line switch. We also need a new variable to contain the item's name.

The final update to the `main()` entails a little bit of logic to determine whether we want to display all available inventory items, or just the ones that match a name provided on the `-i` command line parameter.

```
// Displays all inventory items and the associated vendor record.
int
main (int argc, char *argv[])
```

```
{
    // Initialize the path to the database files
    std::string databaseHome("./");
    std::string itemName;

    // Database names
    std::string vDbName("vendordb.db");
    std::string iDbName("inventorydb.db");
    std::string itemSDbName("itemname.sdb");

    // Parse the command line arguments
    // Omitted for brevity

    try
    {
        // Open all databases.
        MyDb inventoryDB(databaseHome, iDbName);
        MyDb vendorDB(databaseHome, vDbName);
        MyDb itemnameSDB(databaseHome, itemSDbName, true);

        // Associate the secondary to the primary
        inventoryDB.getDb().associate(NULL,
                                      &(itemnameSDB.getDb()),
                                      get_item_name,
                                      0);

        if (itemName.empty())
        {
            show_all_records(inventoryDB, vendorDB);
        } else {
            show_item(itemnameSDB, vendorDB, itemName);
        }
    } catch(DbException &e) {
        std::cerr << "Error reading databases. " << std::endl;
        std::cerr << e.what() << std::endl;
        return(e.get_errno());
    } catch(std::exception &e) {
        std::cerr << "Error reading databases. " << std::endl;
        std::cerr << e.what() << std::endl;
        return(-1);
    }

    return(0);
} // End main
```

The only other thing that we need to add to the application is the implementation of the `show_item()` function.

In the interest of space, we refrain from showing the other functions used by this application. For their implementation, please see Cursor Example (page 44). Alternatively, you can see the entire implementation of this application in:

```
DB_INSTALL/examples_cxx/getting_started
```

where *DB_INSTALL* is the location where you placed your DB distribution.

```
// Shows the records in the inventory database that
// have a specific item name. For each inventory record
// shown, the appropriate vendor record is also displayed.
int
show_item(MyDb &itemnameSDB, MyDb &vendorDB, std::string &itemName)
{
    // Get a cursor to the itemname secondary db
    Dbc *cursorp;

    try {
        itemnameSDB.getDb().cursor(NULL, &cursorp, 0);

        // Get the search key. This is the name on the inventory
        // record that we want to examine.
        std::cout << "Looking for " << itemName << std::endl;
        Dbt key((void *)itemName.c_str(), itemName.length() + 1);
        Dbt data;

        // Position the cursor to the first record in the secondary
        // database that has the appropriate key.
        int ret = cursorp->get(&key, &data, DB_SET);
        if (!ret) {
            do {
                InventoryData inventoryItem(data.get_data());
                inventoryItem.show();

                show_vendor(vendorDB, inventoryItem.getVendor().c_str());

            } while(cursorp->get(&key, &data, DB_NEXT_DUP) == 0);
        } else {
            std::cerr << "No records found for '" << itemName
                      << "'" << std::endl;
        }
    } catch(DbException &e) {
        itemnameSDB.getDb().err(e.get_errno(), "Error in show_item");
        cursorp->close();
        throw e;
    } catch(std::exception &e) {
        itemnameSDB.getDb().errx("Error in show_item: %s", e.what());
        cursorp->close();
        throw e;
    }
```

```
    cursorp->close();
    return (0);
}
```

This completes our update to `example_inventory_read`. Using this update, you can now search for and show all inventory items that match a particular name. For example:

```
    example_inventory_read -i "Zulu Nut"
```

Chapter 6. Database Configuration

This chapter describes some of the database and cache configuration issues that you need to consider when building your DB database. In most cases, there is very little that you need to do in terms of managing your databases. However, there are configuration issues that you need to be concerned with, and these are largely dependent on the access method that you are choosing for your database.

The examples and descriptions throughout this document have mostly focused on the BTree access method. This is because the majority of DB applications use BTree. For this reason, where configuration issues are dependent on the type of access method in use, this chapter will focus on BTree only. For configuration descriptions surrounding the other access methods, see the *Berkeley DB Programmer's Reference Guide*.

Setting the Page Size

Internally, DB stores database entries on pages. Page sizes are important because they can affect your application's performance.

DB pages can be between 512 bytes and 64K bytes in size. The size that you select must be a power of 2. You set your database's page size using `Db::set_pagesize()`.

Note that a database's page size can only be selected at database creation time.

When selecting a page size, you should consider the following issues:

- Overflow pages.
- Locking
- Disk I/O.

These topics are discussed next.

