Oracle Berkeley DB

Getting Started with Berkeley DB for C

12c Release 1

Library Version 12.1.6.2

ORACLE®

BERKELEY DB

Legal Notice

This documentation is distributed under an open source license. You may review the terms of this license at: http://www.oracle.com/technetwork/database/berkeleydb/downloads/oslicense-093458.html

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

Other names may be trademarks of their respective owners.

To obtain a copy of this document's original source code, please submit a request to the Oracle Technology Network forum at: https://forums.oracle.com/forums/forum.jspa?forumID=271

Published 1/30/2019

Table of Contents

Preface

Welcome to Berkeley DB (DB). This document introduces Berkeley DB 12*c* Release 1, which provides DB library version 12.1.6.2.

This document 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:

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

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

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

```
/* File: gettingstarted_common.h */
typedef struct stock_dbs {
    DB *inventory_dbp; /* Database containing inventory information */
    DB *vendor_dbp;    /* Database containing vendor information */

    char *db_home_dir;        /* Directory containing the database files */
    char *inventory_db_name; /* Name of the inventory database */
    char *vendor_db_name;    /* Name of the vendor database */
} STOCK_DBS;
```

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 stock_dbs {
    DB *inventory_dbp; /* Database containing inventory information */
    DB *vendor_dbp;    /* Database containing vendor information */
    DB *itemname_sdbp; /* Index based on the item name index */
    char *db_home_dir;        /* Directory containing the database files */
    char *itemname_db_name;  /* Itemname secondary database */
    char *inventory_db_name; /* Name of the inventory database */
    char *vendor_db_name;    /* Name of the vendor database */
} STOCK_DBS;
```

Note

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

- Berkeley DB Getting Started with Replicated Applications for C

- Berkeley DB Programmer's Reference Guide

- Berkeley DB Installation and Build Guide

- Berkeley DB Getting Started with the SQL APIs

- Berkeley DB C API Reference Guide

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

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

Contact Us

You can post your comments and questions at the Oracle Technology (OTN) forum for Oracle Berkeley DB at: https://forums.oracle.com/forums/forum.jspa?forumID=271, or for Oracle Berkeley DB High Availability at: https://forums.oracle.com/forums/forum.jspa?forumID=272.

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

Chapter 1. Introduction to Berkeley DB

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.

Note

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/c/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. An individual DB database can store up to 256 terabytes of data. By using multiple databases, it is possible to use DB to store and manage petabytes of information. Within a single database, individual record keys or record data can be used to 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 85) 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 some network-shared drives (NFS or Windows networks shares, for example). For more information, see the *Remote filesystems* section in *The Berkeley DB Environment* chapter of the *Berkeley DB Programmer's Reference Guide*.

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.

Error Returns

Before continuing, it is useful to spend a few moments on error returns in DB.

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.

DB also offers programmatic support for displaying error return values. First, the `db_strerror` function returns a pointer to the error message corresponding to any DB error return, similar to the ANSI C `strerror` function, but is able to handle both system error returns and DB-specific return values.

Second, there are two error functions, `DB->err` and `DB->errx`. These functions work like the ANSI C `printf` function, taking a printf-style format string and argument list, and optionally appending the standard error string to a message constructed from the format string and other arguments.

Getting and Using DB

You can obtain DB by visiting the Berkeley DB download page: http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/downloads/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 structures (see Database Records (page 21) for details on this structure). So, fundamentally, using a DB database involves putting, getting, and deleting database records, which in turns involves efficiently managing information contained in DBT structures. The next several chapters of this book are dedicated to those activities.

Opening Databases

To open a database, you must first use the db_create() function to initialize a DB handle. Once you have initialized the DB handle, you use its open() method to open the database.

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.h>

...

DB *dbp;            /* DB structure handle */
u_int32_t flags;    /* database open flags */
int ret;            /* function return value */

/* Initialize the structure. This
 * database is not opened in an environment,
 * so the environment pointer is NULL. */
ret = db_create(&dbp, NULL, 0);
if (ret != 0) {
   /* Error handling goes here */
}

/* Database open flags */
flags = DB_CREATE;    /* If the database does not exist,
                       * create it.*/

/* open the database */
ret = dbp->open(dbp,        /* DB structure pointer */
                NULL,       /* Transaction pointer */
                "my_db.db", /* On-disk file that holds the database. */
                NULL,       /* Optional logical database name */
                DB_BTREE,   /* Database access method */
                flags,      /* Open flags */
```

```
                0);              /* File mode (using defaults) */
if (ret != 0) {
  /* Error handling 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. It is recommended that you close any open cursors 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 39) 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 25).

The following code fragment illustrates a database close:

```
#include <db.h>
...
DB *dbp;            /* DB struct handle */
...

/*
 * Database open and access operations
 * happen here.
 */


...

/* When we're done with the database, close it. */
if (dbp != NULL)
    dbp->close(dbp, 0);
```

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 Reference Guide*.

Note

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.h>
  ...
  DB *dbp;
  u_int32_t open_flags;

  /* Database open and subsequent operations omitted for clarity */

  dbp->get_open_flags(dbp, &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.h>
...
DB *dbp;

/* Database handle creation omitted for clarity */

dbp->remove(dbp,                  /* Database pointer */
            "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.h>
...
DB *dbp;

/* Database handle creation omitted for clarity */

dbp->rename(dbp,                  /* Database pointer */
            "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 structure offers several useful methods.

- `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()`. 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.

  ## Note

  Equivalent methods exist for informational messages. See the `set_msgcall()`, `set_msgfile()`, `set_msgpfx()`, and `msg()` methods for details.

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 DB_ENV *dbenv, const char *error_prefix,
 const 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.h>
#include <stdio.h>


...

DB *dbp;
int ret;

/*
 * Create a database and initialize it for error
 * reporting.
 */
ret = db_create(&dbp, NULL, 0);
if (ret != 0) {
        fprintf(stderr, "%s: %s\n", "my_program",
          db_strerror(ret));
        return(ret);
}

/* Set up error handling for this database */
dbp->set_errcall(dbp, my_error_handler);
dbp->set_errpfx(dbp, "my_example_program");
```

And to issue an error message:

```
ret = dbp->open(dbp,
                NULL,
                "mydb.db",
                NULL,
                DB_BTREE,
                DB_CREATE,
                0);
if (ret != 0) {
    dbp->err(dbp, ret,
      "Database open failed: %s", "mydb.db");
    return(ret);
}
```

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 create the environment handle using , and then 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.h>
...
DB_ENV *myEnv;              /* Env structure handle */
DB *dbp;                    /* DB structure handle */
u_int32_t db_flags;         /* database open flags */
u_int32_t env_flags;        /* env open flags */
int ret;                    /* function return value */

/*
   Create an environment object and initialize it for error
   reporting.
*/
ret = db_env_create(&myEnv, 0);
if (ret != 0) {
    fprintf(stderr, "Error creating env handle: %s\n", db_strerror(ret));
    return -1;
}

/* Open the environment. */
env_flags = DB_CREATE |     /* If the environment does not exist,
                             * create it. */
            DB_INIT_MPOOL; /* Initialize the in-memory cache. */

ret = myEnv->open(myEnv,    /* DB_ENV ptr */
  "/export1/testEnv",       /* env home directory */
  env_flags,                /* Open flags */
  0);                       /* File mode (default) */
if (ret != 0) {
    fprintf(stderr, "Environment open failed: %s", db_strerror(ret));
    return -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:

```
/*
 * Initialize the DB structure. Pass the pointer
 * to the environment in which this DB is opened.
 */
ret = db_create(&dbp, myEnv, 0);
if (ret != 0) {
  /* Error handling goes here */
}

/* Database open flags */
db_flags = DB_CREATE;    /* If the database does not exist,
                          * create it.*/

/* open the database */
ret = dbp->open(dbp,         /* DB structure pointer */
                NULL,        /* Transaction pointer */
                "my_db.db", /* On-disk file that holds the database. */
                NULL,        /* Optional logical database name */
                DB_BTREE,    /* Database access method */
                db_flags,    /* Open flags */
                0);          /* File mode (using defaults) */
if (ret != 0) {
  /* Error handling goes here */
}
```

When you are done with an environment, you must close it. It is recommended that before closing an environment, you close any open databases.

```
/*
* Close the database and environment
*/

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

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

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 some important structures and functions 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/c/getting_started
```

