Базы данных

Лекция 04 – Berkeley DB. Записи.

Преподаватель: Поденок Леонид Петрович, 505а-5

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

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

Кафедра ЭВМ, 2024

2024.09.30/2024.10.14

Оглавление

Записи базы данных

Записи БД состоят из двух частей — ключа и некоторых данных.

И ключ, и соответствующие ему данные инкапсулируются в структуры типа DBT.

Поэтому для доступа к записи БД нужны две структуры — одна для ключа и другая для данных.

Структуры DBT предоставляют поле **`void *`**, которое используется для ссылки на данные, и поле, определяющее длину данных. Поэтому их можно использовать для хранения чего угодно, от простых примитивных данных до сложных структур, если информация, которую необходимо охранить, находится в одном непрерывном блоке памяти.

Использование записей базы данных

Каждая запись базы данных состоит из двух структур DBT — одна для ключа, а другая для данных.

Чтобы сохранить запись базы данных, если ключ и/или данные являются примитивными данными (**`int`**, **`float`** и т. д.), или если ключ и/или данные содержат массивы, достаточно указать на место в памяти, где это данные находятся, и их длину.

```
struct __db_dbt {
    void      *data; // ключ/данные
    u_int32_t  size; // размер ключа/данных

    u_int32_t  ulen; // RO: размер пользовательского буфера
    u_int32_t  dlen; // RO: длина частичной записи для методов get/put
    u_int32_t  doff; // RO: смещениечастичной записи для методов get/put

    void      *app_data; // данные для пользовательских функций обратного вызова
    u_int32_t  flags;    // ФЛАГИ
};
```

Все поля структуры DBT перед первым ее использованием, которые не заданы явно, должны быть инициализированы нулевыми байтами. Это может быть сделано объявлением структуры DBT внешней или статической, также можно вызвать функцию библиотеки C **memset(3)**.

По умолчанию ожидается, что член структуры **flags** будет установлен в 0. В этом случае, когда приложение предоставляет Berkeley DB ключ или элемент данных для сохранения в базе данных, Berkeley DB ожидает, что элемент структуры данных будет указывать на строку байт размером **size**.

При возврате ключа/элемента данных в приложение Berkeley DB сохранит в члене **data** указатель на строку байтов размером **size**, при этом память, на которую ссылается **data**, будет выделена и управляться Berkeley DB.

Важно!!! Использование флагов по умолчанию для возвращаемых DBT совместимо только с однопоточным использованием Berkeley DB.

Элементы структуры DBT определяются следующим образом:

void *data;

Указатель на строку байт.

u_int32_t size;

Длина данных в байтах.

u_int32_t flags;

Параметр **flags** устанавливается либо в 0, либо путем побитового включающего ИЛИ одного или нескольких из следующих значений:

DB_DBT_READONLY — когда этот флаг установлен, Berkeley DB не будет записывать в DBT.

Флаг может устанавливаться для значений ключей в тех случаях, когда ключ представляет собой статическую строку, в которую невозможно писать, а Berkeley DB может попытаться ее обновить.

DB_DBT_MALLOC — если установлен этот флаг, Berkeley DB выделит память для возвращаемого ключа или элемента данных, используя **malloc(3)** (или указанную пользователем функцию **malloc**) и вернет указатель на нее в поле **data** ключа или элемента данных.

DB_DBT_REALLOC — если установлен этот флаг, Berkeley DB выделит память для возвращаемого ключа или элемента данных с помощью **realloc(3)** (или указанной пользователем функции **realloc**) и вернет указатель на нее в поле **data** ключа или данных.

Поскольку за любую выделенную память отвечает вызывающее приложение, вызывающая сторона должна определить, была ли выделена память, используя возвращаемое значение поля данных.

Разница между DB_DBT_MALLOC и DB_DBT_REALLOC заключается в том, что в последнем случае будет вызывать **realloc(3)** вместо **malloc(3)**, поэтому выделенная память будет увеличиваться по мере необходимости вместо того, чтобы приложению пришлось выполнять повторяющиеся вызовы **free/malloc**.

DB_DBT_USERMEM — поле **data** ключа или данных должно ссылаться на область памяти длиной не менее **ulen** байт. Если длина запрошенного элемента меньше или равна этому количеству байтов, элемент копируется в память, на которую ссылается **data**.

