Oracle Berkeley DB

Writing In-Memory Applications

11g Release 2

ORACLE

BERKELEY DB

Table of Contents

Introduction

This document describes how to write a DB application that keeps its data entirely in memory. That is, the application writes no data to disk. For this reason, in-memory only applications typically discard all data durability guarantees.

Note

This document assume familiarity with the *Getting Started with Berkeley DB* guide. If you are using environments or transactions, then you should also have an understanding of the concepts in *Berkeley DB Getting Started with Transaction Processing* guide.

There are several reasons why you might want to write an in-memory only DB application. For platforms on which a disk drive is available to back your data, an in-memory application might be desirable from a performance perspective. In this case, the data that your application manages might be generated during run-time and so is of no interest across application startups.

Other platforms are disk-less. In this case, an in-memory only configuration is the only possible choice. Note that this document's primary focus is disk-less systems for which an on-disk filesystem is not available.

Resources to be Managed

Before continuing, it is worthwhile to briefly describe the DB resources that must be managed if you are going to configure an in-memory only DB application. These are resources that are by default persisted on disk, or backed by a filesystem on disk. Some configuration is therefore required to keep these resources in-memory only.

Note that you can configure only some of these resources to be held in-memory, and allow others to be backed by disk. This might be desireable for some applications that wish to improve application performance by, for example, eliminating disk I/O for some, but not all, of these resources. However, for the purpose of this document, we assume you want to configure all of these resources to be held in memory.

Managing these resources for an in-memory application is described in detail later in this article.

- Database files

  Normally, DB stores your database data within on-disk files. For an entirely in-memory application, you are required to turn off this behavior.

- Environment region files

  DB environments manage region files for a variety of purposes. Normally these are backed by the filesystem, but by using the appropriate configuration option you can cause region files to reside in memory only.

- Database cache

The DB cache must be configured large enough to hold all your data in memory. If you do not size your cache large enough, then DB will attempt to write pages to disk. In a disk-less system, this will result in an abnormal termination of your program.

- Logs

  DB logs describe the write activity that has occurred in your application. They are used for a number of purposes, such as recovery operations for applications that are seeking data durability guarantees.

  For in-memory applications that do not care about durability guarantees, logs are still required if you want transactional benefits other than durability (such as isolation and atomicity). This is because DB's transactional subsystem requires logs, even if you want to discard all data durability guarantees.

  If this describes your application, you must enable logs but configure them to reside only within memory.

- Temporary overflow pages

  You must disallow backing temporary database files with the filesystem. This is mostly a configuration option, but it is also dependent upon sizing your cache correctly.

In addition to these, if you are writing a replicated application (see *Berkeley DB Getting Started with Replicated Applications* for an introduction to writing replicated applications), there is internal replication information that is normally kept on-disk. You can cause this information to be kept in-memory if you are willing to accept some limitations in how your replicated application operates. See In-Memory Replicated Applications (page 14) for more information.

Strategies

DB is an extremely flexible product that can be adapted to suit almost any data management requirements. This means that you can configure DB to operate entirely within memory, but still retain some data durability guarantees or even throw away all durability guarantees.

Data durability guarantees describe how persistent your data is. That is, once you have made a change to the data stored in your database, how much of a guarantee do you require that that modification will persist (not be lost)? There are a great many options here. For the absolute best durability guarantee, you should fully transaction-protect your data and allow your data to be written to disk upon each transaction commit. Of course, this guarantee is not available for disk-less systems.

At the opposite end of the spectrum, you can throw away all your data once your application is done with it (for example, at application shutdown). This is a good option if you are using DB only as a kind of caching mechanism. In this case, you obviously must either generate your data entirely during runtime, or obtain it from some remote location during application startup.

There are also durability options that exist somewhere in between these two extremes. For example, disk-less systems are sometimes backed by some kind of flash memory (e.g.

compact flash cards). These platforms may want to limit the number of writes applied to the backing media because it is capable of accepting only a limited number of writes before it must be replaced. For this reason, you might want to limit data writes to the flash media only during specific moments during your application's runtime; for example, only at application shutdown.

