Базы данных

Лекция 12 – Berkeley DB

Преподаватель: Поденок Леонид Петрович, 505a-5

+375 17 293 8039 (505a-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2023

2023.11.10

Оглавление

Что такое база данных Berkeley DB?
Это то же самое, что таблица реляционной базы данных (SQL)?

ORACLE: Да, концептуально база данных Berkeley DB представляет собой отдельную таблицу реляционной базы данных. Также вы можете рассматривать каждую пару ключ/значение как одну строку в таблице, где столбцы представляют собой данные, закодированные приложением либо в ключе, либо в значении.

«Starting with release 12.1.6.0.x, we have moved under the standard AGPL license.»

Стандартная общественная лицензия GNU Афферо – модифицированная версия обычной GNU GPL версии 3. В ней добавлено одно требование: если вы выполняете измененную программу на сервере и даете другим пользователям общаться на нем с этой программой, ваш сервер должен также позволять им получить исходный текст, соответствующий той модифицированной версии программы, которая выполняется на сервере.

Где взять BDB?

1) В дистрибутивах Linux (5.3.28).

libdb	The Berkeley DB database library for C
libdb-cxx	The Berkeley DB database library for C++
libdb-utils	Command line tools for managing Berkeley DB databases
libdb-devel	C development files for the Berkeley DB library
libdb-cxx-devel	C++ development files for the Berkeley DB library
libdb-devel-static	
libdb-devel-doc	
libdb-sql	Development files for using the Berkeley DB with sql
libdb-tcl	The Berkeley DB database library for C

2) Собрать из исходников (https://edelivery.oracle.com/akam/otn/berkeley-db)

> **401 Oracle Export Error**
> In compliance with U.S. and applicable Export laws we are unable to process your request to https://edelivery.oracle.com/akam/otn/berkeley-db/db-18.1.32.tar.gz. Please contact RPLS-Ops_ww@oracle.com if you believe you are receiving this notice in error.

ftp://student@lsi.bas-net.by/software/BerkeleyDB
db-18.1.32.tar.g
db-18.1.40.tar.gz – Ошибка при установке. Решение: скопировать из 32 недостающее в дистрибутив.

Устанавливается по умолчанию в /usr/local/BerkeleyDB.18.1
./bin
./docs
./include
./lib

Соответственно, нужно самостоятельно прописывать пути к библиотекам и утилитам при использовании BDB.

Методы доступа

1) Приложения Berkeley DB могут выбирать структуру хранения, которая лучше всего подходит для приложения. Berkeley DB поддерживает:

- хеш-таблицы;
- B-деревья;
- простое хранилище на основе номеров записей;
- постоянные очереди (persistent queue).

Программисты могут создавать таблицы, используя любую из этих структур хранения, и могут смешивать операции с разными типами таблиц в одном приложении.

2) Записи в Berkeley DB представляют собой пары (ключ, значение).

3) Berkeley DB никогда не работает со значением записи. Значения — это просто полезная нагрузка, которая хранится вместе с ключами и надежно доставляется обратно в приложение по запросу.

4) И ключи, и значения могут быть произвольными строками байтов фиксированной или переменной длины. Соответственно, программисты могут помещать в базу структуры данных на любом языке программирования, не занимаясь преобразованием их в/из записей чужого формата.

5) Приложению необходимо заранее знать структуру ключа и структуру значения — приложение не может запросить эту информацию у Berkeley DB, потому что Berkeley DB ее не знает.

6) Ключи, и значения могут иметь длину до четырех гигабайт (32 бита).

Метод доступа можно выбрать только при создании базы данных. После этого выбора фактическое использование API, как правило, идентично для всех методов доступа. То есть, хотя существуют некоторые исключения, в принципе, взаимодействие с библиотекой одинаково, независимо от того, какой метод доступа выбран.

Метод доступа, который предстоит выбрать, зависит, во-первых, от того, что необходимо использовать в качестве ключа, а во-вторых, от производительности, которая обеспечивается для данного метода доступа.

Btree

Данные хранятся в отсортированной сбалансированной древовидной структуре.

И ключ, и данные для записей BTree могут быть произвольно сложными.

Они могут содержать отдельные значения, такие как целое число или строка, или сложные типы, такие как структура. Кроме того есть возможность (это не поведение по умолчанию) двум записям использовать ключи, сравниваемые как равные.

Когда это происходит, записи считаются дубликатами друг друга.

Hash

Данные хранятся в расширенной линейной хеш-таблице.

Как и в случае с BTree, ключ и данные, используемые для хеш-записей, могут представлять собой произвольно сложные данные. Также, как и в BTree, опционально поддерживаются дублирующиеся записи.

