9

Berkeley DB Transactional Data Store Applications

It is difficult to write a useful transactional tutorial and still keep within reasonable bounds of documentation; that is, without writing a book on transactional programming. We have two goals in this section: to familiarize readers with the transactional interfaces of Berkeley DB and to provide code building blocks that will be useful for creating applications.

We have not attempted to present this information using a real-world application. First, transactional applications are often complex and time-consuming to explain. Also, one of our goals is to give you an understanding of the wide variety of tools Berkeley DB makes available to you, and no single application would use most of the interfaces included in the Berkeley DB library. For these reasons, we have chosen to simply present the Berkeley DB data structures and programming solutions, using examples that differ from page to page. All the examples are included in a standalone program you can examine, modify, and run; and from which you will be able to extract code blocks for your own applications. Fragments of the program will be presented throughout this chapter, and the complete text of the example program for IEEE/ANSI Std 1003.1 (POSIX) standard systems is included in the Berkeley DB distribution.

Why Transactions?

Perhaps the first question to answer is "Why transactions?" There are a number of reasons to include transactional support in your applications. The most common ones are the following:

- **Recoverability** Applications often need to ensure that no matter how the system or application fails, previously saved data is available the next time the application runs.
- **Deadlock avoidance** When multiple threads of control change the database at the same time, there is usually the possibility of deadlock; that is, each of the threads of control owns a resource that another thread wants, so no thread is able to make forward progress—all waiting for a resource. Deadlocks are resolved by having one of the operations involved release the resources it controls so the other operations can proceed. (The operation releasing its resources usually just tries again later.) Transactions are necessary so that any changes that were already made to the database can be undone as part of releasing the held resources.
- **Atomicity** Applications often need to make multiple changes to one or more databases, but want to ensure that either all of the changes happen, or none of them happens. Transactions guarantee that a group of changes are atomic; that is, if the application or system fails, either all the changes to the databases will appear when the application next runs, or none of them will appear.
- **Repeatable reads** Applications sometimes need to ensure that while doing a group of operations on a database, the value returned as a result of a database retrieval doesn't change; that is, if you retrieve the same key more than once, the data item will be the same each time. Transactions guarantee this behavior.

Terminology

The following are some definitions that will be helpful for understanding transactions:

- **Thread of control** Berkeley DB is indifferent to the type or style of threads being used by the application; or, for that matter, if threads are being used at all—because Berkeley DB supports multiprocess access. In the Berkeley DB documentation, any time we refer to a *thread of control*, it can be read as a true thread (one of many in an application's address space) or a process.
- **Free-threaded** A Berkeley DB handle that can be used by multiple threads simultaneously without any application-level synchronization is called *free-threaded*.
- **Transaction** A *transaction* is one or more operations on one or more databases that should be treated as a single unit of work. For example, changes to a set of databases, in which either all of the changes must be applied to the database(s) or none of them should. Applications specify when each transaction starts, what database operations are included in it, and when it ends.

- **Transaction abort/commit** Every transaction ends by *committing* or *aborting*. If a transaction commits, Berkeley DB guarantees that any database changes included in the transaction will never be lost, even after system or application failure. If a transaction aborts, or is uncommitted when the system or application fails, then the changes involved will never appear in the database.
- **System or application failure** *System or application failure* is the phrase that we use to describe something bad happening near your data. It can be an application dumping core, being interrupted by a signal, the disk filling up, or the entire system crashing. In any case, for whatever reason, the application can no longer make forward progress, and its databases are left in an unknown state.
- **Recovery** Whenever system or application failure occurs, the application must run recovery. *Recovery* is what makes the database consistent; that is, the recovery process includes a review of log files and databases to ensure that the changes from each committed transaction appear in the database, and that no changes from an unfinished (or aborted) transaction do.
- **Deadlock** In its simplest form, deadlock happens when one thread of control owns resource A, but needs resource B; whereas another thread of control owns resource B, but needs resource A. Neither thread of control can make progress, and so one has to give up and release all its resources, at which time the remaining thread of control can make forward progress.

Application Structure

When building transactionally protected applications, there are some special issues that must be considered. The most important one is that if any thread of control exits for any reason while holding Berkeley DB resources, recovery must be performed to do the following:

- Recover the Berkeley DB resources.
- Release any locks or mutexes that may have been held to avoid starvation as the remaining threads of control convoy behind the failed thread's locks.
- Clean up any partially completed operations that may have left a database in an inconsistent or corrupted state.

Complicating this problem is the fact that the Berkeley DB library itself cannot determine whether recovery is required; the application itself must make that decision. A further complication is that recovery *must* be single-threaded; that is, one thread of control or process must perform recovery before any other thread of control or processes attempts to create or join the Berkeley DB environment.

There are two approaches to handling this problem. The first is the hard way. An application can track its own state carefully enough that it knows when recovery needs to be performed. Specifically, the rule to use is that recovery must be performed

before using a Berkeley DB environment any time the threads of control previously using the Berkeley DB environment did not shut the environment down cleanly before exiting the environment for any reason (including application or system failure).

Requirements for shutting down the environment cleanly differ, depending on the type of environment created. If the environment is public and persistent (that is, the DB_PRIVATE flag was not specified to the **DBENV→open** function), recovery must be performed if any transaction was not committed or aborted, or **DBENV→close** function was not called for any open DB_ENV handle.

If the environment is private and temporary (that is, the DB_PRIVATE flag was specified to the **DBENV→open** function), recovery must be performed if any transaction was not committed or aborted, or **DBENV→close** function was not called for any open DB_ENV handle. In addition, at least one transaction checkpoint must be performed after all existing transactions have been committed or aborted.

The second method is the easy way. It greatly simplifies matters that recovery may be performed regardless of whether recovery strictly needs to be performed; that is, it is not an error to run recovery on a database in which no recovery is necessary. Because of this fact, it is almost invariably simpler to ignore the previous rules about shutting an application down cleanly, and simply run recovery each time a thread of control accessing a database environment fails for any reason, as well as before accessing any database environment after system reboot.

There are two common ways to build transactionally protected Berkeley DB applications. The most common way is as a single, usually multithreaded, process. This architecture is simplest because it requires no monitoring of other threads of control. When the application starts, it opens and potentially creates the environment, runs recovery (whether it was needed or not), and then opens its databases. From then on, the application can create new threads of control as it chooses. All threads of control share the open Berkeley DB DB_ENV and DB handles. In this model, databases are rarely opened or closed when more than a single thread of control is running; that is, they are opened when only a single thread is running, and closed after all threads but one have exited. The last thread of control to exit closes the databases and the environment.

An alternative way to build Berkeley DB applications is as a set of cooperating processes, which may or may not be multithreaded. This architecture is more complicated.

First, this architecture requires that the order in which threads of control are created and subsequently access the Berkeley DB environment be controlled because recovery must be single-threaded. The first thread of control to access the environment must run recovery, and no other thread should attempt to access the environment until recovery is complete. (Note that this ordering requirement does not apply to environment creation without recovery. If multiple threads attempt to create a Berkeley DB environment, only one will perform the creation and the others will join the already existing environment.)

Second, this architecture requires that threads of control be monitored. If any thread of control that owns Berkeley DB resources exits without first cleanly discarding those resources, recovery is usually necessary. Before running recovery, all threads using the Berkeley DB environment must relinquish all of their Berkeley DB resources (it does not matter if they do so gracefully or because they are forced to exit). Then, recovery can be run and the threads of control continued or restarted.