В противном случае в поле **size** устанавливается длина, необходимая для запрошенного элемента, и возвращается ошибка **DB_BUFFER_SMALL**. Приложения могут определить длину записи, установив в поле **ulen** значение 0 и проверив размер возвращаемого значения в поле **size**. Поле **data** при этом не изменяется.

u_int32_t ulen;

Используется с флагом **DB_DBT_USERMEM** и указывает размер пользовательского буфера, на который ссылается член **data**, в байтах.

Указание более одного из **DB_DBT_MALLOC**, **DB_DBT_REAALLOC** и **DB_DBT_USERMEM** является ошибкой.

DB_DBT_PARTIAL — выполнить частичное извлечение или сохранение элемента.

Если вызывающее приложение выполняет операцию **GET**, возвращается **dlen** байт, начиная со смещения **doff** от начала полученной записи данных, как если бы они составляли всю запись.

u_int32_t dlen;

Длина частичной записи, читаемой или записываемой приложением, в байтах. Дополнительная информация во флаге **DB_DBT_PARTIAL**.

u_int32_t doff;

Смещение частичной записи, читаемой или записываемой приложением, в байтах. Дополнительная информация во флаге **DB_DBT_PARTIAL**.

Если какие-либо или все указанные байты в записи не существуют, получение завершается успешно, при этом возвращаются все существующие байты.

Например, если часть данных извлекаемой записи составляла 100 байт, а частичное извлечение было выполнено с использованием DBT, имеющего поле **dlen**, равное 20, и поле **doff**, равное 85, вызов **get** будет успешным, поле данных будет ссылаться на последние 15 байт записи, а поле размера будет установлено на 15.

Если вызывающее приложение выполняет операцию **PUT**, **dlen** байт, начиная со смещения **doff** от начала записи указанной ключом, заменяются данными, указанными элементами структуры **data** и **size**.

Если **dlen** меньше **size**, запись вырастет в размере.

Если **dlen** больше **size**, запись будет обрезана.

Если указанные байты не существуют, запись будет расширена с использованием нулевых байтов по мере необходимости, а вызов **PUT** завершится успешно.

Попытка частичного размещения с использованием метода **DB->put()** в базе данных, поддерживающей повторяющиеся записи, является ошибкой.

Частичное размещение в базах данных, поддерживающих повторяющиеся записи, должно выполняться с использованием метода **DBcursor->put()**.

Попытка частичного размещения с разными значениями **dlen** и **size** в базах данных Queue или Recno с записями фиксированной длины является ошибкой.

Например, если часть данных извлеченной записи составляла 100 байт, а частичная запись была выполнена с использованием DBT, имеющего поле **dlen** 20, поле **doff** 85 и поле **size** 30, результирующая запись будет иметь 115 байт, где последние 30 байтов будут указанными в вызове **PUT**.

Этот флаг игнорируется при использовании с параметром **pkey** в методе **DB->pget()** или в методе **Dbcursor->pget()**.

DB_DBT_APPMALLOC — после выполнения предоставляемой приложением процедуры обратного вызова, вызванной методами **DB->associate()**, либо **DB->set_append_recno()**, поле **data** в DBT может ссылаться на память, выделенную с помощью **malloc(3)** или **realloc(3)**. В этом случае обратный вызов в DBT может установить флаг **DB_DBT_APPMALLOC**, чтобы Berkeley DB вызывала **free(3)** для освобождения памяти, когда она больше не требуется.

Флаг **DB_DBT_APPMALLOC** может быть установлен в любой из структур DBT, чтобы указать, что их поле **data** необходимо освободить.

DB_DBT_MULTIPLE — устанавливается в процедуре обратного вызова создания вторичного ключа, вызванной методом **DB->associate()**, чтобы указать, что с данной парой первичный_ключ/данные должны быть связаны несколько вторичных ключей. Если флаг установлен, поле **size** указывает количество вторичных ключей, а поле **data** относится к массиву из этого количества структур DBT.

Функция **DB->associate()** используется для объявления одной базы данных вторичным индексом для первичной базы данных.