Queue

Данные хранятся в очереди в виде записей фиксированной длины.

Каждая запись в качестве своего ключа использует логический номер записи. Этот метод доступа предназначен для быстрой вставки в хвост очереди и имеет специальную операцию, удаляющую и возвращающую запись из головы очереди.

Этот метод доступа необычен тем, что предусматривает блокировку на уровне записи. Это может улучшить производительность приложений, которым требуется одновременный доступ к очереди.

Recno

Данные хранятся в записях фиксированной или переменной длины.

Как и в **Queue**, в записях типа **Recno** в качестве ключей используются номера логических записей.

Выбор методов доступа

Чтобы выбрать метод доступа, необходимо сначала решить, что использовать в качестве ключа для записей базы данных.

1) если необходимо использовать произвольные данные (включая строки), следует использовать **`BTree`** или **`Hash`**.

2) если нужно использовать логические номера записей (по существу, это целые числа), следует использовать **`Queue`** или **`Recno`**.

После этого нужно выбрать между **`BTree`** или **`Hash`**, **`Queue`** или **`Recno`**.

Выбор между BTree и Hash

Для небольших рабочих наборов данных, которые полностью помещаются в память, никакой разницы между BTree и Hash нет — оба будут работать одинаково хорошо.

В такой ситуации рекомендуется использовать BTree, хотя бы по той причине, что большинство приложений БД используют именно BTree.

Основной отправной точкой выбора является именно ***рабочий набор данных***, а не весь набор данных. Многие приложения поддерживают большие объемы информации, но им требуется доступ только к небольшой части этих данных. Поэтому нужно принимать во внимание те данные, которые будут регулярно использоваться, а не общую массу всех данных, которыми управляет приложение.

Объемы данных имеют тенденцию неуправляемо расти и в конце концов могут не умещаться в памяти целиком, поэтому нужно быть более осторожным при выборе метода доступа.

В частности, имеет смысл выбирать:

BTree, если ключи имеют определенную ссылочную локальность.

То есть, если предполагается их сортировка и можно ожидать, что запрос для данного ключа, вероятно, будет сопровождаться запросом для одного из его соседей.

Hash, если набор данных очень большой.

Для любого заданного метода доступа БД должна поддерживать определенный объем внутренней информации. Однако объем информации, которую БД должна хранить для **`BTree`**, намного больше, чем для **`Hash`**. В результате по мере роста набора данных эта внутренняя информация может доминировать в кэше до такой степени, что для данных приложения остается относительно мало места.

В результате **`BTree`** может быть вынужден выполнять дисковый ввод-вывод гораздо чаще, чем **`Hash`** при том же объеме данных.

Если ваш набор данных станет настолько большим, что БД почти наверняка придется выполнять дисковый ввод-вывод для удовлетворения случайного запроса, тогда **`Hash`** по производительности превзойдет **`BTree`**, потому что у него меньше внутренних записей для обеспечения поиска.

Выбор между Queue и Recno

Queue или **Recno** используются, когда в качестве первичного ключа базы данных приложение хочет использовать номера логических записей.

Логические номера записей — это, по существу, целые числа, которые однозначно идентифицируют запись базы данных. Они могут быть как изменяемыми, так и фиксированными.

Изменяемый номер записи — это номер, который может меняться при сохранении или удалении записей базы данных.

Фиксированные номера логических записей никогда не меняются независимо от выполняемых операций с базой данных.

При выборе между **Queue** и **Recno** следует иметь ввиду:

Queue, если приложение требует высокой степени параллелизма.

Очередь обеспечивает блокировку на уровне записи (в отличие от блокировки на уровне страницы, которую используют другие методы доступа), и это может привести к значительному увеличению производительности для приложений с высокой степенью параллелизма.

Однако, **Queue** поддерживает только записи фиксированной длины. Поэтому, если размер данных, которые необходимо хранить, сильно различается от записи к записи, скорее всего лучше выбрать метод доступа, отличный от **Queue**.

Recno, если нужны изменяемые номера записей.

Окружение (environment)

Окружения обычно не используются приложениями, работающими во встроенных средах, где важен каждый байт. Однако, если приложение требует чего-либо, кроме минимальной функциональности, без них не обойтись.

Окружение — это, по сути, инкапсуляция одной или нескольких баз данных. Сначала открывается окружение, а затем базы данных в этом окружении. При этом базы данных создаются/располагаются в локациях относительно домашнего каталога.