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

Example 2.1. The stock_db Structure

To begin, we create a structure that we will use to hold all our database pointers and database names:

```
/* File: gettingstarted_common.h */
#include <db.h>

typedef struct stock_dbs {
    DB *inventory_dbp; /* Database containing inventory information */
    DB *vendor_dbp;    /* Database containing vendor information */

    char *db_home_dir;       /* Directory containing the database files */
    char *inventory_db_name; /* Name of the inventory database */
    char *vendor_db_name;    /* Name of the vendor database */
} STOCK_DBS;

/* Function prototypes */
int databases_setup(STOCK_DBS *, const char *, FILE *);
int databases_close(STOCK_DBS *);
void initialize_stockdbs(STOCK_DBS *);
int open_database(DB **, const char *, const char *,
    FILE *);
void set_db_filenames(STOCK_DBS *my_stock);
```

Example 2.2. The stock_db Utility Functions

Before continuing, we want some utility functions that we use to make sure the stock_db structure is in a sane state before using it. One is a simple function that initializes all the structure's pointers to a useful default. The second is more interesting in that it is used to place a common path on all our database names so that we can explicitly identify where all the database files should reside.

```
/* File: gettingstarted_common.c */
#include "gettingstarted_common.h"

/* Initializes the STOCK_DBS struct.*/
void
initialize_stockdbs(STOCK_DBS *my_stock)
{
    my_stock->db_home_dir = DEFAULT_HOMEDIR;
    my_stock->inventory_dbp = NULL;
    my_stock->vendor_dbp = NULL;

    my_stock->inventory_db_name = NULL;
    my_stock->vendor_db_name = NULL;
```

```
}

/* Identify all the files that will hold our databases. */
void
set_db_filenames(STOCK_DBS *my_stock)
{
    size_t size;

    /* Create the Inventory DB file name */
    size = strlen(my_stock->db_home_dir) + strlen(INVENTORYDB) + 1;
    my_stock->inventory_db_name = malloc(size);
    snprintf(my_stock->inventory_db_name, size, "%s%s",
      my_stock->db_home_dir, INVENTORYDB);

    /* Create the Vendor DB file name */
    size = strlen(my_stock->db_home_dir) + strlen(VENDORDB) + 1;
    my_stock->vendor_db_name = malloc(size);
    snprintf(my_stock->vendor_db_name, size, "%s%s",
      my_stock->db_home_dir, VENDORDB);
}
```

Example 2.3. open_database() Function

We are opening multiple databases, and we are opening those databases using identical flags
and error reporting settings. It is therefore worthwhile to create a function that performs this
operation for us:

```
/* File: gettingstarted_common.c */

/* Opens a database */
int
open_database(DB **dbpp,           /* The DB handle that we are opening */
    const char *file_name,         /* The file in which the db lives */
    const char *program_name,      /* Name of the program calling this
                                     * function */
    FILE *error_file_pointer)      /* File where we want error messages
                                      sent */
{
    DB *dbp;      /* For convenience */
    u_int32_t open_flags;
    int ret;

    /* Initialize the DB handle */
    ret = db_create(&dbp, NULL, 0);
    if (ret != 0) {
        fprintf(error_file_pointer, "%s: %s\n", program_name,
                db_strerror(ret));
        return(ret);
    }
```

```
      /* Point to the memory malloc'd by db_create() */
      *dbpp = dbp;

      /* Set up error handling for this database */
      dbp->set_errfile(dbp, error_file_pointer);
      dbp->set_errpfx(dbp, program_name);

      /* Set the open flags */
      open_flags = DB_CREATE;

      /* Now open the database */
      ret = dbp->open(dbp,          /* Pointer to the database */
                      NULL,         /* Txn pointer */
                      file_name,    /* File name */
                      NULL,         /* Logical db name (unneeded) */
                      DB_BTREE,     /* Database type (using btree) */
                      open_flags,   /* Open flags */
                      0);           /* File mode. Using defaults */
      if (ret != 0) {
          dbp->err(dbp, ret, "Database '%s' open failed.", file_name);
          return(ret);
      }

      return (0);
}
```

Example 2.4. The databases_setup() Function

Now that we have our open_database() function, we can use it to open a database. We now
create a simple function that will open all our databases for us.

```
/* opens all databases */
int
databases_setup(STOCK_DBS *my_stock, const char *program_name,
  FILE *error_file_pointer)
{
    int ret;

    /* Open the vendor database */
    ret = open_database(&(my_stock->vendor_dbp),
      my_stock->vendor_db_name,
      program_name, error_file_pointer);
    if (ret != 0)
        /*
         * Error reporting is handled in open_database() so just return
         * the return code here.
         */
        return (ret);

    /* Open the inventory database */
```

```
        ret = open_database(&(my_stock->inventory_dbp),
          my_stock->inventory_db_name,
          program_name, error_file_pointer);
        if (ret != 0)
            /*
             * Error reporting is handled in open_database() so just return
             * the return code here.
             */
            return (ret);

        printf("databases opened successfully\n");
        return (0);
 }
```

Example 2.5. The databases_close() Function

Finally, it is useful to have a function that can close all our databases for us:

```
/* Closes all the databases. */
int
databases_close(STOCK_DBS *my_stock)
{
    int ret;
    /*
     * Note that closing a database automatically flushes its cached data
     * to disk, so no sync is required here.
     */

    if (my_stock->inventory_dbp != NULL) {
        ret = my_stock->inventory_dbp->close(my_stock->inventory_dbp, 0);
        if (ret != 0)
            fprintf(stderr, "Inventory database close failed: %s\n",
              db_strerror(ret));
    }

    if (my_stock->vendor_dbp != NULL) {
        ret = my_stock->vendor_dbp->close(my_stock->vendor_dbp, 0);
        if (ret != 0)
            fprintf(stderr, "Vendor database close failed: %s\n",
              db_strerror(ret));
    }

    printf("databases closed.\n");
    return (0);
}
```

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 structures. Therefore, to access a DB record, you need two such structures, one for the key and one for the data.

DBT structures provide a void * field that you use to point to your data, and another field that identifies the data length. They can therefore be used to store anything from simple primitive data to complex structures 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 structures — one for the key and another for the data.

To store a database record where the key and/or the data are primitive data (int, float, and so forth), or where the key and/or the data contain an array, we need only to point to the memory location where that data resides and identify its length. For example:

```c
#include <db.h>
#include <string.h>

...

DBT key, data;
float money = 122.45;
char *description = "Grocery bill.";

/* Zero out the DBTs before using them. */
memset(&key, 0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));

key.data = &money;
key.size = sizeof(float);

data.data = description;
data.size = strlen(description) + 1;
```

To retrieve the record, simply assign the void * returned in the DBT to the appropriate variable.

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.h>
#include <string.h>

...

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

/* Initialize the DBTs */
memset(&key, 0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));

key.data = &money;
key.ulen = sizeof(float);
key.flags = DB_DBT_USERMEM;

/* Database retrieval code goes here */

/*
 * Money is set into the memory that we supplied.
 */
description = data.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 DBT->put() and DBT->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 39).

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 85) 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` structures. 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.h>
#include <string.h>

...

char *description = "Grocery bill.";
DBT key, data;
DB *my_database;
int ret;
float money;

/* Database open omitted for clarity */

money = 122.45;

/* Zero out the DBTs before using them. */
memset(&key, 0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));

key.data = &money;
key.size = sizeof(float);

data.data = description;
data.size = strlen(description) +1;

ret = my_database->put(my_database, NULL, &key, &data, DB_NOOVERWRITE);
if (ret == DB_KEYEXIST) {
    my_database->err(my_database, 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 39).

(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.h>
#include <string.h>

...
#define DESCRIPTION_SIZE 199
DBT key, data;
DB *my_database;
float money;
char description[DESCRIPTION_SIZE + 1];

/* Database open omitted for clarity */

money = 122.45;

/* Zero out the DBTs before using them. */
memset(&key, 0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));

key.data = &money;
key.size = sizeof(float);

data.data = description;
data.ulen = DESCRIPTION_SIZE + 1;
data.flags = DB_DBT_USERMEM;
my_database->get(my_database, 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 39).

You can also delete every record in the database by using DB->truncate().