После того как вторичная база данных будет «связана» с первичной базой данных, все обновления первичной базы данных будут автоматически отражаться во вторичной базе данных, и все операции чтения из вторичной базы данных будут возвращать соответствующие данные из первичной базы данных.

void *app_data;

Необязательное поле, которое можно использовать для передачи информации через вызовы API Berkeley DB в определяемые пользователем функции обратного вызова. Например, к этому полю можно получить доступ для передачи определяемого пользователем содержимого при реализации обратного вызова, используемого **DB->set_dup_compare()**.

DB_DBT_EXT_FILE — этот флаг устанавливается для DBT, используемой для части данных записи, чтобы указать, что DBT хранит данные внешнего файла. Если этот флаг установлен, и если база данных поддерживает внешние файлы, то данные, содержащиеся в этом DBT, будут сохраняться как внешний файл.

Пример

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

...

DBT key, data;
float  money       = 122.45;
char  *description = "Счет за продукты";

// Очистка DBTs перед использованием
memset(&key,  0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));

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

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

Чтобы получить запись, иногда необходимо указать куда следует ее возвращать.

В следующем примере BDB не позволяется выделять память для извлечения денежного значения. Причина в том, что некоторые системы могут требовать, чтобы значения с плавающей запятой имели определенное выравнивание, а память, возвращаемая BDB, может быть неправильно выровнена (такая же проблема может существовать для структур в некоторых системах).

Поэтому указывается флаг DB_DBT_USERMEM – использовать память программы вместо памяти BDB.

В этом случае в поле **ulen** необходимо указать, сколько пользовательской памяти доступно.

```
#include <db.h>
#include <string.h>
...
float money;
DBT   key, data;
char *description;

// Очистка DBTs перед использованием
memset(&key,  0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));

key.data  = &money;
key.ulen  = sizeof(float);
key.flags = DB_DBT_USERMEM; // использовать память приложения

// Здесь код для полученния записи из БД.
// money будет установлено в памяти, которую мы указали

description = data.data;
```

Чтение и запись записей базы данных

При чтении и записи записей базы данных надо иметь в виду, что существуют небольшие различия в поведении в зависимости от того, поддерживает ли база данных дубликаты записей.

Две или более записей базы данных считаются дубликатами друг друга, если они имеют один и тот же ключ.

Набор записей, использующих один и тот же ключ, называется набором дубликатов.

В BDB данный ключ сохраняется только один раз для одного набора дубликатов.

По умолчанию базы данных BerkeleyDB не поддерживают повторяющиеся записи.

В тех случаях, когда повторяющиеся записи поддерживаются, для доступа ко всем записям в наборе дубликатов обычно используются ***курсоры***.

BDB предоставляет два основных механизма для хранения и извлечения пар ключ/данные из базы данных:

Самый простой доступ ко всем ***неповторяющимся*** записям в базе данных обеспечивают методы **`DBT->put()`** и **`DBT->get()`**.

Курсоры предоставляют несколько способов размещения и получения записей в/из базы данных.

Запись в базу данных

Записи хранятся в базе данных с использованием любой организации данных, необходимой для выбранного метода доступа.

В некоторых случаях (например, в BTree) записи хранятся в порядке сортировки, который можно определить явно, указав функцию сравнения[1].

После того, как выбран метод доступа, настроены процедуры сортировки (если они есть) и открыта база данных, механизм размещения и получения записей базы данных изменить нельзя.

С точки зрения кодирования простые **put()** и **get()** практически не отличаются независимо от того, какой метод доступа используется.

Для размещения записи в базе данных используется метод **DB->put()**, сохраняющий пары ключ/данные в базе данных.

```
#include <db.h>

int DB->put(DB        *db,     // дескриптор базы данных
            DB_TXN    *txnid,  // дескриптор транзакции
            DBT       *key,    //
            DBT       *data,   //
            u_int32_t  flags); //
```

key, **data** – метод требует, предоставления ключа записи и данных в виде пары структур **DBT**.

Поведение **DB->put()** по умолчанию заключается в:

- вводе новой пары ключ/данные;
- замене пары ключ/данные для любого ранее существующего ключа, если дубликаты запрещены;
- добавлении дублирующего элемента данных, если дубликаты разрешены.

---

1) см. главу «Установка функции сравнения» LK06

Если база данных поддерживает дубликаты, **DB->put()** добавляет новое значение данных в конец набора дубликатов.