Another way to improve data durability for in-memory configurations is to use DB replication to commit data to the network. This strategy takes advantage of the fact that running clients have in-memory copies of the data and can take over in the event of an outage at the master. The use of replication in this way increases durability for your data while providing the benefit of avoiding disk I/O on transaction commit.

In-memory replicated applications are described in more detail in In-Memory Replicated Applications (page 14).

The point here is to be aware that a great many options are available to you when writing an in-memory only application. That said, the focus of this document is strictly disk-less systems; that is, systems that provide no means by which data can be written to persistent media.

Keeping the Database in Memory

Normally DB databases are backed by the filesystem. For in-memory applications, such as is required on disk-less systems, you can cause your database to only reside in-memory. That is, their contents are stored entirely within DB's cache.

There are two requirements for keeping your database(s) in-memory. The first is to size your cache such that it is big enough to hold all your data in-memory. If your cache fills up, then DB will return `ENOMEM` on the next operation that requests additional pages in the cache. As with all errors while updating a database, the current transaction must be aborted. If the update was being done without a transaction, then the application must close its environment and database handles, reopen them, and then refresh the database from some backup data source.

For information on setting the cache size, see Sizing the Cache (page 7).

Beyond cache sizing, you also must tell DB not to back your database with an on-disk file. You do this by NOT providing a database file name when you open the database. Note that the database file name is different from the database name; you can name your in-memory databases even if you are not storing them in an on-disk file.

For example:

```
#include "db.h"

...

    int ret, ret_c;
    const char *db_name = "in_mem_db1";
    u_int32_t db_flags;  /* For open flags */
    DB *dbp;             /* Database handle */

...
```

```
   /* Initialize the DB handle */
    ret = db_create(&dbp, NULL, 0);
    if (ret != 0) {
        fprintf(stderr, "Error creating database handle: %s\n",
                    db_strerror(ret));
        goto err;
    }

    db_flags = DB_CREATE;       /* If it doesn't exist, create it */

    /*
     * Open the database. Note that the file name is NULL.
     * This forces the database to be stored in the cache only.
     * Also note that the database has a name, even though its
     * file name is NULL.
     */
    ret = dbp->open(dbp,        /* Pointer to the database */
                    NULL,       /* Txn pointer */
                    NULL,       /* File name is not specified
                                 * on purpose       */
                    db_name,    /* Logical db name. */
                    DB_BTREE,   /* Database type (using btree) */
                    db_flags,   /* Open flags */
                    0);         /* File mode. Using defaults */
    if (ret != 0) {
        dbp->err(dbp, ret, "Database open failed");
        goto err;
    }

err:
    /* Close the database */
    if (dbp != NULL) {
        ret_c = dbp->close(dbp, 0);
        if (ret_c != 0) {
            fprintf(stderr, "%s database close failed.\n",
                 db_strerror(ret_c));
            ret = ret_c
        }
    }
```

Keeping Environments in Memory

Like databases, DB environments are usually backed by the filesystem. In fact, a big part of what environments do is identify the location on disk where resources (such as log and database files) are kept.

However, environments are also used for managing resources, such as obtaining new transactions, so they are useful even when building an in-memory application. Therefore, if

you are going to use an environment for your in-memory DB application, you must configure it such that it does not want to use the filesystem. There are two things you need to do here.

First, when you open your environment, do NOT identify a home directory. To accomplish this, you must:

- NOT provide a value for the db_home parameter on the DB_ENV->open() method.
- NOT have a DB_HOME environment variable set.
- NOT call any of the methods that affect file naming (DB_ENV->set_data_dir(), DB_ENV->set_lg_dir(), or DB_ENV->set_tmp_dir()).