For example:

```c
#include <db.h>
#include <string.h>

...

DBT key;
DB *my_database;
float money = 122.45;

/* Database open omitted for clarity */

/* Zero out the DBTs before using them. */
memset(&key, 0, sizeof(DBT));

key.data = &money;
key.size = sizeof(float);

my_database->del(my_database, 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()`.

Note

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.

Using C Structures with DB

Storing data in structures is a handy way to pack varied types of information into each database record. DB databases are sometimes thought of as a two column table where column 1 is the key and column 2 is the data. By using structures, you can effectively turn this table into *n* columns where *n-1* columns are contained in the structure.

So long as a C structure contains fields that are not pointers, you can safely store and retrieve them in the same way as you would any primitive datatype. The following code fragment illustrates this:

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

typedef struct my_struct {
    int id;
    char familiar_name[MAXLINE]; /* Some suitably large value */
    char surname[MAXLINE];
} MY_STRUCT;


...

DBT key, data;
DB *my_database;
MY_STRUCT user;
char *fname = "David";
char *sname = "Rider";

/* Database open omitted for clarity */

user.id = 1;
strncpy(user.familiar_name, fname, strlen(fname)+1);
```

```
strncpy(user.surname, sname, strlen(sname)+1);

/* Zero out the DBTs before using them. */
memset(&key, 0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));

key.data = &(user.id);
key.size = sizeof(int);

data.data = &user;
data.size = sizeof(MY_STRUCT);

my_database->put(my_database, NULL, &key, &data, DB_NOOVERWRITE);
```

To retrieve the structure, make sure you supply your own memory. The reason why is that like real numbers, some systems require structures to be aligned in a specific way. Because it is possible that the memory DB provides is not aligned properly, for safest result simply use your own memory:

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

...

DBT key, data;
DB *my_database;
MY_STRUCT user;

/* Database open omitted for clarity */

/* Zero out the DBTs before using them. */
memset(&key, 0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));

/* Initialize the structure */
memset(&user, 0, sizeof(MY_STRUCT));
user.id = 1;

key.data = &user.id;
key.size = sizeof(int);

/* Use our memory to retrieve the structure */
data.data = &user;
data.ulen = sizeof(MY_STRUCT);
data.flags = DB_DBT_USERMEM;

my_database->get(my_database, NULL, &key, &data, 0);

printf("Familiar name: %s\n", user.familiar_name);
printf("Surname: %s\n", user.surname);
```

Be aware that while this is the easiest way to manage structures stored in DB databases, this approach does suffer from causing your database to be larger than is strictly necessary. Each structure stored in the database is of a fixed size, and you do not see any space savings from storing a (for example) 5 character surname versus a 20 character surname.

For a simple example such as this, the padding stored with each record is probably not critical. However, if you are storing structures that contain a very large number of character arrays, or if you are simply storing millions of records, then you may want to avoid this approach. The wasted space in each record will only serve to make your databases larger than need be, which will in turn require a larger cache and more disk I/O than you would ordinarily need.

An alternative approach is described next.

C Structures with Pointers

It is often necessary in C structures to use fields that are pointers to dynamically allocated memory. This is particularly true if you want to store character strings (or any kind of an array for that matter), and you want to avoid any overhead caused by pre-designating the size of the array.

When storing structures like these you need to make sure that all of the data pointed to and contained by the structure is lined up in a single contiguous block of memory. Remember that DB stores data located at a specific address and of a particular size. If your structure includes fields that are pointing to dynamically allocated memory, then the data that you want to store can be located in different, not necessarily contiguous, locations on the heap.

The easiest way to solve this problem is to pack your data into a single memory location and then store the data in that location. (This process is sometimes called *marshalling the data*.) For example:

```
#include <db.h>
#include <string.h>
#include <stdlib.h>

typedef struct my_struct {
    int id;
    char *familiar_name;
    char *surname;
} MY_STRUCT;

...

DBT key, data;
DB *my_database;
MY_STRUCT user;
int buffsize, bufflen;
char fname[ ] = "Pete";
char sname[10];
char *databuff;

strncpy(sname, "Oar", strlen("Oar")+1);
```

```
/* Database open omitted for clarity */

user.id = 1;
user.familiar_name = fname;
user.surname = sname;

/* Some of the structure's data is on the stack, and
 * some is on the heap. To store this structure's data, we
 * need to marshall it -- pack it all into a single location
 * in memory.
 */

/* Get the buffer */
buffsize = sizeof(int) +
  (strlen(user.familiar_name) + strlen(user.surname) + 2);
databuff = malloc(buffsize);
memset(databuff, 0, buffsize);

/* copy everything to the buffer */
memcpy(databuff, &(user.id), sizeof(int));
bufflen = sizeof(int);

memcpy(databuff + bufflen, user.familiar_name,
  strlen(user.familiar_name) + 1);
bufflen += strlen(user.familiar_name) + 1;

memcpy(databuff + bufflen, user.surname,
  strlen(user.surname) + 1);
bufflen += strlen(user.surname) + 1;

/* Now store it */

/* Zero out the DBTs before using them. */
memset(&key, 0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));

key.data = &(user.id);
key.size = sizeof(int);

data.data = databuff;
data.size = bufflen;

my_database->put(my_database, NULL, &key, &data, DB_NOOVERWRITE);
free(sname);
free(databuff);
```

To retrieve the stored structure:

```
#include <db.h>
```

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

typedef struct my_struct {
    char *familiar_name;
    char *surname;
    int id;
} MY_STRUCT;

...

int id;
DBT key, data;
DB *my_database;
MY_STRUCT user;
char *buffer;

/* Database open omitted for clarity */


/* Zero out the DBTs before using them. */
memset(&key, 0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));

id = 1;
key.data = &id;
key.size = sizeof(int);

my_database->get(my_database, NULL, &key, &data, 0);

/*
 * Some compilers won't allow pointer arithmetic on void *'s,
 * so use a char * instead.
 */
buffer = data.data;

user.id = *((int *)data.data);
user.familiar_name = buffer + sizeof(int);
user.surname = buffer + sizeof(int) + strlen(user.familiar_name) + 1;
```

Database Usage Example

In we created several functions that will open and close the databases that we will use for our inventory application. We now make use of those functions to load inventory data into the two databases that we use for this application.

Again, remember that you can find the complete implementation for these functions in:
 DB_INSTALL/examples/c/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 create 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.

For the vendor data, we add the VENDOR structure to the same file as holds our STOCK_DBS 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.

Note that for the inventory data, we will store the data by marshaling it into a buffer, described below.

```c
/* File: gettingstarted_common.h */
#include <db.h>


...


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. example_database_load

Our initial sample application will load 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/c/getting_started
```

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

We begin with the normal include directives and forward declarations:

```c
/* example_database_load.c */
#include "gettingstarted_common.h"


/* Forward declarations */
int load_vendors_database(STOCK_DBS, char *);
size_t pack_string(char *, char *, size_t);
int load_inventory_database(STOCK_DBS, char *);
```

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[])
{
    STOCK_DBS my_stock;
    int ret;
    size_t size;
    char *basename, *inventory_file, *vendor_file;

    /* Initialize the STOCK_DBS struct */
    initialize_stockdbs(&my_stock);

    /*
     * Initialize the base path. This path is used to
     * identify the location of the flat-text data
     * input files.
     */
    basename = "./";

    /*
     * 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 files that will hold our databases
     * This function uses information obtained from the
     * command line to identify the directory in which
     * the database files reside.
     */
    set_db_filenames(&my_stock);

    /* Find our input files */
    size = strlen(basename) + strlen(INVENTORY_FILE) + 1;
    inventory_file = malloc(size);
    snprintf(inventory_file, size, "%s%s", basename, INVENTORY_FILE);

    size = strlen(basename) + strlen(VENDORS_FILE) + 1;
    vendor_file = malloc(size);
    snprintf(vendor_file, size, "%s%s", basename, VENDORS_FILE);

    /* Open all databases */
    ret = databases_setup(&my_stock, "example_database_load", stderr);
```

```
        if (ret) {
                fprintf(stderr, "Error opening databases.\n");
                databases_close(&my_stock);
                return (ret);
        }

        ret = load_vendors_database(my_stock, vendor_file);
        if (ret) {
            fprintf(stderr, "Error loading vendors database.\n");
            databases_close(&my_stock);
            return (ret);
        }
        ret = load_inventory_database(my_stock, inventory_file);
        if (ret) {
            fprintf(stderr, "Error loading inventory database.\n");
            databases_close(&my_stock);
            return (ret);
        }