Overflow Pages

Overflow pages are used to hold a key or data item that cannot fit on a single page. You do not have to do anything to cause overflow pages to be created, other than to store data that is too large for your database's page size. Also, the only way you can prevent overflow pages from being created is to be sure to select a page size that is large enough to hold your database entries.

Because overflow pages exist outside of the normal database structure, their use is expensive from a performance perspective. If you select too small of a page size, then your database will be forced to use an excessive number of overflow pages. This will significantly harm your application's performance.

For this reason, you want to select a page size that is at least large enough to hold multiple entries given the expected average size of your database entries. In BTree's case, for best results select a page size that can hold at least 4 such entries.

You can see how many overflow pages your database is using by using the `Db::stat()` method, or by examining your database using the `db_stat` command line utility.

Locking

Locking and multi-threaded access to DB databases is built into the product. However, in order to enable the locking subsystem and in order to provide efficient sharing of the cache between databases, you must use an *environment*. Environments and multi-threaded access are not fully described in this manual (see the Berkeley DB Programmer's Reference Manual for information), however, we provide some information on sizing your pages in a multi-threaded/multi-process environment in the interest of providing a complete discussion on the topic.

If your application is multi-threaded, or if your databases are accessed by more than one process at a time, then page size can influence your application's performance. The reason why is that for most access methods (Queue is the exception), DB implements page-level locking. This means that the finest locking granularity is at the page, not at the record.

In most cases, database pages contain multiple database records. Further, in order to provide safe access to multiple threads or processes, DB performs locking on pages as entries on those pages are read or written.

As the size of your page increases relative to the size of your database entries, the number of entries that are held on any given page also increase. The result is that the chances of two or more readers and/or writers wanting to access entries on any given page also increases.

When two or more threads and/or processes want to manage data on a page, lock contention occurs. Lock contention is resolved by one thread (or process) waiting for another thread to give up its lock. It is this waiting activity that is harmful to your application's performance.

It is possible to select a page size that is so large that your application will spend excessive, and noticeable, amounts of time resolving lock contention. Note that this scenario is particularly likely to occur as the amount of concurrency built into your application increases.

Oh the other hand, if you select too small of a page size, then that that will only make your tree deeper, which can also cause performance penalties. The trick, therefore, is to select a reasonable page size (one that will hold a sizeable number of records) and then reduce the page size if you notice lock contention.

You can examine the number of lock conflicts and deadlocks occurring in your application by examining your database environment lock statistics. Either use the `DbEnv::lock_stat()` `Environment.getLockStats()` method, or use the `db_stat` command line utility. The number of unavailable locks that your application waited for is held in the lock statistic's `st_lock_wait` field.

IO Efficiency

Page size can affect how efficient DB is at moving data to and from disk. For some applications, especially those for which the in-memory cache can not be large enough to hold the entire working dataset, IO efficiency can significantly impact application performance.

Most operating systems use an internal block size to determine how much data to move to and from disk for a single I/O operation. This block size is usually equal to the filesystem's block size. For optimal disk I/O efficiency, you should select a database page size that is equal to the operating system's I/O block size.

Essentially, DB performs data transfers based on the database page size. That is, it moves data to and from disk a page at a time. For this reason, if the page size does not match the I/O block size, then the operating system can introduce inefficiencies in how it responds to DB's I/O requests.

For example, suppose your page size is smaller than your operating system block size. In this case, when DB writes a page to disk it is writing just a portion of a logical filesystem page. Any time any application writes just a portion of a logical filesystem page, the operating system brings in the real filesystem page, over writes the portion of the page not written by the application, then writes the filesystem page back to disk. The net result is significantly more disk I/O than if the application had simply selected a page size that was equal to the underlying filesystem block size.

Alternatively, if you select a page size that is larger than the underlying filesystem block size, then the operating system may have to read more data than is necessary to fulfill a read request. Further, on some operating systems, requesting a single database page may result in the operating system reading enough filesystem blocks to satisfy the operating system's criteria for read-ahead. In this case, the operating system will be reading significantly more data from disk than is actually required to fulfill DB's read request.

While transactions are not discussed in this manual, a page size other than your filesystem's block size can affect transactional guarantees. The reason why is that page sizes larger than the filesystem's block size causes DB to write pages in block size increments. As a result, it is possible for a partial page to be written as the result of a transactional commit. For more information, see http://www.oracle.com/technology/documentation/berkeley-db/db/ref/transapp/reclimit.html.

Page Sizing Advice

Page sizing can be confusing at first, so here are some general guidelines that you can use to select your page size.