Если база данных поддерживает отсортированные дубликаты, новое значение данных вставляется в правильное место в порядке сортировки.

DB->put() возвращает ненулевое значение ошибки в случае неудачи и 0 в случае успеха.

txnid

- если операция является частью транзакции, заданной приложением, параметр **txnid** — это дескриптор транзакции, возвращаемый из **DB_ENV->txn_begin()**;

- если операция является частью группы Concurrent Data Store Berkeley DB, параметр **txnid** — это дескриптор, возвращаемый из **DB_ENV->cdsgroup_begin()**;

- в противном случае **NULL**.

Если дескриптор транзакции не указан, но операция происходит в транзакционной базе данных, операция будет защищена транзакциями неявно.

Также можно указать один или несколько флагов, которые управляют поведением БД при записи в базу данных.

DB_APPEND — добавить пару ключ/данные в конец базы данных.

Флаг DB_APPEND может быть указан только для базы данных Heap[2], Queue или Recno.

Номер, присвоенный записи, возвращается в указанном ключе.

Этот флаг является обязательным для базы данных типа Heap в том случае, если операция **put** приводит к созданию новой записи.

---

2) Метод доступа Heap сохраняет записи в файле кучи. При удалении записей не требуется уплотнение, что позволяет более эффективно использовать пространство, чем при использовании Btree. Метод доступа к куче предназначен для платформ с ограниченным дисковым пространством, особенно если эти системы создают и удаляют большое количество записей.

DB_NODUPDATA — в случае методов доступа Btree и Hash водит новую пару ключ/данные только в том случае, если она еще не присутствует в базе данных.

Флаг DB_NODUPDATA можно указать только в том случае, если база данных настроена на поддержку отсортированных дубликатов.

Флаг DB_NODUPDATA нельзя указывать для методов доступа Queue или Recno.

Если установлен флаг DB_NODUPDATA и пара ключ/данные уже присутствует в базе данных, **DB->put()** вернет ошибку **DB_KEYEXIST**.

DB_NOOVERWRITE — вводит новую пару ключ/данные только в том случае, если ключ еще не появился в базе данных.

Если ключ уже существует в базе данных, вызов **DB->put()** с установленным флагом DB_NOOVERWRITE завершится ошибкой, даже если база данных поддерживает дубликаты.

Если установлен DB_NOOVERWRITE и ключ уже присутствует в базе данных, **DB->put()** вернет ошибку **DB_KEYEXIST**.

Данное обеспечение уникальности ключей применяется только к первичному ключу.

Флаг DB_NOOVERWRITE не влияет на поведение вставок во вторичные базы данных. В частности, использование этого флага не предотвратит вставку записи, которая приведет к созданию дублирующего ключа во вторичной базе данных, допускающей дублирование.

DB_MULTIPLE — массовая операция, размещает несколько элементов данных, используя ключи из массового буфера, на который ссылается параметр **key**, и значения данных из буфера, на который ссылается параметр **data**. Буфера готовятся с помощью впециальных методов, которые различаются для разных организаций доступа.

Успешная массовая операция логически эквивалентна циклу по каждой паре ключ/данные с выполнением **DB->put()** для каждой из них.

Флаг DB_MULTIPLE можно использовать только отдельно или с опцией DB_OVERWRITE_DUP.

DB_MULTIPLE_KEY — массовая операция, размещает несколько элементов данных, используя ключи и данные из буфера, на который ссылается **key**.

DB_OVERWRITE_DUP — игнорировать повторяющиеся записи при перезаписи в базе данных, настроенной для сортировки дубликатов.

Обычно, если база данных настроена для сортировки дубликатов, попытка поместить запись, идентичную записи, уже существующей в базе данных, завершится неудачей. Использование этого флага приводит к тому, что **PUT** выполняется автоматически и без сбоев.

Этот флаг чрезвычайно полезен при выполнении массовых операций размещения (с использованием флагов DB_MULTIPLE или DB_MULTIPLE_KEY).

В зависимости от количества записей, которые записываются в базу данных с помощью массового размещения, может быть нежелательным, чтобы операция завершалась неудачей в случае обнаружения повторяющейся записи.