        /* close our environment and databases */
        databases_close(&my_stock);

        printf("Done loading databases.\n");
        return (0);
 }
```

Notice that there is not a lot to this function because we have pushed off all the database activity to other places. In particular our databases are all opened and configured in databases_setup() which we implemented in The databases_setup() Function (page 19).

Next we show the implementation of load_vendors_database(). We load this data by scanning (line by line) the contents of the vendors.txt 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.
  */
 int
 load_vendors_database(STOCK_DBS my_stock, char *vendor_file)
 {
     DBT key, data;
     char buf[MAXLINE];
     FILE *ifp;
     VENDOR my_vendor;
```

```
        /* Open the vendor file for read access */
        ifp = fopen(vendor_file, "r");
        if (ifp == NULL) {
            fprintf(stderr, "Error opening file '%s'\n", vendor_file);
            return(-1);
        }

        /* Iterate over the vendor file */
        while(fgets(buf, MAXLINE, ifp) != NULL) {
            /* zero out the structure */
            memset(&my_vendor, 0, sizeof(VENDOR));
            /* Zero out the DBTs */
            memset(&key, 0, sizeof(DBT));
            memset(&data, 0, sizeof(DBT));

            /*
             * 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(buf,
             "%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);

            /*
             * Now that we have our structure we can load it
             * into the database.
             */

            /* Set up the database record's key */
            key.data = my_vendor.name;
            key.size = (u_int32_t)strlen(my_vendor.name) + 1;

            /* Set up the database record's data */
            data.data = &my_vendor;
            data.size = sizeof(VENDOR);

            /*
             * Note that given the way we built our struct, there are extra
             * bytes in it. Essentially we're using fixed-width fields with
             * the unused portion of some fields padded with zeros. This
             * is the easiest thing to do, but it does result in a bloated
             * database. Look at load_inventory_data() for an example of how
             * to avoid this.
             */
```

```
         /* Put the data into the database.
          * Omitting error handling for clarity.
          */
         my_stock.vendor_dbp->put(my_stock.vendor_dbp, 0,
                                     &key, &data, 0);
    } /* end vendors database while loop */

    /* Close the vendor.txt file */
    fclose(ifp);
    return(0);
}
```

Finally, we need to write the `load_inventory_database()` function. We made this function a bit more complicated than is necessary by avoiding the use of a structure to manage the data. Instead, we manually pack all our inventory data into a single block of memory, and store that data in the database.

While this complicates our code somewhat, this approach allows us to use the smallest amount of space possible for the data that we want to store. 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.

Before we actually implement our inventory loading function, it is useful to create a simple utility function that copies a character array into a buffer at a designated offset:

```
/*
 * Simple little convenience function that takes a buffer, a string,
 * and an offset and copies that string into the buffer at the
 * appropriate location. Used to ensure that all our strings
 * are contained in a single contiguous chunk of memory.
 */
size_t
pack_string(char *buffer, char *string, size_t start_pos)
{
    size_t string_size;
    string_size = strlen(string) + 1;

    memcpy(buffer+start_pos, string, string_size);

    return(start_pos + string_size);
}
```

That done, we can now load the inventory database:

```
/*
 * Loads the contents of the inventory.txt file into
 * a database. Note that because the itemname
```

```
 * secondary database is associated to the inventorydb
 * (see env_setup() in gettingstarted_common.c), the
 * itemname index is automatically created when this
 * database is loaded.
 */
int
load_inventory_database(STOCK_DBS my_stock, char *inventory_file)
{
    DBT key, data;
    char buf[MAXLINE];
    char databuf[MAXDATABUF];
    size_t bufLen, dataLen;
    FILE *ifp;

    /*
     * Rather than lining everything up nicely in a struct, we're being
     * deliberately a bit sloppy here. This function illustrates how to
     * store mixed data that might be obtained from various locations
     * in your application.
     */
    float price;
    int quantity;
    char category[MAXFIELD], name[MAXFIELD];
    char vendor[MAXFIELD], sku[MAXFIELD];

    /* Load the inventory database */
    ifp = fopen(inventory_file, "r");
    if (ifp == NULL) {
        fprintf(stderr, "Error opening file '%s'\n", inventory_file);
        return(-1);
    }

    /*
     * Read the inventory.txt file line by line, saving each line off to
     * the database as we go.
     */
    while(fgets(buf, MAXLINE, ifp) != NULL) {
        /*
         * Scan the line into the appropriate buffers and variables.
         * Convenient, but not particularly safe. In a real
         * program, there would be a lot more defensive code here.
         */
        sscanf(buf,
          "%20[^#]#%20[^#]#%f#%i#%20[^#]#%20[^\n]",
          name, sku, &price, &quantity, category, vendor);

        /*
         * Now pack it into a single contiguous memory location for
         * storage.
```

```
                      */
                     memset(databuf, 0, MAXDATABUF);
                     bufLen = 0;
                     dataLen = 0;

                     /*
                      * We first store the fixed-length elements. This makes our code
                      * to retrieve this data from the database a little bit easier.
                      */

                     /* First discover how long the data element is. */
                     dataLen = sizeof(float);
                     /* Then copy it to our buffer */
                     memcpy(databuf, &price, dataLen);
                     /*
                      * Then figure out how much data is actually in our buffer.
                      * We repeat this pattern for all the data we want to store.
                      */
                     bufLen += dataLen;

                     /* Rinse, lather, repeat. */
                     dataLen = sizeof(int);
                     memcpy(databuf + bufLen, &quantity, dataLen);
                     bufLen += dataLen;

                     bufLen = pack_string(databuf, name, bufLen);
                     bufLen = pack_string(databuf, sku, bufLen);
                     bufLen = pack_string(databuf, category, bufLen);
                     bufLen = pack_string(databuf, vendor, bufLen);

                     /*
                      * Now actually save the contents of the buffer off
                      * to our database.
                      */

                     /* Zero out the DBTs */
                     memset(&key, 0, sizeof(DBT));
                     memset(&data, 0, sizeof(DBT));

                     /*
                      * The key is the item's SKU. This is a unique value, so we need
                      * not support duplicates for this database.
                      */
                     key.data = sku;
                     key.size = (u_int32_t)strlen(sku) + 1;

                     /* The data is the information that we packed into databuf. */
                     data.data = databuf;
                     data.size = (u_int32_t)bufLen;
```

```
          /* Put the data into the database */
          my_stock.vendor_dbp->put(my_stock.inventory_dbp, 0,
                                  &key, &data, 0);
    } /* end vendors database while loop */

    /* Cleanup */
    fclose(ifp);

    return(0);
}
```

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 structure. To use a cursor, you must open it using the DB->cursor() method.

For example:

```
#include <db.h>

...

DB *my_database;
DBC *cursorp;

/* Database open omitted for clarity */

/* Get a cursor */
my_database->cursor(my_database, 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. It is recommended that you close all cursor handles after their use to ensure concurrency and to release resources such as page locks.

```
#include <db.h>

...

DB *my_database;
DBC *cursorp;

/* Database and cursor open omitted for clarity */

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

if (my_database != NULL)
    my_database->close(my_database, 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.h>
#include <string.h>

...

DB *my_database;
DBC *cursorp;
DBT key, data;
int ret;

/* Database open omitted for clarity */

/* Get a cursor */
my_database->cursor(my_database, NULL, &cursorp, 0);

/* Initialize our DBTs. */
memset(&key, 0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));

/* Iterate over the database, retrieving each record in turn. */
while ((ret = cursorp->get(cursorp, &key, &data, DB_NEXT)) == 0) {
        /* Do interesting things with the DBTs here. */
}
if (ret != DB_NOTFOUND) {
        /* Error handling goes here */
}

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

if (my_database != NULL)
    my_database->close(my_database, 0);
```

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

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

...

DB *my_database;
DBC *cursorp;
```

```
DBT key, data;
int ret;

/* Database open omitted for clarity */

/* Get a cursor */
my_database->cursor(my_database, NULL, &cursorp, 0);

/* Initialize our DBTs. */
memset(&key, 0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));

/* Iterate over the database, retrieving each record in turn. */
while ((ret = cursorp->get(cursorp, &key,
        &data, DB_PREV)) == 0) {
        /* Do interesting things with the DBTs here. */
}
if (ret != DB_NOTFOUND) {
        /* Error handling goes here */
}

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

if (my_database != NULL)
    my_database->close(my_database, 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:

Note

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 DBTs 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

a search key of ...	and a search data of ...	moves the cursor to ...
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:

```c
#include <db.h>
#include <string.h>


...

DBC *cursorp;
DBT key, data;
DB *dbp;
int ret;
char *search_data = "Fa";
char *search_key = "Alaska";

/* database open omitted for clarity */

/* Get a cursor */
dbp->cursor(dbp, NULL, &cursorp, 0);

/* Set up our DBTs */
key.data = search_key;
key.size = strlen(search_key) + 1;
data.data = search_data;
data.size = 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.
 */
ret = cursorp->get(cursorp, &key, &data, DB_GET_BOTH_RANGE);
if (!ret) {
    /* Do something with the data */
} else {
    /* Error handling goes here */
}

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

/* Close the database */
if (dbp != NULL)
```

```
    dbp->close(dbp, 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 82).