In general, and given no other considerations, a page size that is equal to your filesystem block size is the ideal situation.

If your data is designed such that 4 database entries cannot fit on a single page (assuming BTree), then grow your page size to accommodate your data. Once you've abandoned matching your filesystem's block size, the general rule is that larger page sizes are better.

The exception to this rule is if you have a great deal of concurrency occurring in your application. In this case, the closer you can match your page size to the ideal size needed for your application's data, the better. Doing so will allow you to avoid unnecessary contention for page locks.

Selecting the Cache Size

Cache size is important to your application because if it is set to too small of a value, your application's performance will suffer from too much disk I/O. On the other hand, if your cache is too large, then your application will use more memory than it actually needs. Moreover, if your application uses too much memory, then on most operating systems this can result in your application being swapped out of memory, resulting in extremely poor performance.

You select your cache size using either `Db::set_cachesize()`, or `DbEnv::set_cachesize()`, depending on whether you are using a database environment or not. You cache size must be a power of 2, but it is otherwise limited only by available memory and performance considerations.

Selecting a cache size is something of an art, but fortunately you can change it any time, so it can be easily tuned to your application's changing data requirements. The best way to determine how large your cache needs to be is to put your application into a production environment and watch to see how much disk I/O is occurring. If your application is going to disk quite a lot to retrieve database records, then you should increase the size of your cache (provided that you have enough memory to do so).

You can use the `db_stat` command line utility with the `-m` option to gauge the effectiveness of your cache. In particular, the number of pages found in the cache is shown, along with a percentage value. The closer to 100% that you can get, the better. If this value drops too low, and you are experiencing performance problems, then you should consider increasing the size of your cache, assuming you have memory to support it.

BTree Configuration

In going through the previous chapters in this book, you may notice that we touch on some topics that are specific to BTree, but we do not cover those topics in any real detail. In this section, we will discuss configuration issues that are unique to BTree.

Specifically, in this section we describe:

- Allowing duplicate records.
- Setting comparator callbacks.

Allowing Duplicate Records

BTree databases can contain duplicate records. One record is considered to be a duplicate of another when both records use keys that compare as equal to one another.

By default, keys are compared using a lexicographical comparison, with shorter keys collating higher than longer keys. You can override this default using the `Db::set_bt_compare()` method. See the next section for details.

By default, DB databases do not allow duplicate records. As a result, any attempt to write a record that uses a key equal to a previously existing record results in the previously existing record being overwritten by the new record.

Allowing duplicate records is useful if you have a database that contains records keyed by a commonly occurring piece of information. It is frequently necessary to allow duplicate records for secondary databases.

For example, suppose your primary database contained records related to automobiles. You might in this case want to be able to find all the automobiles in the database that are of a particular color, so you would index on the color of the automobile. However, for any given color there will probably be multiple automobiles. Since the index is the secondary key, this means that multiple secondary database records will share the same key, and so the secondary database must support duplicate records.

Sorted Duplicates

Duplicate records can be stored in sorted or unsorted order. You can cause DB to automatically sort your duplicate records by specifying the `DB_DUPSORT` flag at database creation time.

If sorted duplicates are supported, then the sorting function specified on `Db::set_dup_compare()` is used to determine the location of the duplicate record in its duplicate set. If no such function is provided, then the default lexicographical comparison is used.

Unsorted Duplicates

For performance reasons, BTrees should always contain sorted records. (BTrees containing unsorted entries must potentially spend a great deal more time locating an entry than does a BTree that contains sorted entries). That said, DB provides support for suppressing automatic sorting of duplicate records because it may be that your application is inserting records that are already in a sorted order.

That is, if the database is configured to support unsorted duplicates, then the assumption is that your application will manually perform the sorting. In this event, expect to pay a significant performance penalty. Any time you place records into the database in a sort order not know to DB, you will pay a performance penalty

That said, this is how DB behaves when inserting records into a database that supports non-sorted duplicates:

- If your application simply adds a duplicate record using `Db::put()`, then the record is inserted at the end of its sorted duplicate set.
- If a cursor is used to put the duplicate record to the database, then the new record is placed in the duplicate set according to the flags that are provided on the `Dbc::put()` method. The relevant flags are:
  - `DB_AFTER`

    The data provided on the call to `Dbc::put()` is placed into the database as a duplicate record. The key used for this operation is the key used for the record to which the cursor currently refers. Any key provided on the call to `Dbc::put()` is therefore ignored.

    The duplicate record is inserted into the database immediately after the cursor's current position in the database.