Окружения предлагают очень много функций, которые не может предложить автономная база данных BDB:

1) Файлы с несколькими базами данных

В BDB возможно содержать несколько баз данных в одном физическом файле на диске.

Это желательно для тех приложений, которые открывают несколько баз данных. Однако для того, чтобы в одном физическом файле содержалось более одной базы данных, приложение должно использовать *окружение*.

Поддержка многопоточности и многопроцессности

При использовании окружения такие ресурсы, как кэши в памяти и блокировки, могут совместно использоваться всеми базами данных, открытыми в данном окружении.

Окружение дает возможность использовать подсистемы, предназначенные для предоставления нескольким потокам и/или процессам доступа к базам данных BDB.

Например, можно использовать окружения, чтобы предоставить возможность реализовать параллельное хранилище данных (CDS — Concurrent Data Store), подсистемы блокировок и/или пулы буферов общей памяти.

2) Транзакционная обработка

BDB предлагает транзакционную подсистему, которая обеспечивает полную ACID-защиту базы данных при записи. Для включения транзакционной подсистемы и получения идентификаторов транзакций требуется использовать окружения.

3) Поддержка высокой доступности (репликации)

BDB предлагает подсистему репликации, которая обеспечивает репликацию базы данных с одним мастер-набором данных с несколькими копиями реплицируемых данных только для чтения. Чтобы подключить эту подсистему и впоследствии ею управлять, требуется использовать окружения.

Подсистема протоколирования (регистрации)

BDB предлагает ведение журнала с опережающей записью тем приложениям, которые хотят получить высокую степень восстанавливаемости в случае сбоя как приложения, так и системы.

После подключения подсистема ведения журнала позволяет приложению выполнять два вида восстановления, «обычное» и «катастрофическое», с использованием информации, содержащейся в файлах журнала.

Возвращение ошибок

1) Интерфейсы БД всегда возвращают значение 0 в случае успеха.

2) Если операция по какой-либо причине не удалась, возвращаемое значение будет ненулевым.

3) Если произошла системная ошибка (например, в БД закончилось место на диске, или было отказано в доступе к файлу, или для одного из интерфейсов был указан недопустимый аргумент), БД возвращает значение **`errno`**.

Все возможные значения **`errno`** больше 0.

4) Если операция завершилась не из-за системной ошибки, но и не была успешной, БД возвращает специальное значение ошибки.

Например, при попытке получить данные из базы, а искомая запись не существует, DB вернет **`DB_NOTFOUND`**, специальное значение ошибки, означающее, что запрошенный ключ не отображается (map) в базе данных.

Все возможные значения таких специальных ошибок меньше 0.

DB предлагает программную поддержку для отображения возвращаемых значений ошибок.

1) Функция **`db_strerror()`** возвращает указатель на сообщение об ошибке, соответствующее возвращенному коду ошибки БД, аналогично C-функции **`strerror()`**. Она может обрабатывать кроме возвращаемых значений, специфичных для БД, также возвраты и системных ошибок.

2) Есть две функции ошибок: **`DB->err()`** и **`DB->errx()`**. Эти функции работают так же, как C-функция **`printf()`**, принимая строку формата в стиле **`printf()`** и список аргументов. Также она может добавить стандартную строку ошибки к сообщению, составленному из строки формата и других аргументов.

Документация не содержит man'ов.

Базы данных

В Berkeley DB база данных представляет собой набор записей.
Записи, в свою очередь, состоят из пар ключ/данные.
Концептуально базу данных можно представить как содержащую таблицу с двумя столбцами, где столбец 1 содержит ключ, а столбец 2 содержит данные.
И ключ, и данные управляются с помощью структур **DBT** (db.h).

Использование базы данных БД включает в себя:
- добавление;
- получение;
- удаление

записей базы данных.

Открытие базы данных

Чтобы открыть базу данных, сначала необходимо использовать функцию **db_create()**, которая инициализирует дескриптор БД.
Для открытия базы данных после получения дескриптора нужно использовать его метод **open()**.
По умолчанию, если базы данных еще не существуют, они не создаются. Это поведение можно переопределить, указав флаг **DB_CREATE** в методе[1] **open()**.

```
int db_create(DB **, DB_ENV *, u_int32_t);
```