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 40).

- 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 41) 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.

```c
#include <db.h>
#include <string.h>

...

DB *dbp;
DBC *cursorp;
DBT key, data;
int ret;
char *search_key = "Al";

/* database open omitted for clarity */

/* Get a cursor */
dbp->cursor(dbp, NULL, &cursorp, 0);

/* Set up our DBTs */
key.data = search_key;
key.size = strlen(search_key) + 1;

/*
 * Position the cursor to the first record in the database whose
 * key and data begin with the correct strings.
 */
ret = cursorp->get(cursorp, &key, &data, DB_SET);
while (ret != DB_NOTFOUND) {
    printf("key: %s, data: %s\n", (char *)key.data, (char *)data.data);
    ret = cursorp->get(cursorp, &key, &data, DB_NEXT_DUP);
}

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

/* Close the database */
if (dbp != NULL)
    dbp->close(dbp, 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 82).

- 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.h>
#include <string.h>

...

DB *dbp;
DBC *cursorp;
```

```
DBT data1, data2, data3;
DBT key1, key2;
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";
int ret;

/* Set up our DBTs */
key1.data = key1str;
key1.size = strlen(key1str) + 1;
data1.data = data1str;
data1.size = strlen(data1str) + 1;

key2.data = key2str;
key2.size = strlen(key2str) + 1;
data2.data = data2str;
data2.size = strlen(data2str) + 1;
data3.data = data3str;
data3.size = strlen(data3str) + 1;

/* Database open omitted */

/* Get the cursor */
dbp->cursor(dbp, NULL, &cursorp, 0);

/*
 * Assuming an empty database, this first put places
 * "My first string"/"My first data" in the first
 * position in the database
 */
ret = cursorp->put(cursorp, &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(cursorp, &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(cursorp, &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.
                         */
```

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.h>
#include <string.h>

...

DB *dbp;
DBC *cursorp;
DBT key, data;
char *key1str = "My first string";
int ret;

/* Database open omitted */

/* Initialize our DBTs. */
memset(&key, 0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));

/* Set up our DBTs */
key.data = key1str;
key.size = strlen(key1str) + 1;

/* Get the cursor */
dbp->cursor(dbp, NULL, &cursorp, 0);

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

/* Cursors must be closed */
if (cursorp != NULL)
```

```
        cursorp->close(cursorp);

if (dbp != NULL)
    dbp->close(dbp, 0);
```

Replacing Records Using Cursors

You replace the data for a database record by using DBC->put() with the DB_CURRENT flag.

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

...

DB *dbp;
DBC *cursorp;
DBT key, data;
char *key1str = "My first string";
char *replacement_data = "replace me";
int ret;

/* Initialize our DBTs. */
memset(&key, 0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));

/* Set up our DBTs */
key.data = key1str;
key.size = strlen(key1str) + 1;

/* Database open omitted */

/* Get the cursor */
dbp->cursor(dbp, NULL, &cursorp, 0);

/* Position the cursor */
ret = cursorp->get(cursorp, &key, &data, DB_SET);
if (ret == 0) {
    data.data = replacement_data;
    data.size = strlen(replacement_data) + 1;
    cursorp->put(cursorp, &key, &data, DB_CURRENT);
}

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

if (dbp != NULL)
    dbp->close(dbp, 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 30) 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 31) 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/c/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.

```
/* File: example_database_read.c */
/* gettingstarted_common.h includes db.h for us */
#include "gettingstarted_common.h"
```

```
/* Forward declarations */
char * show_inventory_item(void *);
int show_all_records(STOCK_DBS *);
int show_records(STOCK_DBS *, char *);
int show_vendor_record(char *, DB *);
```

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.

```
/*
 * Searches for an inventory item based on that item's name. The search is
 * performed using the item name secondary database. Displays all
 * inventory items that use the specified name, as well as the vendor
 * associated with that inventory item.
 *
 * If no item name is provided, then all inventory items are displayed.
 */
int
main(int argc, char *argv[])
{
    STOCK_DBS my_stock;
    int ret;

    /* Initialize the STOCK_DBS struct */
    initialize_stockdbs(&my_stock);

    /*
     * Parse the command line arguments here and determine
     * the location of the database files. This step is
     * omitted for brevity.
     */

    /*
     * Identify the files that will hold our databases
     * This function uses information obtained from the
     * command line to identify the directory in which
     * the database files reside.
     */
    set_db_filenames(&my_stock);

    /* Open all databases */
    ret = databases_setup(&my_stock, "example_database_read", stderr);
    if (ret != 0) {
        fprintf(stderr, "Error opening databases\n");
        databases_close(&my_stock);
        return (ret);
    }

    ret = show_all_records(&my_stock);
```

```
    /* close our databases */
    databases_close(&my_stock);
    return (ret);
}
```

Next we need to write the `show_all_records()` function. This function takes a `STOCK_DBS` structure and 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:

```
int show_all_records(STOCK_DBS *my_stock)
{
    DBC *inventory_cursorp;
    DBT key, data;
    char *the_vendor;
    int exit_value, ret;

    /* Initialize our DBTs. */
    memset(&key, 0, sizeof(DBT));
    memset(&data, 0, sizeof(DBT));

    /* Get a cursor to the itemname db */
    my_stock->inventory_dbp->cursor(my_stock->inventory_dbp, NULL,
      &inventory_cursorp, 0);

    /*
     * Iterate over the inventory database, from the first record
     * to the last, displaying each in turn.
     */
    exit_value = 0;
    while ((ret =
      inventory_cursorp->get(inventory_cursorp, &key, &data, DB_NEXT))
      == 0)
    {
        the_vendor = show_inventory_item(data.data);
        ret = show_vendor_record(the_vendor, my_stock->vendor_dbp);
        if (ret) {
            exit_value = ret;
            break;
        }
    }

    /* Close the cursor */
    inventory_cursorp->close(inventory_cursorp);
    return(exit_value);
}
```

The `show_inventory_item()` simply extracts the inventory information from the record data and displays it. It then returns the vendor's name. Note that in order to extract the

inventory information, we have to unpack it from the data buffer. How we do this is entirely dependent on how we packed the buffer in the first place. For more information, see the load_inventory_database() function implementation in example_database_load (page 31).

```
/*
 * Shows an inventory item. How we retrieve the inventory
 * item values from the provided buffer is strictly dependent
 * on the order that those items were originally stored in the
 * DBT. See load_inventory_database in example_database_load
 * for how this was done.
 */
char *
show_inventory_item(void *vBuf)
{
    float price;
    int quantity;
    size_t buf_pos;
    char *category, *name, *sku, *vendor_name;
    char *buf = (char *)vBuf;

    /* Get the price.  */
    price = *((float *)buf);
    buf_pos = sizeof(float);

    /* Get the quantity. */
    quantity = *((int *)(buf + buf_pos));
    buf_pos += sizeof(int);

    /* Get the inventory item's name */
    name = buf + buf_pos;
    buf_pos += strlen(name) + 1;

    /* Get the inventory item's sku */
    sku = buf + buf_pos;
    buf_pos += strlen(sku) + 1;

    /*
     * Get the category (fruits, vegetables, desserts) that this
     * item belongs to.
     */
    category = buf + buf_pos;
    buf_pos += strlen(category) + 1;

    /* Get the vendor's name */
    vendor_name = buf + buf_pos;

    /* Display all this information */
    printf("name: %s\n", name);
    printf("\tSKU: %s\n", sku);
    printf("\tCategory: %s\n", category);
```

```
        printf("\tPrice: %.2f\n", price);
        printf("\tQuantity: %i\n", quantity);
        printf("\tVendor:\n");