Beyond this, you must also ensure that regions are backed by heap memory instead of by the filesystem or system shared memory. You do this when you open your environment by specifying the DB_PRIVATE flag. Note that the use of DB_PRIVATE means that other processes cannot share the environment. Consequently, your in-memory only application must be a single-process, although it can be multi-threaded.

For example:

```
#include "db.h"

...

    int ret, ret_c;
    u_int32_t env_flags;  /* For open flags */
    DB_ENV *envp;         /* Environment handle */

...

   /* Initialize the ENV handle */
    ret = db_env_create(&envp, 0);
    if (ret != 0) {
        fprintf(stderr, "Error creating environment handle: %s\n",
                    db_strerror(ret));
        goto err;
    }


    /*
     * Environment flags. These are for a non-threaded
     * in-memory application.
     */
    env_flags =
      DB_CREATE     |  /* Create the environment if it does not exist */
      DB_INIT_LOCK  |  /* Initialize the locking subsystem */
      DB_INIT_LOG   |  /* Initialize the logging subsystem */
      DB_INIT_TXN   |  /* Initialize the transactional subsystem. This
                        * also turns on logging. */
      DB_INIT_MPOOL |  /* Initialize the memory pool (in-memory cache) */
```

```
      DB_PRIVATE    |  /* Region files are not backed by the
                        * filesystem. Instead, they are backed by
                        * heap memory.  */

   /*
    * Now open the environment. Notice that we do not provide a location
    * for the environment's home directory. This is required for an
    * in-memory only application.
    */
    ret = envp->open(envp, NULL, env_flags, 0);
    if (ret != 0) {
        fprintf(stderr, "Error opening environment: %s\n",
            db_strerror(ret));
        goto err;
    }

err:
    /* Close the environment */
    if (envp != NULL) {
        ret_c = envp->close(envp, 0);
        if (ret_c != 0) {
            fprintf(stderr, "environment close failed: %s\n",
                 db_strerror(ret_c));
            ret = ret_c
        }
    }
```

Sizing the Cache

One of the most important considerations for an in-memory application is to ensure that your database cache is large enough. In a normal application that is not in-memory, the cache provides a mechanism by which frequently-used data can be accessed without resorting to disk I/O. For an in-memory application, the cache is the only location your data can exist so it is critical that you make the cache large enough for your data set.

You specify the size of your cache at application startup. Obviously you should not specify a size that is larger than available memory. Note that the size you specify for your cache is actually a *maximum* size; DB will only use memory as required so if you specify a cache size of 1 GB but your data set is only ever 10 MB in size, then 10 MB is what DB will use.

Note that if you specify a cache size less than 500 MB, then the cache size is automatically increased by 25% to account for internal overhead purposes.

There are two ways to specify a cache size, depending on whether you are using a database environment.

Specifying a Cache Size using the Database Handle

To select a cache size using the database handle, use the `DB->set_cachesize()` method. Note that you cannot use this method after the database has been opened.

Also, if you are using a database environment, it is an error to use this method. See the next section for details on selecting your cache size.