We have found that the safest way to structure groups of cooperating processes is to first create a single process (often a shell script) that opens/creates the Berkeley DB environment and runs recovery, and that then creates the processes or threads that will actually perform work. The initial thread has no further responsibilities other than to monitor the threads of control it has created, to ensure that none of them unexpectedly exits. If one exits, the initial process then forces all of the threads of control using the Berkeley DB environment to exit, runs recovery, and restarts the working threads of control.

If it is not practical to have a single parent for the processes sharing a Berkeley DB environment, each process sharing the environment should log its connection to and exit from the environment in some fashion that permits a monitoring process to detect whether a thread of control may have potentially acquired Berkeley DB resources and never released them.

Obviously, it is important that the monitoring process in either case be as simple and well-tested as possible because there is no recourse if it fails.

Opening the Environment

Creating transaction-protected applications using the Berkeley DB library is quite easy. Applications first use **DBENV→open** to initialize the database environment. Transaction-protected applications normally require all four Berkeley DB subsystems, so the DB_INIT_MPOOL, DB_INIT_LOCK, DB_INIT_LOG, and DB_INIT_TXN flags should be specified.

Once the application has called **DBENV→open**, it opens its databases within the environment. Once the databases are opened, the application makes changes to the databases inside of transactions. Each set of changes that entails a unit of work should be surrounded by the appropriate **txn_begin**, **txn_commit**, and **txn_abort** calls. The Berkeley DB access methods will make the appropriate calls into the Locking, Logging and Memory Pool subsystems in order to guarantee transaction semantics. When the application is ready to exit, all outstanding transactions should have been committed or aborted.

Databases accessed by a transaction must not be closed during the transaction. Once all outstanding transactions are finished, all open Berkeley DB files should be closed. When the Berkeley DB database files have been closed, the environment should be closed by calling **DBENV→close**.

The following code fragment creates the database environment directory and then opens the environment, running recovery. Our DB_ENV database environment handle is declared to be free-threaded using the DB_THREAD flag, and so may be used by any number of threads that we may subsequently create.

```
#include <sys/types.h>
#include <sys/stat.h>

#include <errno.h>
#include <pthread.h>
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <db.h>
#define ENV_DIRECTORY "TXNAPP"

void env_dir_create(void);
void env_open(DB_ENV **);

int
main(int argc, char *argv)
{
        extern char *optarg;
        extern int optind;
        DB *db_cats, *db_color, *db_fruit;
        DB_ENV *dbenv;
        pthread_t ptid;
        int ch;

        while ((ch = getopt(argc, argv, "")) != EOF)
                switch (ch) {
                case '?':
                default:
                        usage();
                }
        argc -= optind;
        argv += optind;

        env_dir_create();
        env_open(&dbenv);

        return (0);
}

void
env_dir_create()
{
       struct stat sb;
```

```
        /*
         * If the directory exists, we're done. We do not further check
         * the type of the file, DB will fail appropriately if it's the
         * wrong type.
         */
        if (stat(ENV_DIRECTORY, &sb) == 0)
                return;

        /* Create the directory, read/write/access owner only. */
        if (mkdir(ENV_DIRECTORY, S_IRWXU) != 0) {
                fprintf(stderr,
                    "txnapp: mkdir: %s: %s\n", ENV_DIRECTORY, strerror(errno));
                exit (1);
         }
}

void
env_open(DB_ENV **dbenvp)
{
        DB_ENV *dbenv;
        int ret;

        /* Create the environment handle. */
        if ((ret = db_env_create(&dbenv, 0)) != 0) {
                fprintf(stderr,
                    "txnapp: db_env_create: %s\n", db_strerror(ret));
                exit (1);
        }
        /* Set up error handling. */
        dbenv->set_errpfx(dbenv, "txnapp");

        /*
         * Open a transactional environment:
         * create if it doesn't exist
         * free-threaded handle
         * run recovery
         * read/write owner only
         */
        if ((ret = dbenv->open(dbenv, ENV_DIRECTORY,
            DB_CREATE | DB_INIT_LOCK | DB_INIT_LOG |
            DB_INIT_MPOOL | DB_INIT_TXN | DB_RECOVER | DB_THREAD,
            S_IRUSR | S_IWUSR)) != 0) {
               dbenv->err(dbenv, ret, "dbenv->open: %s", ENV_DIRECTORY);
               exit (1);
        }
        *dbenvp = dbenv;
}
```

After running this initial program, we can use the **db_stat** utility to display the contents of the environment directory:

```
prompt> db_stat -e -h TXNAPP
3.2.1   Environment version.
120897  Magic number.
```

```
0       Panic value.
1       References.
6       Locks granted without waiting.
0       Locks granted after waiting.
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Mpool Region: 4.
264KB   Size (270336 bytes).
-1      Segment ID.
1       Locks granted without waiting.
0       Locks granted after waiting.
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Log Region: 3.
96KB    Size (98304 bytes).
-1      Segment ID.
3       Locks granted without waiting.
0       Locks granted after waiting.
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Lock Region: 2.
240KB   Size (245760 bytes).
-1      Segment ID.
1       Locks granted without waiting.
0       Locks granted after waiting.
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Txn Region: 5.
8KB     Size (8192 bytes).
-1      Segment ID.
1       Locks granted without waiting.
0       Locks granted after waiting.
```

Opening the Databases

Next, we open three databases ("color," "fruit," and "cats") in the database environment. Again, our DB database handles are declared to be free-threaded using the DB_THREAD flag, and so may be used by any number of threads we subsequently create.

```
int
main(int argc, char *argv)
{
        extern char *optarg;
        extern int optind;
        DB *db_cats, *db_color, *db_fruit;
        DB_ENV *dbenv;
        pthread_t ptid;
        int ch;

        while ((ch = getopt(argc, argv, "")) != EOF)
                switch (ch) {
                case '?':
                default:
                        usage();
                }
        argc -= optind;
        argv += optind;
```

```
        env_dir_create();
        env_open(&dbenv);

        /* Open database: Key is fruit class; Data is specific type. */
        db_open(dbenv, &db_fruit, "fruit", 0);

        /* Open database: Key is a color; Data is an integer. */
        db_open(dbenv, &db_color, "color", 0);

        /*
         * Open database:
         *      Key is a name; Data is: company name, address, cat breeds.
         */
         db_open(dbenv, &db_cats, "cats", 1);

         return (0);
}

void
db_open(DB_ENV *dbenv, DB **dbp, char *name, int dups)
{
         DB *db;
         int ret;

         /* Create the database handle. */
         if ((ret = db_create(&db, dbenv, 0)) != 0) {
                 dbenv→err(dbenv, ret, "db_create");
                 exit (1);
        }

        /* Optionally, turn on duplicate data items. */
        if (dups && (ret = db→set_flags(db,DB_DUP)) != 0) {
                dbenv→err(dbenv, ret, "db→set_flags: DB_DUP");
                exit (1);
        }

        /*
         * Open a database in the environment:
         *      create if it doesn't exist
         *      free-threaded handle
         *      read/write owner only
         */
        if ((ret = db→open(db, name, NULL,
            DB_BTREE, DB_CREATE ¦ DB_THREAD, S_IRUSR ¦ S_IWUSR)) != 0) {
                dbenv→err(dbenv, ret, "db→open: %s", name);
                exit (1);
        }

        *dbp = db;
}
```

There is no reason to wrap database opens inside of transactions. All database opens are transaction-protected internally to Berkeley DB, and applications using transaction-protected environments can simply rely on files either being successfully re-created in a recovered environment or not appearing at all.