1) >120 методов

Пример:

```
#include <db.h>

...

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

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

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

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

Закрытие баз данных

Для закрытия используется метод **DB->close()**.

Прежде чем закрыть базу данных, всегда необходимо убедиться, что все обращения к базе данных завершены.

Закрытие базы данных делает ее непригодной для использования до тех пор, пока она не будет открыта снова. Перед закрытием базы данных рекомендуется закрыть все открытые *курсоры*.

Курсоры — это механизм, с помощью которого можно перебирать записи в базе данных.

С их использованием можно получать, добавлять и удалять записи базы данных.

Если база данных допускает дублирование записей, то курсоры — это самый простой способ получить доступ ко всем записям с данным ключом, кроме первой.

Активные курсоры во время закрытия базы данных могут привести к неожиданным результатам, особенно если какой-либо из этих курсоров выполняет запись в базу данных.

Когда закрывается последний открытый дескриптор базы данных, по умолчанию ее кэш сбрасывается на диск. Это означает, что любая информация, которая была в кэше изменена, при закрытии последнего дескриптора гарантированно будет записана на диск.

Эту операцию можно выполнить вручную с помощью метода **DB->sync()**, но для обычных операций завершения работы в этом нет необходимости.

Пример:

```
#include <db.h>
...
if (dbp != NULL) {
    dbp->close(dbp, 0);
}
```

Флаги открытия базы данных

Флаги приведены здесь не все, а только используемые в однопоточных приложениях.
Чтобы при вызове **DB->open()** указать более одного флага , нужно использовать побитовое ИЛИ:

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

DB_CREATE

Если база данных в настоящее время не существует, она будет создана.
По умолчанию попытка открытия несуществующей базы данных завершается ошибкой.

DB_EXCL

Создание базы данных для использования в эксклюзивном режиме.
Если база данных уже существует, произойдет сбой. Этот флаг имеет смысл только при использовании вместе с **DB_CREATE**, чтобы гарантированно создать новую базу данных.

DB_RDONLY

База данных открывается только для операций чтения.
Любая последующая операция записи в базу данных вызывает сбой.

DB_TRUNCATE

Файл на диске, содержащий базу данных, будет физически обрезан (очищен).
Будут удалены все базы данных, физически содержащиеся в файле.

Функции сообщения об ошибках

БД предлагает несколько полезных методов.

```
set_errcall( )
```

Определяет функцию, которая вызывается, когда БД выдает сообщение об ошибке.
Префикс ошибки и сообщение передаются этому обратному вызову.
Приложение должно самостоятельно отображать эту информацию.

```
set_errfile( )
```

Устанавливает **`FILE *`** (стандартная C-библиотека) для отображения сообщений об ошибках, выдаваемых библиотекой DB.

```
set_errpfx( )
```

Задает префикс, используемый для любых сообщений об ошибках, выдаваемых библиотекой БД.

```
err( )
```

Выдает сообщение об ошибке. Сообщение об ошибке отправляется функции обратного вызова, определенной вызовом **`set_errcall( )`**. Если данный вызов не использовался, то сообщение об ошибке отправляется в файл, определенный функцией **`set_errfile( )`**. Если ни один из этих методов не использовался, то сообщение об ошибке отправляется в **`stderr`**.

Сообщение об ошибке состоит из строки префикса, определенного вызовом **`set_errpfx( )`**, необязательного сообщения, отформатированного в стиле **`printf( )`**, сообщения об ошибке и символа новой строки.

```
errx( )
```

Ведет себя идентично **err( )**, за исключением того, что текст сообщения, связанный со значением ошибки, к строке ошибки не добавляется.

Помимо этих для информационных сообщений существуют и другие вызовы.

```
set_msgcall( )
set_msgfile( )
set_msgpfx( )
msg( )
```

Также можно использовать функцию **db_strerror( )** для прямого возврата строки ошибки, соответствующей конкретному номеру ошибки.

Чтобы отправить все сообщения об ошибках данного дескриптора базы данных в функцию обратного вызова для специфической обработки, сначала следует создать функцию обратного вызова:

```
void error_handler(const DB_ENV *dbenv,
                   const char   *error_prefix,
                   const char   *msg) {
    ... // code to handle the error prefix and error message.
}
```

Затем ее регистрируем:

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

DB *dbp;
int ret;

ret = db_create(&dbp, NULL, 0);
if (ret != 0) {
        fprintf(stderr, "%s: %s\n", "my_program", db_strerror(ret));
        return(ret);
}

dbp->set_errcall(dbp, error_handler);     // регистрация ф-ции обратного вызова
dbp->set_errpfx(dbp, "example_program");  // префикс
```

Выдаем сообщение об ошибке открытия :

```
ret = dbp->open(dbp,       // DB structure pointer
                NULL,      // Transaction pointer
                "mydb.db", // On-disk file that holds the database
                NULL,      // Optional logical database name
                DB_BTREE,  // Database access method
                DB_CREATE, // Open flags
                0);        // File mode (using defaults)
if (ret != 0) {
    dbp->err(dbp, ret, "Database open failed: %s", "mydb.db");
    return(ret);
}
```

Управление базами данных в окружениях

Окружения часто используются для широкого класса приложений БД.