The following code fragment creates a database handle, sets the cache size to 10 MB and then opens the database:

```
#include "db.h"

...

    int ret, ret_c;
    const char *db_name = "in_mem_db1";
    u_int32_t db_flags;  /* For open flags */
    DB *dbp;             /* Database handle */

...

   /* Initialize the DB handle */
    ret = db_create(&dbp, NULL, 0);
    if (ret != 0) {
        fprintf(stderr, "Error creating database handle: %s\n",
                    db_strerror(ret));
        goto err;
    }

    /*************************************************************/
    /*************************************************************/
    /*************  Set the cache size here **********************/
    /*************************************************************/
    /*************************************************************/

    ret = dbp->set_cachesize(dbp,
                              0,     /* 0 gigabytes */
              10 * 1024 * 1024,     /* 10 megabytes */
                             1);     /* Create 1 cache. All memory will
                                      * be allocated contiguously. */
    if (ret != 0) {
        dbp->err(dbp, ret, "Database open failed");
        goto err;
    }


    db_flags = DB_CREATE;       /* If it doesn't exist, create it */
    ret = dbp->open(dbp,        /* Pointer to the database */
                    NULL,       /* Txn pointer */
                    NULL,       /* File name is not specified on
                                 * purpose */
                    db_name,    /* Logical db name */
                    DB_BTREE,   /* Database type (using btree) */
                    db_flags,   /* Open flags */
```

```
                    0);         /* File mode. Using defaults */
    if (ret != 0) {
        dbp->err(dbp, ret, "Database open failed");
        goto err;
    }

err:
    /* Close the database */
    if (dbp != NULL) {
        ret_c = dbp->close(dbp, 0);
        if (ret_c != 0) {
            fprintf(stderr, "%s database close failed.\n",
                 db_strerror(ret_c));
            ret = ret_c
        }
    }
```

Specifying a Cache Size using the Environment Handle

To select a cache size using the environment handle, use the `ENV->set_cachesize()` method. Note that you cannot use this method after the environment has been opened.

The following code fragment creates an environment handle, sets the cache size to 10 MB and then opens the environment: Once opened, you can use the environment when you open your database(s). This means all your databases will use the same cache.

```
#include "db.h"

...

    int ret, ret_c;
    u_int32_t env_flags;  /* For open flags */
    DB_ENV *envp;         /* Environment handle */

...

   /* Initialize the ENV handle */
    ret = db_env_create(&envp, 0);
    if (ret != 0) {
        fprintf(stderr, "Error creating environment handle: %s\n",
                    db_strerror(ret));
        goto err;
    }

    /*****************************************************************/
    /*****************************************************************/
    /*************  Set the cache size here **************************/
    /*****************************************************************/
    /*****************************************************************/
```

```
    ret =
        envp->set_cachesize(envp,
                                0,    /* 0 gigabytes */
                10 * 1024 * 1024,    /* 10 megabytes */
                                1);    /* Create 1 cache. All memory will
                                        * be allocated contiguously. */
    if (ret != 0) {
        envp->err(envp, ret, "Environment open failed");
        goto err;
    }

    /*
     * Environment flags. These are for a non-threaded
     * in-memory application.
     */
    env_flags =
      DB_CREATE     |  /* Create the environment if it does not exist */
      DB_INIT_LOCK  |  /* Initialize the locking subsystem */
      DB_INIT_LOG   |  /* Initialize the logging subsystem */
      DB_INIT_TXN   |  /* Initialize the transactional subsystem. This
                        * also turns on logging. */
      DB_INIT_MPOOL |  /* Initialize the memory pool (in-memory cache) */
      DB_PRIVATE    |  /* Region files are not backed by the filesystem.
                        * Instead, they are backed by heap memory.  */

   /*
    * Now open the environment. Notice that we do not provide a location
    * for the environment's home directory. This is required for an
    * in-memory only application.
    */
    ret = envp->open(envp, NULL, env_flags, 0);
    if (ret != 0) {
        fprintf(stderr, "Error opening environment: %s\n",
            db_strerror(ret));
        goto err;
    }

err:
    /* Close the environment */
    if (envp != NULL) {
        ret_c = envp->close(envp, 0);
        if (ret_c != 0) {
            fprintf(stderr, "environment close failed: %s\n",
                 db_strerror(ret_c));
            ret = ret_c
        }
    }
```

Keeping Temporary Overflow Pages in Memory

Normally, when a database is opened, a temporary file is opened on-disk to back the database. This file is used if the database grows so large that it fills the entire cache. At that time, database pages that do not fit into the in-memory cache file are written temporarily to this file.

For disk-less systems, you should configure your databases so that this temporary file is not created. When you do this, any attempt to create new database pages once the cache is full will fail.

You configure this option on a per-database handle basis. That means you must configure this for every in-memory database that your application uses.