After running this initial code, we can use the **db_stat** utility to display information about a database we have created:

```
prompt> db_stat -h TXNAPP -d color
53162   Btree magic number.
8       Btree version number.
Flags:
2       Minimum keys per-page.
8192    Underlying database page size.
1       Number of levels in the tree.
0       Number of unique keys in the tree.
0       Number of data items in the tree.
0       Number of tree internal pages.
0       Number of bytes free in tree internal pages (0% ff).
1       Number of tree leaf pages.
8166    Number of bytes free in tree leaf pages (0.% ff).
0       Number of tree duplicate pages.
0       Number of bytes free in tree duplicate pages (0% ff).
0       Number of tree overflow pages.
0       Number of bytes free in tree overflow pages (0% ff).
0       Number of pages on the free list.
```

Recoverability and Deadlock Avoidance

The first reason listed for using transactions was recoverability. Any logical change to a database may require multiple changes to underlying data structures. For example, modifying a record in a Btree may require leaf and internal pages to split, so a single **DB→put** method call can potentially require that multiple physical database pages be written. If only some of those pages are written and then the system or application fails, the database is left inconsistent and cannot be used until it has been recovered; that is, until the partially completed changes have been undone.

Write-ahead-logging is the term that describes the underlying implementation that Berkeley DB uses to ensure recoverability. What it means is that before any change is made to a database, information about the change is written to a database log. During recovery, the log is read, and databases are checked to ensure that changes described in the log for committed transactions appear in the database. Changes that appear in the database but are related to aborted or unfinished transactions in the log are undone from the database.

For recoverability after application or system failure, operations that modify the database must be protected by transactions. More specifically, operations are not recoverable unless a transaction is begun and each operation is associated with the transaction via the Berkeley DB interfaces, and then the transaction successfully committed. This is true even if logging is turned on in the database environment.

Here is an example function that updates a record in a database in a transactionally protected manner. The function takes a key and data items as arguments and then attempts to store them into the database.

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

```
        extern char *optarg;
        extern int optind;
        DB *db_cats, *db_color, *db_fruit;
        DB_ENV *dbenv;
        pthread_t ptid;
        int ch;

        while ((ch = getopt(argc, argv, "")) != EOF)
                switch (ch) {
                case '?':
                default:
                        usage();
                }
        argc -= optind;
        argv += optind;

        env_dir_create();
        env_open(&dbenv);

       /* Open database: Key is fruit class; Data is specific type. */
       db_open(dbenv, &db_fruit, "fruit", 0);

       /* Open database: Key is a color; Data is an integer. */
       db_open(dbenv, &db_color, "color", 0);

       /*
        * Open database:
        *      Key is a name; Data is: company name, address, cat breeds.
        */
       db_open(dbenv, &db_cats, "cats", 1);

       add_fruit(dbenv, db_fruit, "apple", "yellow delicious");

       return (0);
}

void
add_fruit(DB_ENV *dbenv, DB *db, char *fruit, char *name)
{
       DBT key, data;
       DB_TXN *tid;
       int ret;

       /* Initialization. */
       memset(&key, 0, sizeof(key));
       memset(&data, 0, sizeof(data));
       key.data = fruit;
       key.size = strlen(fruit);
       data.data = name;
       data.size = strlen(name);
```

```
        for (;;) {
                /* Begin the transaction. */
                if ((ret = txn_begin(dbenv, NULL, &tid, 0)) != 0) {
                        dbenv→err(dbenv, ret, "txn_begin");
                        exit (1);
                }

                /* Store the value. */
                switch (ret = db→put(db, tid, &key, &data, 0)) {
                case 0:
                        /* Success: commit the change. */
                        if ((ret = txn_commit(tid, 0)) != 0) {
                                dbenv→err(dbenv, ret, "txn_commit");
                                exit (1);
                        }
                        return;
                case DB_LOCK_DEADLOCK:
                        /* Deadlock: retry the operation. */
                        if ((ret = txn_abort(tid)) != 0) {
                                dbenv→err(dbenv, ret, "txn_abort");
                                exit (1);
                        }
                        break;
                 default:
                         /* Error: run recovery. */
                         dbenv→err(dbenv, ret, "dbc→put: %s/%s", fruit, name);
                         exit (1);
                 }
         }
}
```

The second reason listed for using transactions was deadlock avoidance. Each database operation (that is, any call to a function underlying the handles returned by **DB→open** and **DB→cursor**) is normally performed on behalf of a unique locker. If multiple calls on behalf of the same locker are desired within a single thread of control, transactions must be used. For example, consider the case in which a cursor scan locates a record and then accesses some other item in the database, based on that record. If these operations are done using the default lockers for the handle, they may conflict. If the application wishes to guarantee that the operations do not conflict, locks must be obtained on behalf of a transaction, instead of the default locker ID; and a transaction must be specified to subsequent **DB→cursor** and other Berkeley DB calls.

There is a new error return in this function that you may not have seen before. In Transactional (not Concurrent Data Store) applications supporting both readers and writers, or just multiple writers, Berkeley DB functions have an additional possible error return: DB_LOCK_DEADLOCK. This return means that our thread of control was deadlocked with another thread of control, and our thread was selected to discard all its Berkeley DB resources in order to resolve the problem. In the sample code, any time the **DB→put** function returns DB_LOCK_DEADLOCK, the transaction is

aborted (by calling **txn_abort**, which releases the transaction's Berkeley DB resources and undoes any partial changes to the databases) and then the transaction is retried from the beginning.

There is no requirement that the transaction be attempted again, but that is a common course of action for applications. Applications may want to set an upper boundary on the number of times an operation will be retried because some operations on some data sets may simply be unable to succeed. For example, updating all the pages on a large Web site during prime business hours may simply be impossible because of the high access rate to the database.

Atomicity

The third reason listed for using transactions is atomicity. Consider an application suite in which multiple threads of control (multiple processes or threads in one or more processes) are changing the values associated with a key in one or more databases. Specifically, they are taking the current value, incrementing it, and then storing it back into the database.

Such an application requires atomicity. Because we want to change a value in the database, we must make sure that after we read it, no other thread of control modifies it. For example, assume that both thread #1 and thread #2 are doing similar operations in the database, where thread #1 is incrementing records by 3, and thread #2 is incrementing records by 5. We want to increment the record by a total of 8. If the operations interleave in the right (well, wrong) order, that is not what will happen:

```
thread #1  read record: the value is 2
thread #2  read record: the value is 2
thread #2  write record + 5 back into the database (new value 7)
thread #1  write record + 3 back into the database (new value 5)
```

As you can see, instead of incrementing the record by a total of 8, we've incremented it only by 3 because thread #1 overwrote thread #2's change. By wrapping the operations in transactions, we ensure that this cannot happen. In a transaction, when the first thread reads the record, locks are acquired that will not be released until the transaction finishes, guaranteeing that all other readers and writers will block, waiting for the first thread's transaction to complete (or to be aborted).