        /* Return the vendor's name */
        return(vendor_name);
}
```

Having returned the vendor's name, we can now use it to look up and display the appropriate vendor 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 load_vendor_database() in
 * example_database_load for how this structure was originally
 * put into the database.
 */
int
show_vendor_record(char *vendor_name, DB *vendor_dbp)
{
    DBT key, data;
    VENDOR my_vendor;
    int ret;

    /* Zero our DBTs */
    memset(&key, 0, sizeof(DBT));
    memset(&data, 0, sizeof(DBT));

    /* Set the search key to the vendor's name */
    key.data = vendor_name;
    key.size = (u_int32_t)strlen(vendor_name) + 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.data = &my_vendor;
    data.ulen = sizeof(VENDOR);
    data.flags = DB_DBT_USERMEM;

    /* Get the record */
    ret = vendor_dbp->get(vendor_dbp, NULL, &key, &data, 0);
    if (ret != 0) {
        vendor_dbp->err(vendor_dbp, ret,
            "Error searching for vendor: '%s'", vendor_name);
```

```
            return(ret);
        } else {
            printf("\t\t%s\n", my_vendor.name);
            printf("\t\t%s\n", my_vendor.street);
            printf("\t\t%s, %s\n", my_vendor.city, my_vendor.state);
            printf("\t\t%s\n\n", my_vendor.zipcode);
            printf("\t\t%s\n\n", my_vendor.phone_number);
            printf("\t\tContact: %s\n", my_vendor.sales_rep);
            printf("\t\t%s\n", my_vendor.sales_rep_phone);
        }
        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 62) for more information.

Note

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 58) 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.h>

...

DB *dbp, *sdbp;    /* Primary and secondary DB handles */
u_int32_t flags;   /* Primary database open flags */
int ret;           /* Function return value */

/* Primary */
ret = db_create(&dbp, NULL, 0);
if (ret != 0) {
  /* Error handling goes here */
}

/* Secondary */
ret = db_create(&sdbp, NULL, 0);
if (ret != 0) {
  /* Error handling goes here */
}

/* Usually we want to support duplicates for secondary databases */
ret = sdbp->set_flags(sdbp, DB_DUPSORT);
if (ret != 0) {
  /* Error handling goes here */
}


/* Database open flags */
flags = DB_CREATE;    /* If the database does not exist,
                       * create it.*/

/* open the primary database */
ret = dbp->open(dbp,        /* DB structure pointer */
            NULL,       /* Transaction pointer */
```

```
                "my_db.db", /* On-disk file that holds the database. */
                NULL,       /* Optional logical database name */
                DB_BTREE,   /* Database access method */
                flags,      /* Open flags */
                0);         /* File mode (using defaults) */
if (ret != 0) {
  /* Error handling goes here */
}

/* open the secondary database */
ret = sdbp->open(sdbp,          /* DB structure pointer */
                NULL,           /* Transaction pointer */
                "my_secdb.db", /* On-disk file that holds the
                                  database. */
                NULL,           /* Optional logical database name */
                DB_BTREE,       /* Database access method */
                flags,          /* Open flags */
                0);             /* File mode (using defaults) */
if (ret != 0) {
  /* Error handling goes here */
}

/* Now associate the secondary to the primary */
dbp->associate(dbp,             /* Primary database */
                NULL,           /* TXN id */
                sdbp,           /* Secondary database */
                get_sales_rep,  /* Callback used for key creation. Not
                                 * defined in this example. See 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 */
if (sdbp != NULL)
    sdbp->close(sdbp, 0);
if (dbp != NULL)
    dbp->close(dbp, 0);
```

Implementing Key Extractors

You must provide every secondary database with a callback that creates keys from primary records. You identify this callback 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.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 = pdata->data;

    /* Now set the secondary key's data to be the representative's name */
    memset(skey, 0, sizeof(DBT));
    skey->data = vendor->sales_rep;
    skey->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:

```
dbp->associate(dbp,            /* Primary database */
               NULL,           /* TXN id */
               sdbp,           /* 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:

Note

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].data = tmpdata1;
```

```
        tmpdbt[0].size = (u_int32_t)strlen(tmpdbt[0].data) + 1;
        tmpdbt[1].data = tmpdata2;
        tmpdbt[1].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->flags = DB_DBT_MULTIPLE | DB_DBT_APPMALLOC;

    /* Point the results data field to the arrays of DBTs */
    skey->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.h>
#include <string.h>

...

DB *my_secondary_database;
DBT key; /* Used for the search key */
DBT pkey, pdata; /* Used to return the primary key and data */
char *search_name = "John Doe";

/* Primary and secondary database opens omitted for brevity */

/* Zero out the DBTs before using them. */
memset(&key, 0, sizeof(DBT));
memset(&pkey, 0, sizeof(DBT));
memset(&pdata, 0, sizeof(DBT));
```

```
key.data = search_name;
key.size = strlen(search_name) + 1;

/* Returns the key from the secondary database, and the data from the
 * associated primary database entry.
 */
my_secondary_database->get(my_secondary_database, 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(my_secondary_database, 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 39).

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.

Note

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.h>
#include <string.h>

...

DB *dbp, *sdbp;    /* Primary and secondary DB handles */
DBT key;           /* DBTs used for the delete */
int ret;           /* Function return value */
char *search_name = "John Doe"; /* Name to delete */
```

```
/* Primary */
ret = db_create(&dbp, NULL, 0);
if (ret != 0) {
  /* Error handling goes here */
}

/* Secondary */
ret = db_create(&sdbp, NULL, 0);
if (ret != 0) {
  /* Error handling goes here */
}

/* Usually we want to support duplicates for secondary databases */
ret = sdbp->set_flags(sdbp, DB_DUPSORT);
if (ret != 0) {
  /* Error handling goes here */
}

/* open the primary database */
ret = dbp->open(dbp,          /* DB structure pointer */
                NULL,         /* Transaction pointer */
                "my_db.db",   /* On-disk file that holds the database.
                               * Required. */
                NULL,         /* Optional logical database name */
                DB_BTREE,     /* Database access method */
                0,            /* Open flags */
                0);           /* File mode (using defaults) */
if (ret != 0) {
  /* Error handling goes here */
}

/* open the secondary database */
ret = sdbp->open(sdbp,          /* DB structure pointer */
                 NULL,          /* Transaction pointer */
                 "my_secdb.db", /* On-disk file that holds the database.
                                 * Required. */
                 NULL,          /* Optional logical database name */
                 DB_BTREE,      /* Database access method */
                 0,             /* Open flags */
                 0);            /* File mode (using defaults) */
if (ret != 0) {
  /* Error handling goes here */
}

/* Now associate the secondary to the primary */
dbp->associate(dbp,                /* Primary database */
               NULL,               /* TXN id */
               sdbp,               /* Secondary database */
               get_sales_rep,      /* Callback used for key creation. */
```

```
                   0);                    /* Flags */

/*
 * Zero out the DBT before using it.
 */
memset(&key, 0, sizeof(DBT));

key.data = search_name;
key.size = 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.
 */
sdbp->del(sdbp, 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 39).

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 58). 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.h>
#include <string.h>

...

DB *sdbp;          /* Secondary DB handle */
DBC *cursorp;      /* Cursor */
DBT key, data;     /* DBTs used for the delete */
char *search_name = "John Doe"; /* Name to delete */

/* Primary and secondary database opens omitted for brevity. */
```

```
/* Get a cursor on the secondary database */
sdbp->cursor(sdbp, NULL, &cursorp, 0);

/*
 * Zero out the DBT before using it.
 */
memset(&key, 0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));

key.data = search_name;
key.size = strlen(search_name) + 1;


/* Position the cursor */
while (cursorp->get(cursorp, &key, &data, DB_SET) == 0)
    cursorp->del(cursorp, 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.h>
#include <string.h>

...

DB *automotiveDB;
DB *automotiveColorDB;
DB *automotiveMakeDB;
DB *automotiveTypeDB;
DBC *color_curs, *make_curs, *type_curs, *join_curs;
DBC *carray[4];
DBT key, data;
int ret;

char *the_color = "red";
char *the_type = "minivan";
char *the_make = "Toyota";

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

```
/* initialize pointers and structures */
color_curs = NULL;
make_curs = NULL;
type_curs = NULL;
join_curs = NULL;

memset(&key, 0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));

/* open the cursors */
if (( ret =
    automotiveColorDB->cursor(automotiveColorDB, NULL,
      &color_curs, 0)) != 0) {
        /* Error handling goes here */
}

if (( ret =
    automotiveMakeDB->cursor(automotiveMakeDB, NULL,
      &make_curs, 0)) != 0) {
        /* Error handling goes here */
}

if (( ret =
    automotiveTypeDB->cursor(automotiveTypeDB, NULL,
      &type_curs, 0)) != 0) {
        /* Error handling goes here */
}

/* Position the cursors */
key.data = the_color;
key.size = strlen(the_color) + 1;
if ((ret = color_curs->get(color_curs, &key, &data, DB_SET)) != 0)
    /* Error handling goes here */

key.data = the_make;
key.size = strlen(the_make) + 1;
if ((ret = make_curs->get(make_curs, &key, &data, DB_SET)) != 0)
    /* Error handling goes here */

key.data = the_type;
key.size = strlen(the_type) + 1;
if ((ret = type_curs->get(type_curs, &key, &data, DB_SET)) != 0)
    /* Error handling goes here */