To set this option, obtain the `DB_MPOOLFILE` field from you `DB` and then configure `DB_MPOOL_NOFILE` using the `DB_MPOOLFILE->set_flags()` method.

For example:

```
#include "db.h"

...

    int ret, ret_c;
    u_int32_t env_flags;  /* For open flags */
    DB_ENV *envp;         /* Environment handle */

...

    /*
     * Configure the cache file. This can be done
     * at any point in the application's life once the
     * DB handle has been created.
     */
    mpf = dbp->get_mpf(dbp);
    ret = mpf->set_flags(mpf, DB_MPOOL_NOFILE, 1);

    if (ret != 0) {
        fprintf(stderr,
         "Attempt failed to configure for no backing of temp files: %s\n",
                    db_strerror(ret));
        goto err;
    }
```

Keeping Logs in Memory

DB logs describe the write activity that has occurred in your application. For a purely in-memory application, logs should be used only if you wish to transaction-protect your database writes as logs are required by the DB transactional subsystem.

Note that transactions provide a number of guarantees. One of these is not interesting to a purely in-memory application (data durability). However, other transaction guarantees such as isolation and atomicity might be of interest to your application.

If this is the case for your application, then you must configure your logs to be kept entirely in-memory. You do this by setting a configuration option that prevents DB from writing log data to disk. Do this by setting the `DB_LOG_IN_MEMORY` flag using the `DB_ENV->log_set_config()` method.

In addition, you must configure your log buffer size so that it is capable of holding all log information that can accumulate during your longest running transaction. That is, make sure the in-memory log buffer is large enough that no transaction will ever span the entire buffer. Also, avoid a state where the in-memory buffer is full and no space can be freed because a transaction that started the first log "file" is still active.

How much log buffer space is required is a function of the number of transactions you have running concurrently, how long they last, and how much write activity occurs within them. When in-memory logging is configured, the default log buffer space is 1 MB.

You set your log buffer space using the `DB_ENV->set_lg_bsize()`.

For example, the following code fragment configure in-memory log usage, and it configures the log buffer size to 10 MB:

```
#include "db.h"

...

    int ret, ret_c;
    u_int32_t env_flags;  /* For open flags */
    DB_ENV *envp;         /* Environment handle */

...

   /* Initialize the ENV handle */
    ret = db_env_create(&envp, 0);
    if (ret != 0) {
        fprintf(stderr, "Error creating environment handle: %s\n",
                    db_strerror(ret));
        goto err;
    }

    /*
     * Environment flags. These are for a non-threaded
     * in-memory application.
     */
    env_flags =
      DB_CREATE     |  /* Create the environment if it does not exist */
      DB_INIT_LOCK  |  /* Initialize the locking subsystem */
      DB_INIT_LOG   |  /* Initialize the logging subsystem */
      DB_INIT_TXN   |  /* Initialize the transactional subsystem. This
```

```
                         * also turns on logging. */
      DB_INIT_MPOOL |  /* Initialize the memory pool (in-memory cache) */
      DB_PRIVATE    |  /* Region files are not backed by the filesystem.
                         * Instead, they are backed by heap memory.  */


    /* Specify in-memory logging */
    ret = envp->log_set_config(envp, DB_LOG_IN_MEMORY, 1);
    if (ret != 0) {
        fprintf(stderr, "Error setting log subsystem to in-memory: %s\n",
            db_strerror(ret));
        goto err;
    }

    /*
     * Specify the size of the in-memory log buffer.
     */
    ret = envp->set_lg_bsize(envp, 10 * 1024 * 1024);
    if (ret != 0) {
        fprintf(stderr, "Error increasing the log buffer size: %s\n",
            db_strerror(ret));
        goto err;
    }

   /*
    * Now open the environment. Notice that we do not provide a location
    * for the environment's home directory. This is required for an
    * in-memory only application.
    */
    ret = envp->open(envp, NULL, env_flags, 0);
    if (ret != 0) {
        fprintf(stderr, "Error opening environment: %s\n",
            db_strerror(ret));
        goto err;
    }

err:
    /* Close the environment */
    if (envp != NULL) {
        ret_c = envp->close(envp, 0);
        if (ret_c != 0) {
            fprintf(stderr, "environment close failed: %s\n",
                 db_strerror(ret_c));
            ret = ret_c
        }
    }
```