Here is an example function that does transaction-protected increments on database records to ensure atomicity:

```
int
main(int argc, char *argv)
{
        extern char *optarg;
        extern int optind;
        DB *db_cats, *db_color, *db_fruit;
        DB_ENV *dbenv;
```

```
        pthread_t ptid;
        int ch;


        while ((ch = getopt(argc, argv, "")) != EOF)
                switch (ch) {
                case '?':
                default:
                        usage();
                }
        argc -= optind;
        argv += optind;

        env_dir_create();
        env_open(&dbenv);

        /* Open database: Key is fruit class; Data is specific type. */
        db_open(dbenv, &db_fruit, "fruit", 0);

        /* Open database: Key is a color; Data is an integer. */
        db_open(dbenv, &db_color, "color", 0);

        /*
         * Open database:
         *      Key is a name; Data is: company name, address, cat breeds.
         */
        db_open(dbenv, &db_cats, "cats", 1);

        add_fruit(dbenv, db_fruit, "apple", "yellow delicious");

        add_color(dbenv, db_color, "blue", 0);
        add_color(dbenv, db_color, "blue", 3);

        return (0);
}

void
add_color(DB_ENV *dbenv, DB *dbp, char *color, int increment)
{
        DBT key, data;
        DB_TXN *tid;
        int original, ret;
        char buf64;

        /* Initialization. */
        memset(&key, 0, sizeof(key));
        key.data = color;
        key.size = strlen(color);
        memset(&data, 0, sizeof(data));
        data.flags = DB_DBT_MALLOC;
```

```
for (;;) {
        /* Begin the transaction. */
        if ((ret = txn_begin(dbenv, NULL, &tid, 0)) != 0) {
                dbenv→err(dbenv, ret, "txn_begin");
                exit (1);
        }

        /*
         * Get the key. If it exists, we increment the value. If it
         * doesn't exist, we create it.
         */
        switch (ret = dbp→get(dbp, tid, &key, &data, 0)) {
        case 0:
               original = atoi(data.data);
               break;
        case DB_LOCK_DEADLOCK:
                /* Deadlock: retry the operation. */
                if ((ret = txn_abort(tid)) != 0) {
                        dbenv→err(dbenv, ret, "txn_abort");
                        exit (1);
                }
                continue;
        case DB_NOTFOUND:
                original = 0;
                break;
        default:
                /* Error: run recovery. */
                dbenv→err(
                    dbenv, ret, "dbc→get: %s/%d", color, increment);
                exit (1);
        }
        if (data.data != NULL)
                free(data.data);

        /* Create the new data item. */
        (void)snprintf(buf, sizeof(buf), "%d", original + increment);
        data.data = buf;
        data.size = strlen(buf) + 1;

        /* Store the new value. */
        switch (ret = dbp→put(dbp, tid, &key, &data, 0)) {
        case 0:
                /* Success: commit the change. */
                if ((ret = txn_commit(tid, 0)) != 0) {
                        dbenv→err(dbenv, ret, "txn_commit");
                        exit (1);
                }
                return;
        case DB_LOCK_DEADLOCK:
                /* Deadlock: retry the operation. */
                if ((ret = txn_abort(tid)) != 0) {
                        dbenv→err(dbenv, ret, "txn_abort");
                        exit (1);
```

```
                }
                break;
            default:
                /* Error: run recovery. */
                dbenv→err(
                    dbenv, ret, "dbc→put: %s/%d", color, increment);
                exit (1);
            }
        }
    }
```

Any number of operations on any number of databases can be included in a single transaction to ensure the atomicity of the operations. There is, however, a trade-off between the number of operations included in a single transaction and both throughput and the possibility of deadlock. The reason for this is because transactions acquire locks throughout their lifetime, and do not release the locks until commit or abort time. So, the more operations included in a transaction, the more likely it is that a transaction will block other operations and that deadlock will occur. However, each transaction commit requires a synchronous disk I/O, so grouping multiple operations into a transaction can increase overall throughput. (There is one exception to this. The DB_TXN_NOSYNC option causes transactions to exhibit the ACI (atomicity, consistency and isolation) properties, but not D (durability); avoiding the synchronous disk I/O on transaction commit and greatly increasing transaction throughput for some applications.

When applications do create complex transactions, they often avoid having more than one complex transaction at a time because simple operations such as a single **DB→put** are unlikely to deadlock with each other or the complex transaction; whereas multiple complex transactions are likely to deadlock with each other because they will both acquire many locks over their lifetime. Alternatively, complex transactions can be broken up into smaller sets of operations, and each of those sets may be encapsulated in a nested transaction. Because nested transactions may be individually aborted and retried without causing the entire transaction to be aborted, this allows complex transactions to proceed even in the face of heavy contention, repeatedly trying the suboperations until they succeed.

It is also helpful to order operations within a transaction; that is, access the databases and items within the databases in the same order, to the extent possible, in all transactions. Accessing databases and items in different orders greatly increases the likelihood of operations being blocked and failing due to deadlocks.

Repeatable Reads

The fourth reason listed for using transactions is *repeatable reads*. Generally, most applications do not need to place reads inside a transaction for performance reasons. The problem is that a transactionally protected cursor, reading each key/data pair in a database, will acquire a read lock on most of the pages in the database, and so will gradually block all write operations on the databases until the transaction commits or aborts. Note, however, that if there are update transactions present in the application, the

reading transactions must still use locking, and should be prepared to repeat any operation (possibly closing and reopening a cursor) that fails with a return value of DB_LOCK_DEADLOCK.

The exceptions to this rule are when the application is doing a read-modify-write operation and so requires atomicity, and when an application requires the ability to repeatedly access a data item knowing that it will not have changed. A repeatable read simply means that, for the life of the transaction, every time a request is made by any thread of control to read a data item, it will be unchanged from its previous value; that is, that the value will not change until the transaction commits or aborts.

Transactional Cursors

Berkeley DB cursors may be used inside a transaction, exactly as any other DB method. The enclosing transaction ID must be specified when the cursor is created, but it does not then need to be further specified on operations performed using the cursor. One important point to remember is that a cursor must be closed before the enclosing transaction is committed or aborted.

The following code fragment uses a cursor to store a new key in the cats database with four associated data items. The key is a name. The data items are a company name, an address, and a list of the breeds of cat owned. Each of the data entries is stored as a duplicate data item. In this example, transactions are necessary to ensure that either all or none of the data items appear in case of system or application failure:

```
int
main(int argc, char *argv)
{
        extern char *optarg;
        extern int optind;
        DB *db_cats, *db_color, *db_fruit;
        DB_ENV *dbenv;
        pthread_t ptid;
        int ch;

        while ((ch = getopt(argc, argv, "")) != EOF)
                switch (ch) {
                case '?':
                default:
                        usage();
                }
        argc -= optind;
        argv += optind;

        env_dir_create();
        env_open(&dbenv);

        /* Open database: Key is fruit class; Data is specific type. */
        db_open(dbenv, &db_fruit, "fruit", 0);
```

```
        /* Open database: Key is a color; Data is an integer. */
        db_open(dbenv, &db_color, "color", 0);

        /*
         * Open database:
         * Key is a name; Data is: company name, address, cat breeds.
         */
        db_open(dbenv, &db_cats, "cats", 1);

        add_fruit(dbenv, db_fruit, "apple", "yellow delicious");

        add_color(dbenv, db_color, "blue", 0);
        add_color(dbenv, db_color, "blue", 3);

        add_cat(dbenv, db_cats,
                "Amy Adams",
                "Sleepycat Software",
                "394 E. Riding Dr., Carlisle, MA 01741, USA",
                "abyssinian",
                "bengal",
                "chartreaux",
                NULL);

        return (0);
}

   void
add_cat(DB_ENV *dbenv, DB *db, char *name, ...)
{
        va_list ap;
        DBC *dbc;
        DBT key, data;
        DB_TXN *tid;
        int ret;
        char *s;

        /* Initialization. */
        memset(&key, 0, sizeof(key));
        memset(&data, 0, sizeof(data));
        key.data = name;
        key.size = strlen(name);

retry: /* Begin the transaction. */
        if ((ret = txn_begin(dbenv, NULL, &tid, 0)) != 0) {
                dbenv→err(dbenv, ret, "txn_begin");
                exit (1);
        }

        /* Delete any previously existing item. */
        switch (ret = db→del(db, tid, &key, 0)) {
        case 0:
```

```
case DB_NOTFOUND:
        break;
case DB_LOCK_DEADLOCK:
        /* Deadlock: retry the operation. */
        if ((ret = txn_abort(tid)) != 0) {
                dbenv→err(dbenv, ret, "txn_abort");
                exit (1);
        }
        goto retry;
default:
        dbenv→err(dbenv, ret, "db→del: %s", name);
        exit (1);
}

/* Create a cursor. */
if ((ret = db→cursor(db, tid, &dbc, 0)) != 0) {
        dbenv→err(dbenv, ret, "db→cursor");
        exit (1);
}

/* Append the items, in order. */
va_start(ap, name);
while ((s = va_arg(ap, char *)) != NULL) {
        data.data = s;
        data.size = strlen(s);
        switch (ret = dbc→c_put(dbc, &key, &data, DB_KEYLAST)) {
        case 0:
               break;
        case DB_LOCK_DEADLOCK:
                va_end(ap);

                /* Deadlock: retry the operation. */
                if ((ret = dbc→c_close(dbc)) != 0) {
                        dbenv→err( dbenv, ret, "dbc→c_close");
                        exit (1);
                }
                if ((ret = txn_abort(tid)) != 0) {
                        dbenv→err(dbenv, ret, "txn_abort");
                        exit (1);
                }
                goto retry;
        default:
                /* Error: run recovery. */
                dbenv→err(dbenv, ret, "dbc→put: %s/%s", name, s);
                exit (1);
        }
}
va_end(ap);
```

```
        /* Success: commit the change. */
        if ((ret = dbc→c_close(dbc)) != 0) {
                dbenv→err(dbenv, ret, "dbc→c_close");
                exit (1);
        }
        if ((ret = txn_commit(tid, 0)) != 0) {
                dbenv→err(dbenv, ret, "txn_commit");
                exit (1);
        }
}
```