Использование этого флага вместе с флагами DB_MULTIPLE или DB_MULTIPLE_KEY позволяет завершить массовое размещение, даже если обнаружена повторяющаяся запись.

Этот флаг также полезен, если используется пользовательская функция сравнения, которая сравнивает только часть данных записи. В этом случае две записи могут сравниваться одинаково, хотя на самом деле они не равны. Этот флаг позволяет завершить операцию PUT, даже если пользовательская процедура сравнения утверждает, что две записи равны.

Пример кода

```
#include <db.h>
#include <string.h>
...
char *description = "Grocery bill.";
DBT   key, data;
DB   *my_database;
int   ret;
float money;

// Открытие БД опущено для пущей простоты примера

money = 122.45;

// Очистка DBT перед их использованием
memset(&key,  0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));

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

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

ret = my_database->put(my_database, NULL, &key, &data, DB_NOOVERWRITE);
if (ret == DB_KEYEXIST) {
    my_database->err(my_database, ret,
      "PUT не удался, поскольку key %f уже существует", money);
}
```

Получение записей из базы данных

Для получения записей базы данных используется метод **DB->get()**.

```
#include <db.h>

int DB->get(DB        *db,     // дескриптор базы данных
            DB_TXN    *txnid,  // дескриптор транзакции
            DBT       *key,    // ключ
            DBT       *data,   // данные
            u_int32_t  flags); // флаги

int DB->pget(DB        *db,     // дескриптор базы данных
             DB_TXN    *txnid,  // дескриптор транзакции
             DBT       *key,    // ключ
             DBT       *pkey,   // ключ из первичной базы
             DBT       *data,   // данные
             u_int32_t  flags); // флаги
```

DB->get() извлекает пары ключ/данные из базы данных. Адрес и длина данных, связанных с указанным ключом **key**, возвращаются в структуре, на которую ссылается **data**.

При наличии повторяющихся значений ключа DB->get() вернет первый элемент данных для назначенного ключа.

Дубликаты сортируются:

- в порядке, указанном функцией сортировки дубликатов (если была указана);
- в порядке курсора вставки;
- в порядке вставки. Это поведение по умолчанию.

Для поиска дубликатов требуется использование операций с курсором.

При вызове для базы данных, которая была сделана вторичным индексом с помощью метода **DB->associate()**, **DB->get()** и **DB->pget()** возвращают ключ из вторичного индекса и элемент данных из первичной базы данных.

Кроме того, **DB->pget()** возвращает ключ из первичной базы данных.

В базах данных, которые не являются вторичными индексами, метод DB->pget() всегда будет завершаться ошибкой.

DB->get() вернет **DB_NOTFOUND**, если указанного ключа нет в базе данных.

DB->get() возвращает ненулевое значение ошибки в случае неудачи и 0 в случае успеха.

Также можно получить набор повторяющихся записей, используя «массовый» **get**.

Для этого при вызове **DB->get()** используется флаг **DB_MULTIPLE**.

По умолчанию **DB->get()** возвращает первую найденную запись, ключ которой совпадает с ключом, предоставленным при вызове этого метода. Если база данных поддерживает повторяющиеся записи, можно немного изменить это поведение, указав флаг DB_GET_BOTH.

DB_GET_BOTH – этот флаг заставляет **DB->get()** возвращать первую запись, которая соответствует предоставленному ключу и данным.

Если указанный ключ и/или данные в базе данных не существуют, этот метод возвращает значение **DB_NOTFOUND**.

DB_RMW – Чтение-изменение-запись. Флаг обеспечивает получение блокировок записи вместо блокировок чтения во время извлечения. Это может повысить производительность многопоточных приложений за счет уменьшения вероятности взаимоблокировки.

DB_SET_RECNO – если база данных представляет собой BTree и настроена так, что в ней можно выполнять поиск по номеру логической записи, извлекается конкретная запись.

Пример

```
#include <db.h>
#include <string.h>
...
#define DESCRIPTION_SIZE 199
DBT    key, data;
DB    *my_database;
float  money;
char   description[DESCRIPTION_SIZE + 1];

// Открытие БД опущено для пущей простоты примера

money = 122.45;

// Очистка DBT перед их использованием
memset(&key,  0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));

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

data.data  = description;
data.ulen  = DESCRIPTION_SIZE + 1;
data.flags = DB_DBT_USERMEM;
my_database->get(my_database, NULL, &key, &data, 0);

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

В этом примере в поле **`data.size`** будет автоматически установлен размер извлеченных данных.

Удаление записей

Для удаления записи из базы данных используется метод **DB->del()**.