In-Memory Replicated Applications

If you are unfamiliar with writing DB replicated applications, or if you are simply uninterested in this topic, you can skip this section. For an introductory description of DB replication, please see the *Berkeley DB Getting Started with Replicated Applications* guide.

In-memory replicated applications can improve transaction throughput by avoiding disk I/O. Network connections are often faster than local synchronous disk writes, so in-memory replicated applications can provide significantly improved performance without entirely sacrificing reliability.

All of your replication participants must be configured in the same way. That is, they all must be configured for in-memory storage of data, or they must all be configured for on-disk storage of data. As a result, in-memory replicated applications can achieve improved throughput performance, but they do so at the cost of reduced durability guarantees. This is because the transaction commit is not backed by stable storage anywhere in the replication group. However, this "commit to the network" does not sacrifice all durability. Because running clients should have in-memory copies of the data, a client can take over in the event of an outage at the master and in this way avoid the loss of data.

There are many internal resources used by the DB replication subsystem which are by default backed by disk. These internal resources help the DB subsystem ensure election accuracy. While a complete description of these resources is beyond the scope of this article, you should know that you can cause all of these resources to be held entirely in-memory. But you do so with some small chance of operational errors in your replicated application.

If you cause a replicated application to keep its internal replication resources in-memory, you run a small risk that elections will fail or be unable to complete. However, calling additional elections should eventually yield a winner.

In addition, there is a slight possibility that the wrong site might win an election, which could result in the loss of data. This can happen if you have a site that is repeatedly crashing and trying to come back up. A site like this might be repeatedly sending out election information, and the repeated messages might confuse other sites. For replication applications that are not in-memory, these extra messages would be ignored by other sites because they would also contain some state information that allows other sites to know which election messages are relevant. But strictly in-memory replicated applications cannot maintain this state information, and so some other sites might become confused. The result might be that the wrong site could be elected master due to the inconsistent information that is available to them.

Note

This is very much a corner case that you probably will never see in your production systems, especially if your sites are all stable and well-behaved.

If an election is won by the wrong site (site A), then some other site (site B) probably has more recent log files than the winner does. But since the wrong site A won the election, site B will sync with the new master. This will cause site B (and therefore, your entire replication group) to lose any log files it contains that are more recent than the files contained by site A.

If you are running a master that is configured to run with internal replication resources in-memory, you should never allow that site to appoint itself master again immediately after crashing or rebooting. Doing so results in a slightly higher risk of your client sites crashing. To determine your next master, you should either hold an election or appoint a different site to be master.

In order to cause a replication site to run entirely in-memory, do all of the things described previously in this document to place all other DB resources in-memory. Then, when configuring replication, specify `DB_REP_CONF_INMEM` to the `DB_ENV->rep_set_config()` method.

Example In-Memory Application

The following brief example illustrates how to open an application that is entirely in-memory. The application opens an environment and a single database, and does this in a way that the database is transaction-protected.

Transactions can be desirable for an in-memory application even though you discard your durability guarantees, because of the other things that transactions offer such as atomicity and isolation.