Nested Transactions

Berkeley DB provides support for nested transactions. Nested transactions allow an application to decompose a large or long-running transaction into smaller units that may be independently aborted.

Normally, when beginning a transaction, the application will pass a NULL value for the parent argument to **txn_begin**. If, however, the parent argument is a DB_TXN handle, the newly created transaction will be treated as a nested transaction within the parent. Transactions may nest arbitrarily deeply. For the purposes of this discussion, transactions created with a parent identifier will be called *child transactions*.

Once a transaction becomes a parent, as long as any of its child transactions are unresolved (that is, they have neither committed nor aborted), the parent may not issue any Berkeley DB calls except to begin more child transactions, or to commit or abort. For example, it may not issue any access method or cursor calls. After all of a parent's children have committed or aborted, the parent may again request operations on its own behalf.

The semantics of nested transactions are as follows. When a child transaction is begun, it inherits all the locks of its parent. This means that the child will never block waiting on a lock held by its parent. Further, locks held by two children of the same parent will also conflict. To make this concrete, consider the following set of transactions and lock acquisitions.

Transaction T1 is the parent transaction. It acquires a write lock on item A and then begins two child transactions: C1 and C2. C1 also wishes to acquire a write lock on A; this succeeds. If C2 attempts to acquire a write lock on A, it will block until C1 releases the lock, at which point it will succeed. Now, let's say that C1 acquires a write lock on B. If C2 now attempts to obtain a lock on B, it will block. However, let's now assume that C1 commits. Its locks are anti-inherited, which means they are given to T1, so T1 will now hold a lock on B. At this point, C2 would be unblocked and would then acquire a lock on B.

Child transactions are entirely subservient to their parent transaction. They may abort, undoing their operations regardless of the eventual fate of the parent. However, even if a child transaction commits, if its parent transaction is eventually aborted, the child's changes are undone and the child's transaction is effectively aborted. Any child

transactions that are not yet resolved when the parent commits or aborts are resolved based on the parent's resolution—committing if the parent commits and aborting if the parent aborts. Any child transactions that are not yet resolved when the parent prepares are also prepared.

Environment Infrastructure

When building transactional applications, it is usually necessary to build an administrative infrastructure around the database environment. There are five components to this infrastructure, and each is supported by the Berkeley DB package in two different ways: a standalone utility and one or more library interfaces.

- Deadlock detection: **db_deadlock, lock_detect**, **DBENV→set_lk_detect**
- Checkpoints: **db_checkpoint**, **txn_checkpoint**
- Database and log file archival: **db_archive**, **log_archive**
- Log file removal: **db_archive**, **log_archive**
- Recovery procedures: **db_recover**, **DBENV→open**

When writing multithreaded server applications and/or applications intended for download from the Web, it is usually simpler to create local threads that are responsible for administration of the database environment because scheduling is often simpler in a single-process model, and only a single binary need be installed and run. However, the supplied utilities can be generally useful tools even when the application is responsible for doing its own administration because applications rarely offer external interfaces to database administration. The utilities are required when programming to a Berkeley DB scripting interface because the scripting APIs do not always offer interfaces to the administrative functionality.

Deadlock Detection

The first component of the infrastructure, *deadlock detection*, is not so much a requirement specific to transaction-protected applications, but instead is necessary for almost all applications in which more than a single thread of control will be accessing the database at one time. Although Berkeley DB automatically handles database locking, it is normally possible for deadlock to occur. It is not required by all transactional applications, but exceptions are rare.

When the deadlock occurs, two (or more) threads of control each request additional locks that can never be granted because one of the threads of control waiting holds the requested resource.

For example, consider two processes: A and B. Let's say that A obtains an exclusive lock on item X, and B obtains an exclusive lock on item Y. Then, A requests a lock on Y, and B requests a lock on X. A will wait until resource Y becomes available and B will wait until resource X becomes available. Unfortunately, because both A and B are waiting, neither will release the locks they hold and neither will ever obtain the

resource on which it is waiting. In order to detect that deadlock has happened, a separate process or thread must review the locks currently held in the database. If deadlock has occurred, a victim must be selected, and that victim will then return the error DB_LOCK_DEADLOCK from whatever Berkeley DB call it was making.

Berkeley DB provides a separate UNIX-style utility that can be used to perform this deadlock detection, named **db_deadlock**. Alternatively, applications can create their own deadlock utility or thread using the underlying **lock_detect** function, or specify that Berkeley DB run the deadlock detector internally whenever there is a conflict over a lock (see **DBENV→set_lk_detect** for more information). The following code fragment does the latter:

```
void
env_open(DB_ENV **dbenvp)
{
        DB_ENV *dbenv;
        int ret;


        /* Create the environment handle. */
        if ((ret = db_env_create(&dbenv, 0)) != 0) {
                fprintf(stderr,
                    "txnapp: db_env_create: %s\n", db_strerror(ret));
                exit (1);
        }

        /* Set up error handling. */
        dbenv→set_errpfx(dbenv, "txnapp");

        /* Do deadlock detection internally. */
        if ((ret = dbenv→set_lk_detect(dbenv, DB_LOCK_DEFAULT)) != 0) {
                dbenv→err(dbenv, ret, "set_lk_detect:DB_LOCK_DEFAULT");
                exit (1);
        }

        /*
         * Open a transactional environment:
         *      create if it doesn't exist
         *      free-threaded handle
         *      run recovery
         *      read/write owner only
         */
        if ((ret = dbenv→open(dbenv, ENV_DIRECTORY,
            DB_CREATE | DB_INIT_LOCK | DB_INIT_LOG |
            DB_INIT_MPOOL | DB_INIT_TXN | DB_RECOVER | DB_THREAD,
            S_IRUSR | S_IWUSR)) != 0) {
               dbenv >err(dbenv, ret, "dbenv→open: %s", ENV_DIRECTORY);
               exit (1);
        }

        *dbenvp = dbenv;
}
```

Deciding how often to run the deadlock detector and which of the deadlocked transactions will be forced to abort when the deadlock is detected is a common tuning parameter for Berkeley DB applications.