    This flag is ignored if sorted duplicates are supported for the database.
  - `DB_BEFORE`

    Behaves the same as `DB_AFTER` except that the new record is inserted immediately before the cursor's current location in the database.
  - `DB_KEYFIRST`

    If the key provided on the call to `Dbc::put()` already exists in the database, and the database is configured to use duplicates without sorting, then the new record is inserted as the first entry in the appropriate duplicates list.
  - `DB_KEYLAST`

    Behaves identically to `DB_KEYFIRST` except that the new duplicate record is inserted as the last record in the duplicates list.

Configuring a Database to Support Duplicates

Duplicates support can only be configured at database creation time. You do this by specifying the appropriate flags to `Db::set_flags()` before the database is opened for the first time.

The flags that you can use are:

- `DB_DUP`

  The database supports non-sorted duplicate records.
- `DB_DUPSORT`

  The database supports sorted duplicate records.

The following code fragment illustrates how to configure a database to support sorted duplicate records:

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

Db db(NULL, 0);
const char *file_name = "myd.db";

try {
    // Configure the database for sorted duplicates
    db.set_flags(DB_DUPSORT);

    // Now open the database
    db.open(NULL,       // Txn pointer
            file_name,  // File name
            NULL,       // Logical db name (unneeded)
            DB_BTREE,   // Database type (using btree)
            DB_CREATE,  // Open flags
            0);         // File mode. Using defaults
} catch(DbException &e) {
    db.err(e.get_errno(), "Database '%s' open failed.", file_name);
} catch(std::exception &e) {
    db.errx("Error opening database: %s : %s\n", file_name, e.what());
}

...

try {
    db.close(0);
} catch(DbException &e) {
    db.err(e.get_errno(), "Database '%s' close failed.", file_name);
} catch(std::exception &e) {
    db.errx("Error closing database: %s : %s\n", file_name, e.what());
}
```

Setting Comparison Functions

By default, DB uses a lexicographical comparison function where shorter records collate before longer records. For the majority of cases, this comparison works well and you do not need to manage it in any way.

However, in some situations your application's performance can benefit from setting a custom comparison routine. You can do this either for database keys, or for the data if your database supports sorted duplicate records.

Some of the reasons why you may want to provide a custom sorting function are:

- Your database is keyed using strings and you want to provide some sort of language-sensitive ordering to that data. Doing so can help increase the locality of reference that allows your database to perform at its best.

- You are using a little-endian system (such as x86) and you are using integers as your database's keys. Berkeley DB stores keys as byte strings and little-endian integers do not sort well when viewed as byte strings. There are several solutions to this problem, one being to provide a custom comparison function. See http://www.oracle.com/technology/documentation/berkeley-db/db/ref/am_misc/faq.html for more information.
- You you do not want the entire key to participate in the comparison, for whatever reason. In this case, you may want to provide a custom comparison function so that only the relevant bytes are examined.

Creating Comparison Functions

You set a BTree's key comparison function using `Db::set_bt_compare()`. You can also set a BTree's duplicate data comparison function using `Db::set_dup_compare()`.

You cannot use these methods after the database has been opened. Also, if the database already exists when it is opened, the function provided to these methods must be the same as that historically used to create the database or corruption can occur.

The value that you provide to the `set_bt_compare()` method is a pointer to a function that has the following signature:

```
int (*function)(Db *db, const Dbt *key1, const Dbt *key2)
```

This function must return an integer value less than, equal to, or greater than 0. If key1 is considered to be greater than key2, then the function must return a value that is greater than 0. If the two are equal, then the function must return 0, and if the first key is less than the second then the function must return a negative value.

The function that you provide to `set_dup_compare()` works in exactly the same way, except that the `Dbt` parameters hold record data items instead of keys.

For example, an example routine that is used to sort integer keys in the database is:

```
int
compare_int(Db *dbp, const Dbt *a, const Dbt *b)
{
    int ai, bi;

    // Returns:
    // < 0 if a < b
    // = 0 if a = b
    // > 0 if a > b
    memcpy(&ai, a->get_data(), sizeof(int));
    memcpy(&bi, b->get_data(), sizeof(int));
    return (ai - bi);
}
```

Note that the data must first be copied into memory that is appropriately aligned, as Berkeley DB does not guarantee any kind of alignment of the underlying data, including for comparison routines. When writing comparison routines, remember that databases created on machines

of different architectures may have different integer byte orders, for which your code may need to compensate.

To cause DB to use this comparison function:

```
#include <db_cxx.h>
#include <string.h>

...

Db db(NULL, 0);

// Set up the btree comparison function for this database
db.set_bt_compare(compare_int);

// Database open call follows sometime after this.
```