Notice that the example does nothing other than open and close the environment and database. DB database reads and writes work identically between in-memory-only and durable applications (that is, applications that write database application to durable storage). Consequently, there is no point in illustrating those actions here.

```
/* We assume an ANSI-compatible compiler */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <db.h>


int
main(void)
{
    /* Initialize our handles */
    DB *dbp = NULL;
    DB_ENV *envp = NULL;
    DB_MPOOLFILE *mpf = NULL;

    int ret, ret_t;
    const char *db_name = "in_mem_db1";
    u_int32_t open_flags;

    /* Create the environment */
    ret = db_env_create(&envp, 0);
    if (ret != 0) {
        fprintf(stderr, "Error creating environment handle: %s\n",
            db_strerror(ret));
        goto err;
    }
```

```
open_flags =
  DB_CREATE     |  /* Create the environment if it does not exist */
  DB_INIT_LOCK  |  /* Initialize the locking subsystem */
  DB_INIT_LOG   |  /* Initialize the logging subsystem */
  DB_INIT_MPOOL |  /* Initialize the memory pool (in-memory cache) */
  DB_INIT_TXN   |
  DB_PRIVATE;      /* Region files are not backed by the filesystem.
                    * Instead, they are backed by heap memory.  */

/* Specify in-memory logging */
ret = envp->log_set_config(envp, DB_LOG_IN_MEMORY, 1);
if (ret != 0) {
    fprintf(stderr, "Error setting log subsystem to in-memory: %s\n",
        db_strerror(ret));
    goto err;
}
/*
 * Specify the size of the in-memory log buffer.
 */
ret = envp->set_lg_bsize(envp, 10 * 1024 * 1024);
if (ret != 0) {
    fprintf(stderr, "Error increasing the log buffer size: %s\n",
        db_strerror(ret));
    goto err;
}

/*
 * Specify the size of the in-memory cache.
 */
ret = envp->set_cachesize(envp, 0, 10 * 1024 * 1024, 1);
if (ret != 0) {
    fprintf(stderr, "Error increasing the cache size: %s\n",
        db_strerror(ret));
    goto err;
}

/*
 * Now actually open the environment. Notice that the environment home
 * directory is NULL. This is required for an in-memory only
 * application.
 */
ret = envp->open(envp, NULL, open_flags, 0);
if (ret != 0) {
    fprintf(stderr, "Error opening environment: %s\n",
        db_strerror(ret));
    goto err;
}
```

```
   /* Initialize the DB handle */
    ret = db_create(&dbp, envp, 0);
    if (ret != 0) {
         envp->err(envp, ret,
                "Attempt to create db handle failed.");
        goto err;
    }


    /*
     * Set the database open flags. Autocommit is used because we are
     * transactional.
     */
    open_flags = DB_CREATE | DB_AUTO_COMMIT;
    ret = dbp->open(dbp,         /* Pointer to the database */
             NULL,        /* Txn pointer */
             NULL,        /* File name -- Must be NULL for inmemory! */
             db_name,     /* Logical db name */
             DB_BTREE,    /* Database type (using btree) */
             open_flags,  /* Open flags */
             0);          /* File mode. Using defaults */

    if (ret != 0) {
         envp->err(envp, ret,
                "Attempt to open db failed.");
        goto err;
    }

    /* Configure the cache file */
    mpf = dbp->get_mpf(dbp);
    ret = mpf->set_flags(mpf, DB_MPOOL_NOFILE, 1);

    if (ret != 0) {
         envp->err(envp, ret,
            "Attempt failed to configure for no backing of temp files.");
        goto err;
    }

err:
    /* Close our database handle, if it was opened. */
    if (dbp != NULL) {
        ret_t = dbp->close(dbp, 0);
        if (ret_t != 0) {
            fprintf(stderr, "%s database close failed.\n",
                db_strerror(ret_t));
            ret = ret_t;
        }
    }
```

```
    /* Close our environment, if it was opened. */
    if (envp != NULL) {
        ret_t = envp->close(envp, 0);
        if (ret_t != 0) {
            fprintf(stderr, "environment close failed: %s\n",
                db_strerror(ret_t));
                ret = ret_t;
        }
    }

    /* Final status message and return. */
    printf("I'm all done.\n");
    return (ret == 0 ? EXIT_SUCCESS : EXIT_FAILURE);
}
```