Performing Checkpoints

The second component of the infrastructure is performing checkpoints of the log files. As transactions commit, change records are written into the log files, but the actual changes to the database are not necessarily written to disk. When a checkpoint is performed, the changes to the database that are part of committed transactions are written into the backing database file.

Performing checkpoints is necessary for two reasons. First, you can remove the Berkeley DB log files from your system only after a checkpoint. Second, the frequency of your checkpoints is inversely proportional to the amount of time it takes to run database recovery after a system or application failure.

Once the database pages are written, log files can be archived and removed from the system because they will never be needed for anything other than catastrophic failure. In addition, recovery after system or application failure has to redo or undo changes only since the last checkpoint because changes before the checkpoint have all been flushed to the filesystem.

Berkeley DB provides a separate utility, **db_checkpoint**, which can be used to perform checkpoints. Alternatively, applications can write their own checkpoint utility using the underlying **txn_checkpoint** function. The following code fragment checkpoints the database environment every 60 seconds:

```
int
main(int argc, char *argv)
{
        extern char *optarg;
        extern int optind;
        DB *db_cats, *db_color, *db_fruit;
        DB_ENV *dbenv;
        pthread_t ptid;
        int ch;

        while ((ch = getopt(argc, argv, "")) != EOF)
                switch (ch) {
                case '?':
                default:
                        usage();
                }
        argc -= optind;
        argv += optind;

        env_dir_create();
        env_open(&dbenv);
```

```
        /* Start a checkpoint thread. */
        if ((errno = pthread_create(
            &ptid, NULL, checkpoint_thread, (void *)dbenv)) != 0) {
                fprintf(stderr,
                    "txnapp: failed spawning checkpoint thread: %s\n",
                     strerror(errno));
                exit (1);
        }

        /* Open database: Key is fruit class; Data is specific type. */
        db_open(dbenv, &db_fruit, "fruit", 0);

        /* Open database: Key is a color; Data is an integer. */
        db_open(dbenv, &db_color, "color", 0);

        /*
         * Open database:
         *      Key is a name; Data is: company name, address, cat breeds.
         */
        db_open(dbenv, &db_cats, "cats", 1);

        add_fruit(dbenv, db_fruit, "apple", "yellow delicious");

        add_color(dbenv, db_color, "blue", 0);
        add_color(dbenv, db_color, "blue", 3);

        add_cat(dbenv, db_cats,
               "Amy Adams",
               "Sleepycat Software",
               "118 Tower Rd., Lincoln, MA 01741, USA",
               "abyssinian",
               "bengal",
               "chartreaux",
               NULL);

        return (0);
}

void *
checkpoint_thread(void *arg)
{
        DB_ENV *dbenv;
        int ret;

        dbenv = arg;
        dbenv→errx(dbenv, "Checkpoint thread: %lu", (u_long)pthread_self());

        /* Checkpoint once a minute. */
        for (;; sleep(60))
                switch (ret = txn_checkpoint(dbenv, 0, 0, 0)) {
                case 0:
```

```
                case DB_INCOMPLETE:
                        break;
                default:
                        dbenv→err(dbenv, ret, "checkpoint thread");
                        exit (1);
                }

        /* NOTREACHED */
}
```

Because checkpoints can be quite expensive, choosing how often to perform a checkpoint is a common tuning parameter for Berkeley DB applications.

Database and Log File Archival Procedures

The third component of the administrative infrastructure, archival for catastrophic recovery, concerns the recoverability of the database in the face of catastrophic failure. Recovery after catastrophic failure is intended to minimize data loss when physical hardware has been destroyed—for example, loss of a disk that contains databases or log files. Although the application may still experience data loss in this case, it is possible to minimize it.

First, you may want to periodically create snapshots (that is, backups) of your databases to make it possible to recover from catastrophic failure. These snapshots are either a standard backup, which creates a consistent picture of the databases as of a single instant in time; or an online backup (also known as a *hot* backup), which creates a consistent picture of the databases as of an unspecified instant during the period of time when the snapshot was made. The advantage of a hot backup is that applications may continue to read and write the databases while the snapshot is being taken. The disadvantage of a hot backup is that more information must be archived, and recovery based on a hot backup is to an unspecified time between the start of the backup and when the backup is completed.

Second, after taking a snapshot, you should periodically archive the log files being created in the environment. It is often helpful to think of database archival in terms of full and incremental filesystem backups. A snapshot is a full backup, whereas the periodic archival of the current log files is an incremental backup. For example, it might be reasonable to take a full snapshot of a database environment weekly or monthly, and archive additional log files daily. Using both the snapshot and the log files, a catastrophic crash at any time can be recovered to the time of the most recent log archival; a time long after the original snapshot.

To create a standard backup of your database that can be used to recover from catastrophic failure, take the following steps:

1. Commit or abort all ongoing transactions.
2. Force an environment checkpoint (see **db_checkpoint** for more information).

3. Stop writing your databases until the backup has completed. Read-only operations are permitted, but no write operations and no filesystem operations may be performed (for example, the **DBENV→remove** and **DB→open** functions may not be called).
4. Run **db_archive -l** to identify all the log files, and copy the last one (that is, the one with the highest number) to a backup device such as a CD-ROM, alternate disk, or tape.
5. Run **db_archive -s** to identify all the database data files, and copy them to a backup device such as a CD-ROM, alternate disk, or tape.

   If the database files are stored in a separate directory from the other Berkeley DB files, it may be simpler to archive the directory itself instead of the individual files (see **DBENV→set_data_dir** for additional information). Note: if any of the database files did not have an open DB handle during the lifetime of the current log files, db_archive will not list them in its output! This is another reason it may be simpler to use a separate database file directory and archive the entire directory, instead of archiving only the files listed by **db_archive**.

To create a *hot* backup of your database that can be used to recover from catastrophic failure, take the following steps:

1. Archive your databases, as described in the previous step 4. You do not have to halt ongoing transactions or force a checkpoint. In the case of a hot backup, the utility you use to copy the databases must read database pages atomically (as described in "Berkeley DB Recoverability").
2. When performing a hot backup, you must additionally archive the active log files. Note that the order of these two operations is required, and the database files must be archived before the log files. This means that if the database files and log files are in the same directory, you cannot simply archive the directory; you must make sure that the correct order of archival is maintained.

   To archive your log files, run the **db_archive** utility using the -l option to identify all the database log files, and copy them to your backup media. If the database log files are stored in a separate directory from the other database files, it may be simpler to archive the directory itself instead of the individual files (see the **DBENV→set_lg_dir** function for more information).

Once these steps are completed, your database can be recovered from catastrophic failure (see "Recovery Procedures" for more information).

To update your snapshot so that recovery from catastrophic failure is possible up to a new point in time, repeat step 2 under the hot backup instructions—copying all existing log files to a backup device. This is applicable to both standard and hot backups; that is, you can update snapshots made either way. Each time both the database and log files are copied to backup media, you may discard all previous database snapshots and saved log files. Archiving additional log files does not allow you to discard either previous database snapshots or log files.

The time to restore from catastrophic failure is a function of the number of log records that have been written since the snapshot was originally created. Perhaps more importantly, the more separate pieces of backup media you use, the more likely it is that you will have a problem reading from one of them. For these reasons, it is often best to make snapshots on a regular basis.

Obviously, the reliability of your archive media will affect the safety of your data. For archival safety, ensure that you have multiple copies of your database backups, verify that your archival media is error-free and readable, and that copies of your backups are stored offsite!