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

```
    carray[3] = NULL;

    /* Create the join */
    if ((ret = automotiveDB->join(automotiveDB, carray, &join_curs, 0)) != 0)
        /* Error handling goes here */

    /* Iterate using the join cursor */
    while ((ret = join_curs->get(join_curs, &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 30) we built an application that can open and load data into several databases. In Secondary Databases with example_database_load (page 68) we will extend that application to also open a secondary database for the purpose of indexing inventory item names.

- In Cursor Example (page 50) we built an application to display our inventory database (and related vendor information). In Secondary Databases with example_database_read (page 75) 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

example_database_load uses several utility functions to open and close its databases. In order to cause example_database_load to maintain an index of inventory item names, all we really need to do is update the utility functions to:

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

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

3.   Close the secondary database when we close the rest of our databases.

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. Therefore, we can limit all our work to the code found in gettingstarted_common.h and gettingstarted_common.c.

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

```
DB_INSTALL/examples/c/getting_started
```

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

To begin, we need to update the stock_dbs structure to accommodate the additional database. We defined this structure in gettingstarted_common.h. We can limit our update to this file to just that structure definition:

Remember that new code is in **bold**.

```
/* file: gettingstarted_common.h */
#include <db.h>

typedef struct stock_dbs {
    DB *inventory_dbp; /* Database containing inventory information */
    DB *vendor_dbp;    /* Database containing vendor information */
    DB *itemname_sdbp; /* Index based on the item name index */

    char *db_home_dir;      /* Directory containing the database files */
    char *itemname_db_name;  /* Itemname secondary database */
    char *inventory_db_name; /* Name of the inventory database */
    char *vendor_db_name;    /* Name of the vendor database */
} STOCK_DBS;

/* Function prototypes */
int databases_setup(STOCK_DBS *, const char *, FILE *);
int databases_close(STOCK_DBS *);
void initialize_stockdbs(STOCK_DBS *);
int open_database(DB **, const char *, const char *, FILE *, int);
void set_db_filenames(STOCK_DBS *my_stock);
```

Because we updated our stock_dbs structure, we need to update our stock_dbs utility functions (The stock_db Utility Functions (page 17)) accordingly. The updates are trivial and so we won't show them here in the interest of space. You can find their complete implementation in the gettingstarted_common.c file accompanying this example in your DB distribution.

More importantly, however, we need to go to gettingstarted_common.c and create our secondary key extractor function. When we store our inventory items, we place the item name in the buffer immediately after a float and an int, so retrieving the string from the buffer is fairly easy to do:

```
/* file: gettingstarted_common.c */
#include "gettingstarted_common.h"

/*
 * 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)
{
    int offset;

    /*
     * First, obtain the buffer location where 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 float) and the quantity (an int).
     *
     * See load_inventory_database() in example_database_load.c
     * for how we marshalled the inventory information into the
     * data DBT.
     */
    offset = sizeof(float) + sizeof(int);

    /* Check to make sure there's data */
    if (pdata->size < offset)
        return (-1); /* Returning non-zero means that the
                      * secondary record is not created/updated.
                      */

    /* Now set the secondary key's data to be the item name */
    memset(skey, 0, sizeof(DBT));
    skey->data = (u_int8_t *)pdata->data + offset;
    skey->size = (u_int32_t)strlen(skey->data) + 1;

    return (0);
}
```

Having completed that function, we need to update set_db_filenames() and
initialize_stockdbs() to handle the new secondary databases that our application will
now use. These functions were originally introduced in The stock_db Utility Functions (page
17).

```
/* Initializes the STOCK_DBS struct.*/
void
initialize_stockdbs(STOCK_DBS *my_stock)
{
    my_stock->db_home_dir = DEFAULT_HOMEDIR;
    my_stock->inventory_dbp = NULL;
    my_stock->vendor_dbp = NULL;
    my_stock->itemname_sdbp = NULL;

    my_stock->inventory_db_name = NULL;
```

```
        my_stock->vendor_db_name = NULL;
        my_stock->itemname_db_name = NULL;
}

/* Identify all the files that will hold our databases. */
void
set_db_filenames(STOCK_DBS *my_stock)
{
    size_t size;

    /* Create the Inventory DB file name */
    size = strlen(my_stock->db_home_dir) + strlen(INVENTORYDB) + 1;
    my_stock->inventory_db_name = malloc(size);
    snprintf(my_stock->inventory_db_name, size, "%s%s",
      my_stock->db_home_dir, INVENTORYDB);

    /* Create the Vendor DB file name */
    size = strlen(my_stock->db_home_dir) + strlen(VENDORDB) + 1;
    my_stock->vendor_db_name = malloc(size);
    snprintf(my_stock->vendor_db_name, size, "%s%s",
      my_stock->db_home_dir, VENDORDB);

    /* Create the itemname DB file name */
    size = strlen(my_stock->db_home_dir) + strlen(ITEMNAMEDB) + 1;
    my_stock->itemname_db_name = malloc(size);
    snprintf(my_stock->itemname_db_name, size, "%s%s",
      my_stock->db_home_dir, ITEMNAMEDB);
}
```

We also need to update the open_database() (as described in open_database()
Function (page 18)) to take special actions if we are opening a secondary database. Unlike
our primary databases, we want to support sorted duplicates for our secondary database. This
is because we are indexing based on an item's name, and item names are shared by multiple
inventory records. As a result every key the secondary database (an item name) will be used
by multiple records (pointers to records in our primary database). We allow this by configuring
our secondary database to support duplicate records. Further, because BTrees perform best
when their records are sorted, we go ahead and configure our secondary database for sorted
duplicates.

To do this, we add a parameter to the function that indicates whether we are opening
a secondary database, and we add in the few lines of code necessary to set the sorted
duplicates flags.

```
/* Opens a database */
int
open_database(DB **dbpp,          /* The DB handle that we are opening */
    const char *file_name,     /* The file in which the db lives */
    const char *program_name,  /* Name of the program. */
    FILE *error_file_pointer,
    int is_secondary)
```

```
{
    DB *dbp;    /* For convenience */
    u_int32_t open_flags;
    int ret;

    /* Initialize the DB handle */
    ret = db_create(&dbp, NULL, 0);
    if (ret != 0) {
        fprintf(error_file_pointer, "%s: %s\n", program_name,
                db_strerror(ret));
        return (ret);
    }

    /* Point to the memory malloc'd by db_create() */
    *dbpp = dbp;

    /* Set up error handling for this database */
    dbp->set_errfile(dbp, error_file_pointer);
    dbp->set_errpfx(dbp, program_name);

    /*
     * If this is a secondary database, then we want to allow
     * sorted duplicates.
     */
    if (is_secondary) {
        ret = dbp->set_flags(dbp, DB_DUPSORT);
        if (ret != 0) {
            dbp->err(dbp, ret, "Attempt to set DUPSORT flag failed.",
              file_name);
            return (ret);
        }
    }

    /* Set the open flags */
    open_flags = DB_CREATE;

    /* Now open the database */
    ret = dbp->open(dbp,        /* Pointer to the database */
                    NULL,       /* Txn pointer */
                    file_name,  /* File name */
                    NULL,       /* Logical db name (unneeded) */
                    DB_BTREE,   /* Database type (using btree) */
                    open_flags, /* Open flags */
                    0);         /* File mode. Using defaults */
    if (ret != 0) {
        dbp->err(dbp, ret, "Database '%s' open failed.", file_name);
        return (ret);
    }
```

```
        return (ret);
 }
```

That done, we can now update databases_setup() (see The databases_setup() Function (page 19)) to create and open our secondary database. To do this, we have to add a flag to each call to open_database() that indicates whether the database is a secondary. We also have to associate our secondary database with the inventory database (the primary).

Note that we do not anywhere in this example show the definition of PRIMARY_DB and SECONDARY_DB. See gettingstarted_common.h in your DB examples directory for those definitions (they are just 0 and 1, respectively).