```
#include <db.h>
int DB->del(DB        *db,
            DB_TXN    *txnid,
            DBT       *key,
            u_int32_t  flags);
```

DB->del() удаляет пары ключ/данные, связанные с указанным ключом **key**, из базы данных.

При наличии повторяющихся значений ключа будут удалены все записи, связанные с указанным ключом.

При вызове к базе данных, которая была преобразована во вторичный индекс с помощью метода **DB->associate()**, DB->del() удаляет пару ключ/данные из первичной базы данных и всех вторичных индексов.

DB->del() вернет **DB_NOTFOUND**, если указанного ключа нет в базе данных.

DB->del() вернет **DB_KEYEMPTY**, если база данных является базой данных Queue или Recno и указанный ключ существует, но никогда не был явно создан приложением или позже удален. Если не указано иное, метод DB->del() возвращает ненулевое значение ошибки в случае неудачи и 0 в случае успеха.

Если база данных поддерживает повторяющиеся записи, удаляются все записи, связанные с предоставленным ключом.

Чтобы удалить только одну запись из списка дубликатов, нужно использовать курсор.

Также можете удалить все записи в базе данных, используя **DB->truncate()**.

Пример

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

...

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

/* Database open omitted for clarity */

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

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

my_database->del(my_database, NULL, &key, 0);
```

Сохранность данных – спасут только транзакции

Когда выполняется модификация базы данных, она выполняется в кэше в памяти.

Это означает, что изменения данных не обязательно сбрасываются на диск, и поэтому данные после перезапуска приложения могут *не отображаться в базе данных*.

Обычно при *закрытии базы* данных ее кэш записывается на диск. Однако в случае сбоя приложения или системы нет никакой гарантии, что базы данных будут закрыты корректно.

В этом случае можно потерять данные. В крайне редких случаях также можно столкнуться с повреждением базы данных.

Поэтому, для уверенности в том, что данные будут устойчивыми при системных сбоях, и для защиты от редкой возможности повреждения базы данных, для защиты изменений базы данных необходимо использовать транзакции.

Каждый раз, когда выполняется транзакция, BDB гарантирует, что из-за сбоя приложения или системы данные не будут потеряны.

Если транзакции не используются, это подразумевает, что данные таковы, что они не должны существовать при следующем запуске приложения.

Обычно такое имеет место, если, например, BDB используется для кэширования данных, относящихся только к текущей сессии.

Если, однако, по какой-то причине транзакции не используются, но необходимо гарантировать, что изменения вашей базы данных будут сохранены, следует периодически вызывать **`DB->sync()`**.

Синхронизация приводит к тому, что любые записи в кеше в памяти и файловом кеше операционной системы записываются на диск. Поскольку они довольно дороги с точки зрения производительности, их следует использовать «экономно».

По умолчанию синхронизация выполняется каждый раз, когда нетранзакционная база данных корректно закрывается.

Можно переопределить это поведение, указав **DB_NOSYNC** при вызове **DB->close()**.

Тем не менее, можно запустить синхронизацию вручную, вызвав **DB->sync()**.

Если приложение или система дает сбой и транзакции не используется, необходимо либо отказаться от существующих баз данных и создать их заново, либо их проверить.

Проверить базу данных можно, используя **DB->verify()**.

```
#include<db.h>

int DB->verify(DB         *db,
               const char *file,     //
               const char *database,
               FILE       *outfile,  // восстановленные пары key/data, если указан поток
               u_int32_t   flags);   // DB_SALVAGE
```

DB->verify() проверяет целостность всех баз данных в файле, указанном параметром **file**, и при необходимости выводит пары ключ/данные баз данных в файловый поток, указанный параметром **outfile**.

DB->verify() не выполняет никаких блокировок даже в средах Berkeley DB, в которых настроена подсистема блокировки. Таким образом, метод следует использовать только с файлами, которые не изменяются в других потоках управления.

DB->verify() нельзя вызывать после вызова **DB->open()**.