The functionality provided by the **db_archive** utility is also available directly from the Berkeley DB library. The following code fragment prints out a list of log and database files that need to be archived:

```
void
log_archlist(DB_ENV *dbenv)
{
        int ret;
        char **begin, **list;

        /* Get the list of database files. */
        if ((ret = log_archive(dbenv,
            &list, DB_ARCH_ABS | DB_ARCH_DATA, NULL)) != 0) {
                dbenv->err(dbenv, ret, "log_archive: DB_ARCH_DATA");
                exit (1);
        }
        if (list != NULL) {
                for (begin = list; *list != NULL; ++list)
                        printf("database file: %s\n", *list);
                free (begin);
        }

        /* Get the list of log files. */
        if ((ret = log_archive(dbenv,
            &list, DB_ARCH_ABS | DB_ARCH_LOG, NULL)) != 0) {
                dbenv->err(dbenv, ret, "log_archive: DB_ARCH_LOG");
               exit (1);
        }
        if (list != NULL) {
                for (begin = list; *list != NULL; ++list)
                        printf("log file: %s\n", *list);
                free (begin);
        }
}
```

Log File Removal

The fourth component of the infrastructure, log file removal, concerns the ongoing disk consumption of the database log files. Depending on the rate at which the application writes to the databases and the available disk space, the number of log files may increase quickly enough so that disk space will be a resource problem. For this reason,

you will periodically want to remove log files in order to conserve disk space. This procedure is distinct from database and log file archival for catastrophic recovery, and you cannot remove the current log files simply because you have created a database snapshot or copied log files to archival media.

Log files may be removed at any time, as long as

- the log file is not involved in an active transaction.
- at least two checkpoints have been written subsequent to the log file's creation.
- the log file is not the only log file in the environment.

Obviously, if you are preparing for catastrophic failure, you will want to copy the log files to archival media before you remove them.

To remove log files, take the following steps:

1. If you are concerned with catastrophic failure, first copy the log files to backup media, as described in "Archival for Catastrophic Recovery."
2. Run **db_archive** without options to identify all the log files that are no longer in use (for example, no longer involved in an active transaction).
3. Remove those log files from the system.

The functionality provided by the **db_archive** utility is also available directly from the Berkeley DB library. The following code fragment removes log files that are no longer needed by the database environment:

```
int
main(int argc, char *argv)
{
        ...


        /* Start a logfile removal thread. */
        if ((errno = pthread_create(
            &ptid, NULL, logfile_thread, (void *)dbenv)) != 0) {
                fprintf(stderr,
                    "txnapp: failed spawning log file removal thread: %s\n",
                     strerror(errno));
                exit (1);
        }

        ...
}

void *
logfile_thread(void *arg)
{
        DB_ENV *dbenv;
        int ret;
        char **begin, **list;
```

```
        dbenv = arg;
        dbenv→errx(dbenv,
            "Log file removal thread: %lu", (u_long)pthread_self());

        /* Check once every 5 minutes. */
        for (;; sleep(300)) {
                /* Get the list of log files. */
                if ((ret = log_archive(dbenv, &list, DB_ARCH_ABS, NULL)) != 0) {
                        dbenv→err(dbenv, ret, "log_archive");
                        exit (1);
                }

                /* Remove the log files. */
                if (list != NULL) {
                        for (begin = list; *list != NULL; ++list)
                                if ((ret = remove(*list)) != 0) {
                                        dbenv→err(dbenv,
                                             ret, "remove %s", *list);
                                        exit (1);
                                }
                        free (begin);
                }
        }
        /* NOTREACHED */
}
```

Recovery Procedures

The fifth component of the infrastructure, recovery procedures, concerns the recoverability of the database. After any application or system failure, there are two possible approaches to database recovery:

1. There is no need for recoverability, and all databases can be re-created from scratch. Although these applications may still need transaction protection for other reasons, recovery usually consists of removing the Berkeley DB environment home directory and all files it contains, and then restarting the application.
2. It is necessary to recover information after system or application failure. In this case, recovery processing must be performed on any database environments that were active at the time of the failure. Recovery processing involves running the **db_recover** utility or calling the **DBENV→open** function with the DB_RECOVER or DB_RECOVER_FATAL flags. During recovery processing, all database changes made by aborted or unfinished transactions are undone, and all database changes made by committed transactions are redone, as necessary. Database applications must not be restarted until recovery completes. After recovery finishes, the environment is properly initialized so that applications may be restarted.

If you intend to do recovery, there are two possible types of recovery processing:

- *Catastrophic recovery*. A failure that requires catastrophic recovery is a failure in which either the database or log files are destroyed or corrupted. For example, catastrophic failure includes the case where the disk drive on which either the database or logs are stored has been physically destroyed, or when the system's normal filesystem recovery on startup cannot bring the database and log files to a consistent state. This is often difficult to detect, and is perhaps the most common sign of the need for catastrophic recovery is when the normal recovery procedures fail.

To restore your database environment after catastrophic failure, take the following steps:

1. Restore the most recent snapshots of the database and log files from the backup media into the system directory where recovery will be performed.
2. If any log files were archived since the last snapshot was made, they should be restored into the Berkeley DB environment directory where recovery will be performed. Make sure that you restore them in the order in which they were written. The order is important because it's possible that the same log file appears on multiple backups, and you want to run recovery using the most recent version of each log file.
3. Run the **db_recover** utility, specifying its -c option; or call the **DBENV→open** function, specifying the DB_RECOVER_FATAL flag. The catastrophic recovery process will review the logs and database files to bring the environment databases to a consistent state as of the time of the last uncorrupted log file that is found. It is important to realize that only transactions committed before that date will appear in the databases. It is possible to re-create the database in a location different from the original by specifying appropriate pathnames to the **-h** option of the **db_recover** utility. In order for this to work properly, it is important that your application reference files by names relative to the database home directory or the pathname(s) specified in calls to **DBENV→set_data_dir**, instead of using full path names.

- *Non-catastrophic* or *normal recovery*. If the failure is non-catastrophic and the database files and log are both accessible on a stable filesystem, run the **db_recover** utility without the -c option or call the **DBENV→open** function specifying the DB_RECOVER flag. The normal recovery process will review the logs and database files to ensure that all changes associated with committed transactions appear in the databases, and that all uncommitted transactions do not appear.

Recovery and Filesystem Operations

When running in a transaction-protected environment, database creation and deletion are logged as standalone transactions internal to Berkeley DB. That is, for each such operation, a new transaction is begun and aborted or committed internally, so that they will be recovered during recovery.

The Berkeley DB API supports removing and renaming files. Renaming files is supported by the **DB→rename** method, and removing files is supported by the **DB→remove** method. Berkeley DB does not permit specifying the DB_TRUNCATE flag when opening a file in a transaction-protected environment. This is an implicit file deletion, but one that does not always require the same operating system file permissions as deleting and creating a file do.

If you changed the name of a file or deleted it outside of the Berkeley DB library (for example, you explicitly removed a file using your normal operating system utilities), then it is possible that recovery will not be able to find a database referenced in the log. In this case, **db_recover** will produce a warning message, saying it was unable to locate a file it expected to find. This message is only a warning because the file may have been subsequently deleted as part of normal database operations before the failure occurred, so it is not necessarily a problem.

Generally, any filesystem operations that are performed outside the Berkeley DB interface should be performed at the same time as making a snapshot of the database. To perform filesystem operations correctly, do the following:

1. Cleanly shut down database operations.

   To shut down database operations cleanly, all applications accessing the database environment must be shut down and a transaction checkpoint must be taken. If the applications are not implemented so they can be shut down gracefully (that is, closing all references to the database environment), recovery must be performed after all applications have been killed to ensure that the underlying databases are consistent on disk.
2. Perform the filesystem operations; for example, remove or rename one or more files.
3. Make an archival snapshot of the database.