```
/* opens all databases */
int
databases_setup(STOCK_DBS *my_stock, const char *program_name,
  FILE *error_file_pointer)
{
    int ret;

    /* Open the vendor database */
    ret = open_database(&(my_stock->vendor_dbp),
      my_stock->vendor_db_name,
      program_name, error_file_pointer,
      PRIMARY_DB);
    if (ret != 0)
        /*
         * Error reporting is handled in open_database() so just return
         * the return code here.
         */
        return (ret);

    /* Open the inventory database */
    ret = open_database(&(my_stock->inventory_dbp),
      my_stock->inventory_db_name,
      program_name, error_file_pointer,
      PRIMARY_DB);
    if (ret != 0)
        /*
         * Error reporting is handled in open_database() so just return
         * the return code here.
         */
        return (ret);

    /*
     * Open the itemname secondary database. This is used to
     * index the product names found in the inventory
     * database.
     */
    ret = open_database(&(my_stock->itemname_sdbp),
      my_stock->itemname_db_name,
```

```
        program_name, error_file_pointer,
        SECONDARY_DB);
    if (ret != 0)
        /*
         * Error reporting is handled in open_database() so just return
         * the return code here.
         */
        return (ret);

    /*
     * Associate the itemname db with its primary db
     * (inventory db).
     */
    my_stock->inventory_dbp->associate(
        my_stock->inventory_dbp,     /* Primary db */
        NULL,                        /* txn id */
        my_stock->itemname_sdbp,     /* Secondary db */
        get_item_name,               /* Secondary key extractor */
        0);                          /* Flags */


    printf("databases opened successfully\n");
    return (0);
}
```

Finally, we need to update databases_close() (The databases_close() Function (page 20)) to close our new secondary database. Note that we are careful to close the secondary before the primary, even though the database close routine is single threaded.

```
/* Closes all the databases and secondary databases. */
int
databases_close(STOCK_DBS *my_stock)
{
    int ret;
    /*
     * Note that closing a database automatically flushes its cached data
     * to disk, so no sync is required here.
     */

    if (my_stock->itemname_sdbp != NULL) {
        ret = my_stock->itemname_sdbp->close(my_stock->itemname_sdbp, 0);
        if (ret != 0)
            fprintf(stderr, "Itemname database close failed: %s\n",
              db_strerror(ret));
    }

    if (my_stock->inventory_dbp != NULL) {
        ret = my_stock->inventory_dbp->close(my_stock->inventory_dbp, 0);
        if (ret != 0)
            fprintf(stderr, "Inventory database close failed: %s\n",
```

```
                db_strerror(ret));
    }

    if (my_stock->vendor_dbp != NULL) {
        ret = my_stock->vendor_dbp->close(my_stock->vendor_dbp, 0);
        if (ret != 0)
            fprintf(stderr, "Vendor database close failed: %s\n",
                db_strerror(ret));
    }

    printf("databases closed.\n");
    return (0);
}
```

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:

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 50) 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.

Because we manage all our database open and close activities in `databases_setup()` and `databases_close()`, the update to `example_database_read` is relatively modest. We need only 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.c */
/* gettingstarted_common.h includes db.h for us */
#include "gettingstarted_common.h"

/* Forward declarations */
char * show_inventory_item(void *);
int show_all_records(STOCK_DBS *);
int show_records(STOCK_DBS *, char *);
int show_vendor_record(char *, DB *);
```

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

```
/*
 * Searches for a inventory item based on that item's name. The search is
```

```
 * performed using the item name secondary database. Displays all
 * inventory items that use the specified name, as well as the vendor
 * associated with that inventory item.
 *
 * If no item name is provided, then all inventory items are displayed.
 */
int
main(int argc, char *argv[])
{
    STOCK_DBS my_stock;
    int ret;
    char *itemname;

    /* Initialize the STOCK_DBS struct */
    initialize_stockdbs(&my_stock);

    itemname = NULL;
    /*
     * Parse the command line arguments here and determine
     * the location of the database files as well as the
     * inventory item we want displayed, if any. This step is
     * omitted for brevity.
     */

    /*
     * Identify the files that will hold our databases
     * This function uses information obtained from the
     * command line to identify the directory in which
     * the database files reside.
     */
    set_db_filenames(&my_stock);

    /* Open all databases */
    ret = databases_setup(&my_stock, "example_database_read", stderr);
    if (ret != 0) {
        fprintf(stderr, "Error opening databases\n");
        databases_close(&my_stock);
        return (ret);
    }
```

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.

```
    /*
     * Show either a single item or all items, depending
     * on whether itemname is set to a value.
     */
    if (itemname == NULL)
        ret = show_all_records(&my_stock);
```

```
    else
        ret = show_records(&my_stock, itemname);

    /* Close our databases */
    databases_close(&my_stock);
    return (ret);
}
```

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

Note

In the interest of space, we refrain from showing the other functions used by
this application. For their implementation, please see Cursor Example (page 50).
Alternatively, you can see the entire implementation of this application in:

```
    DB_INSTALL/examples/c/getting_started
```

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

```
/*
 * Search for an inventory item given its name (using the inventory item
 * secondary database) and display that record and any duplicates that may
 * exist.
 */
int
show_records(STOCK_DBS *my_stock, char *itemname)
{
    DBC *itemname_cursorp;
    DBT key, data;
    char *the_vendor;
    int ret, exit_value;

    /* Initialize our DBTs. */
    memset(&key, 0, sizeof(DBT));
    memset(&data, 0, sizeof(DBT));

    /* Get a cursor to the itemname db */
    my_stock->itemname_sdbp->cursor(my_stock->itemname_sdbp, 0,
      &itemname_cursorp, 0);

    /*
     * Get the search key. This is the name on the inventory
     * record that we want to examine.
     */
    key.data = itemname;
    key.size = strlen(itemname) + 1;

    /*
     * Position our cursor to the first record in the secondary
```

```
      * database that has the appropriate key.
      */
     exit_value = 0;
     ret = itemname_cursorp->get(itemname_cursorp, &key, &data, DB_SET);
     if (!ret) {
         do {
             /*
              * Show the inventory record and the vendor responsible
              * for this inventory item.
              */
             the_vendor = show_inventory_item(data.data);
             ret = show_vendor_record(the_vendor, my_stock->vendor_dbp);
             if (ret) {
                 exit_value = ret;
                 break;
             }
             /*
              * Our secondary allows duplicates, so we need to loop over
              * the next duplicate records and show them all. This is done
              * because an inventory item's name is not a unique value.
              */
         } while(itemname_cursorp->get(itemname_cursorp, &key, &data,
             DB_NEXT_DUP) == 0);
     } else {
         printf("No records found for '%s'\n", itemname);
     }

     /* Close the cursor */
     itemname_cursorp->close(itemname_cursorp);

     return (exit_value);
}
```

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 `DB_ENV->lock_stat()` 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.

Note

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://docs.oracle.com/cd/E17076_02/html/programmer_reference/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 `DB_ENV->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. Note that this flag also sets the `DB_DUP` flag for you.

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

```
#include <db.h>
...

DB *dbp;
FILE *error_file_pointer;
int ret;
char *program_name = "my_prog";
char *file_name = "mydb.db";

/* Variable assignments omitted for brevity */

/* Initialize the DB handle */
ret = db_create(&dbp, NULL, 0);
if (ret != 0) {
    fprintf(error_file_pointer, "%s: %s\n", program_name,
```

```
        db_strerror(ret));
    return(ret);
}

/* Set up error handling for this database */
dbp->set_errfile(dbp, error_file_pointer);
dbp->set_errpfx(dbp, program_name);

/*
 * Configure the database for sorted duplicates
 */
ret = dbp->set_flags(dbp, DB_DUPSORT);
if (ret != 0) {
    dbp->err(dbp, ret, "Attempt to set DUPSORT flag failed.");
    dbp->close(dbp, 0);
    return(ret);
}

/* Now open the database */
ret = dbp->open(dbp,         /* Pointer to the database */
                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 */
if (ret != 0) {
    dbp->err(dbp, ret, "Database '%s' open failed.", file_name);
    dbp->close(dbp, 0);
    return(ret);
}
```

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://docs.oracle.com/cd/E17076_02/html/programmer_reference/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, size_t *locp)
```

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, size_t *locp)
{
    int ai, bi;

    locp = NULL;
    /*
     * Returns:
     * < 0 if a < b
     * = 0 if a = b
     * > 0 if a > b
     */
    memcpy(&ai, a->data, sizeof(int));
    memcpy(&bi, b->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.h>
#include <string.h>

...

DB *dbp;
int ret;

/* Create a database */
ret = db_create(&dbp, NULL, 0);
if (ret != 0) {
        fprintf(stderr, "%s: %s\n", "my_program",
          db_strerror(ret));
        return(-1);
}

/* Set up the btree comparison function for this database */
dbp->set_bt_compare(dbp, compare_int);

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