Окружение — это инкапсуляция одной или нескольких баз данных.

Окружения предлагают очень много функций, которые не может предложить автономная база данных БД:

- файлы с несколькими базами данных;
- поддержка многопоточности и многопроцессности;
- транзакционная обработка;
- поддержка высокой доступности (репликация);
- подсистема протоколирования (регистрации).

Чтобы использовать окружение, необходимо сначала создать дескриптор окружения, а затем его открыть.

При открытии необходимо определить каталог, в котором находится окружение.

Этот каталог должен существовать до открытия.

Во время открытия можно определить некоторые свойства окружения, например, можно ли создать окружение, если оно еще не существует.

Также при открытии окружения необходимо инициализировать в памяти кэш.

Пример

```
#include <db.h>
...
DB_ENV    *myEnv;         /* структура-дескриптор окружения */
DB        *dbp;           /* структура-дескриптор базы данных */
u_int32_t  db_flags;      /* флаги открытия базы данных */
u_int32_t  env_flags;     /* флаги открытия окружения  */
int        ret;           /* статус выполнения функции */

// Создаем дескриптор объекта окружения и инициализируем его.
ret = db_env_create(&myEnv, 0);
if (ret != 0) {
    fprintf(stderr, "Error creating env handle: %s\n", db_strerror(ret));
    return -1;
}

// Флаги окружения
env_flags = DB_CREATE |    // If the environment does not exist, create it.
            DB_INIT_MPOOL; // Initialize the in-memory cache.
// Открываем окружение
ret = myEnv->open(myEnv,                    /* DB_ENV ptr */
                  "/export1/testEnv",       /* env home directory */
                  env_flags,                /* Open flags */
                  0);                       /* File mode (default) */
if (ret != 0) {
    fprintf(stderr, "Environment open failed: %s", db_strerror(ret));
    return -1;
}
```

Когда окружение открыто, можно открывать в нем базы данных.

По умолчанию базы данных хранятся в домашнем каталоге окружения или, если указан какой-либо путь в имени файла базы данных, относительно этого каталога:

```
// Инициализируем DB структуру.
// Передаем указатель на окружение, где будет открываться эта DB.
ret = db_create(&dbp, myEnv, 0);
if (ret != 0) {
  /* Error handling goes here */
}

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

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

После завершения работы с окружением, его необходимо закрыть.
Перед закрытием рекомендуется закрыть все открытые базы данных.

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

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

Пример базы данных

Создается несколько приложений, загружающих и извлекающих данные инвентаризации из баз данных BDB.

Пример 2.1 Структура stock_db

Сначала создается структура, которая будет использоваться для хранения всех указателей и имен баз данных:

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

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

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

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

Пример 2.2. Вспомогательные функции stock_db

Нужны некоторые служебные функции, которые используются, чтобы убедиться, что структура **stock_db** находится в нормальном состоянии перед ее использованием.

Одна из функций представляет собой простую функцию, которая инициализирует все указатели структуры полезным значением по умолчанию.

Вторая используется для указания общего пути ко всем именам баз данных, чтобы можно было явно указать, где должны находиться все файлы базы данных.

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

/* Initializes the STOCK_DBS struct.*/
void initialize_stockdbs(STOCK_DBS *my_stock) {

    my_stock->db_home_dir       = DEFAULT_HOMEDIR;
    my_stock->inventory_dbp     = NULL;
    my_stock->vendor_dbp        = NULL;

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

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

    size_t size;

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

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

Пример 2.3 Функция open_database()

Открывается несколько баз данных с одинаковыми флагами и настройками отчетов об ошибках. Для этого создается функция, которая выполняет эту операцию:

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

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

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

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

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

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

    return (0);
}
```

Example 2.4 The databases_setup() Function

С использованием **open_database( )** создается функция, которая откроет все базы данных.

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

    /* Open the vendor database */
    ret = open_database(&(my_stock->vendor_dbp),
                        my_stock->vendor_db_name,
                        program_name, error_file_pointer);
    if (ret != 0) // обработка ошибок в open_database( ) => просто возврат кода ошибки
        return (ret);

    /* Open the inventory database */
    ret = open_database(&(my_stock->inventory_dbp),
                        my_stock->inventory_db_name,
                        program_name, error_file_pointer);
    if (ret != 0)
        return (ret);

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

Функция databases_close()

Полезно иметь функцию, которая может закрыть для все базы данных:

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

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

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

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