   Although this step is not strictly necessary, it is strongly recommended. If this step is not performed, recovery from catastrophic failure will require that recovery first be performed up to the time of the filesystem operations, the filesystem operations be redone, and then recovery be performed from the filesystem operations forward.
4. Restart the database applications.

Berkeley DB Recoverability

Berkeley DB recovery is based on write-ahead logging. This means that when a change is made to a database page, a description of the change is written into a log file. This description in the log file is guaranteed to be written to stable storage before the database pages that were changed are written to stable storage. This is the fundamental feature of the logging system that makes durability and rollback work.

If the application or system crashes, the log is reviewed during recovery. Any database changes described in the log that were part of committed transactions and that were never written to the actual database itself are written to the database as part of recovery. Any database changes described in the log that were never committed and that were written to the actual database itself are backed out of the database as part of recovery. This design allows the database to be written lazily, and only blocks from the log file have to be forced to disk as part of transaction commit.

There are two interfaces that are a concern when considering Berkeley DB recoverability:

- The interface between Berkeley DB and the operating system/filesystem.
- The interface between the operating system/filesystem and the underlying stable storage hardware.

Berkeley DB uses the operating system interfaces and its underlying filesystem when writing its files. This means that Berkeley DB can fail if the underlying filesystem fails in some unrecoverable way. Otherwise, the interface requirements here are simple: The system call that Berkeley DB uses to flush data to disk (normally **fsync**(2)) must guarantee that all the information necessary for a file's recoverability has been written to stable storage before it returns to Berkeley DB, and that no possible application or system crash can cause that file to be unrecoverable.

In addition, Berkeley DB implicitly uses the interface between the operating system and the underlying hardware. The interface requirements here are not as simple.

First, it is necessary to consider the underlying page size of the Berkeley DB databases. The Berkeley DB library performs all database writes using the page size specified by the application. These pages are not checksummed, and Berkeley DB assumes that they are written atomically. This means that if the operating system performs filesystem I/O in blocks of different sizes than the database page size, it may increase the possibility for database corruption. For example, assume that Berkeley DB is writing 32KB pages for a database, and the operating system does filesystem I/O in 16KB blocks. If the operating system writes the first 16KB of the database page successfully, but crashes before being able to write the second 16KB of the database, the database has been corrupted and this corruption will not be detected during recovery. For this reason, it may be important to select database page sizes that will be written as single block transfers by the underlying operating system.

Second, it is necessary to consider the behavior of the system's underlying stable storage hardware. For example, consider a SCSI controller that has been configured to cache data and return to the operating system that the data has been written to stable storage, when, in fact, it has only been written into the controller RAM cache. If power is lost before the controller is able to flush its cache to disk, and the controller cache is not stable (that is, the writes will not be flushed to disk when power returns), the writes will be lost. If the writes include database blocks, there is no loss because recovery will correctly update the database. If the writes include log file blocks, it is

possible that transactions that were already committed may not appear in the recovered database, although the recovered database will be coherent after a crash.

If the underlying hardware can fail in any way so that only part of the block was written, the failure conditions are the same as those described previously for an operating system failure that writes only part of a logical database block.

For these reasons, it is important to select hardware that does not do partial writes and does not cache data writes (or does not return that the data has been written to stable storage until it has either been written to stable storage or the actual writing of all of the data is guaranteed, barring catastrophic hardware failure—that is, your disk drive exploding). You should also be aware that Berkeley DB does not protect against all cases of stable storage hardware failure, nor does it protect against hardware misbehavior.

If the disk drive on which you are storing your databases explodes, you can perform normal Berkeley DB catastrophic recovery because it requires only a snapshot of your databases plus all of the log files you have archived since those snapshots were taken. In this case, you will lose no database changes at all. If the disk drive on which you are storing your log files explodes, you can still perform catastrophic recovery, but you will lose any database changes that were part of transactions committed since your last archival of the log files. For this reason, storing your databases and log files on different disks should be considered a safety measure as well as a performance enhancement.

Finally, if your hardware misbehaves (for example, if a SCSI controller writes incorrect data to the disk), Berkeley DB will not detect it, and your data may be corrupted.

Transaction Throughput

Generally, the speed of a database system is measured by the *transaction throughput*, expressed as the number of transactions per second. The two gating factors for Berkeley DB performance in a transactional system are usually the underlying database files and the log file. Both are factors because they require disk I/O, which is slow relative to other system resources such as CPU.

In the worst-case scenario:

- Database access is truly random and the database is too large to fit into the cache, resulting in a single I/O per requested key/data pair.
- Both the database and the log are on a single disk.

This means that for each transaction, Berkeley DB is potentially performing several filesystem operations:

- Disk seek to database file
- Database file read
- Disk seek to log file
- Log file write

- Flush log file information to disk
- Disk seek to update log file metadata (for example, inode information)
- Log metadata write
- Flush log file metadata to disk

There are a number of ways to increase transactional throughput, all of which attempt to decrease the number of filesystem operations per transaction:

- Tune the size of the database cache. If the Berkeley DB key/data pairs used during the transaction are found in the database cache, the seek and read from the database are no longer necessary, resulting in two fewer filesystem operations per transaction. To determine whether your cache size is too small, see "Selecting a Cache Size."
- Put the database and the log files on different disks. This allows reads and writes to the log files and the database files to be performed concurrently.
- Set the filesystem configuration so that file access and modification times are not updated. Note that although the file access and modification times are not used by Berkeley DB, they may affect other programs—so be careful.
- Upgrade your hardware. When considering the hardware on which to run your application, however, it is important to consider the entire system. The controller and bus can have as much to do with the disk performance as the disk itself. It is also important to remember that throughput is rarely the limiting factor, and that disk seek times are normally the true performance issue for Berkeley DB.
- Turn on the DB_TXN_NOSYNC flag. This changes the Berkeley DB behavior so that the log files are not flushed when transactions are committed. Although this change will greatly increase your transaction throughput, it means that transactions will exhibit the ACI (atomicity, consistency, and isolation) properties, but not D (durability). Database integrity will be maintained, but it is possible that some of the most recently committed transactions may be undone during recovery instead of being redone.

If you are bottlenecked on logging, the following test will help you confirm that the number of transactions per second that your application does is reasonable for the hardware on which you're running. Your test program should repeatedly perform the following operations:

- Seek to the beginning of a file.
- Write to the file.
- Flush the file write to disk.

The number of times that you can perform these three operations per second is a rough measure of the number of transactions per second of which the hardware is capable. This test simulates the operations applied to the log file. (As a simplifying

assumption in this experiment, we assume that the database files are either on a separate disk; or that they fit, with some few exceptions, into the database cache.) We do not have to directly simulate updating the log file directory information because it will normally be updated and flushed to disk as a result of flushing the log file write to disk.

Running this test program, in which we write 256 bytes for 1000 operations on reasonably standard commodity hardware (Pentium II CPU, SCSI disk), returned the following results:

```
% testfile -b256 -o1000
running: 1000 ops
Elapsed time: 16.641934 seconds
1000 ops:   60.09 ops per second
```

Note that the number of bytes being written to the log as part of each transaction can dramatically affect the transaction throughput. The test run used 256, which is a reasonable size log write. Your log writes may be different. To determine your average log write size, use the **db_stat** utility to display your log statistics.

As a quick sanity check, the average seek time is 9.4 msec for this particular disk, and the average latency is 4.17 msec. That results in a minimum requirement for a data transfer to the disk of 13.57 msec, or a maximum of 74 transfers per second. This is close enough to the previous 60 operations per second (which wasn't done on a quiescent disk) that the number is believable.

An implementation of the previous example test program for IEEE/ANSI Std 1003.1 (POSIX) standard systems is included in the Berkeley DB distribution.