После вызова **DB->verify()**, независимо от его возврата, дескриптор БД может быть недоступен.

Есть утилита **db_verify**, она использует **DB->verify()**.

DB->verify() вернет **DB_VERIFY_BAD**, если база данных повреждена.

Если указан флаг **DB_SALVAGE**, возврат DB_VERIFY_BAD означает, что, возможно, не все пары ключ/данные были успешно выведены.

DB->verify() возвращает ненулевое значение ошибки в случае неудачи и 0 в случае успеха.

Если базы данных не проходят корректную проверку, чтобы восстановить как можно большую часть базы данных, следует использовать команду **db_dump**.

Использование C-структур

Хранение данных в структурах — это удобный способ упаковать различные типы информации в каждую запись базы данных.

Базы данных иногда представляют как таблицу с двумя столбцами, где столбец 1 является ключом, а столбец 2 — данными.

Используя структуры, можно эффективно превратить такую таблицу в n столбцов.

Если C-структура содержит поля, не являющиеся указателями, ее можете безопасно хранить и извлекать так же, как любой примитивный тип данных (POD Plain Old Data[3]).

Пример размещения

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

typedef
struct my_struct_s {
    int  id;
    char familiar_name[MAXLINE]; // Какое-то достаточно большое значение MAXLINE
    char surname[MAXLINE];
} my_struct_t;

...

DBT          key, data;
DB          *my_database;
my_struct_t  user;
char *fname = "David";
char *sname = "Hilbert";
```

3) C++

```
// Открытие базы данных для ясности опущено

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

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

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

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

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

Чтобы получить структуру из базы, нужно предоставить свою собственную память.

Причина в том, что, как и в случае с **`float`** числами, — некоторые системы требуют, чтобы структуры были выровнены определенным образом.

Поскольку возможно, что память, предоставляемая BDB, не выровнена должным образом, для наиболее безопасного результата следует использовать собственную память приложения.

Пример извлечения

```
#include <db.h>
#include <string.h>
...
DBT  key, data;
DB  *my_database;
my_struct_t user;

// Открытие базы данных для ясности опущено

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

// Инициализация структуры
memset(&user, 0, sizeof(my_struct_t));
user.id = 1;

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

// Используем память приложения для извлечения структуры
data.data = &user;
data.ulen = sizeof(my_struct_t);
data.flags = DB_DBT_USERMEM;

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

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

Подход со структурами плох тем, что база данных становится больше, чем это строго необходимо, – каждая структура, хранящаяся в базе данных, имеет фиксированный размер, и нет никакой экономии места при хранении (например) 5-символьной фамилии по сравнению с 20-символьной фамилией.

Если нужно хранить структуры, содержащие очень большое количество массивов символов, или если необходимо хранить миллионы записей, нужно избегать этого подхода.

Альтернативный подход — C-структуры с указателями

В C-структурах можно использовать поля, которые являются указателями на динамически выделяемую память. Это полезно, если необходимо хранить строки символов (или любой другой массив) и желательно избежать каких-либо накладных расходов.

При хранении подобных структур необходимо обеспечить, чтобы все данные, на которые указывает структура и которые она содержит, были выстроены в один непрерывный блок памяти.

BDB хранит данные, расположенные по определенному адресу и определенного размера. Если структура включает поля, указывающие на динамически выделенную память, то данные, которые следует сохранить, могут быть расположены в разных местах в куче, не обязательно смежных.

Самый простой способ решить эту проблему — собрать данные в одно место в памяти, а затем сохранить эти данные. Этот процесс иногда называют ***маршалингом*** данных.

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

typedef struct my_struct_s {
    int   id;
    char *familiar_name;
    char *surname;
} my_struct_t;

...

DBT          key, data;
DB          *my_database;  // Почему надо писать T *foo а не T* foo?
my_struct_t  user;
int  buffsize, bufflen;
char fname[ ] = "Pete";
```

```
char sname[10];
char *databuff;

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

// Открытие базы данных для ясности опущено

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

// Некоторые данные структуры находятся в стеке, а некоторые – в куче.
// Чтобы сохранить данные этой структуры, нам нужно их маршаллировать – упаковать все
// в одно место в памяти.

// Получим и приготовим буфер
buffsize = sizeof(int) + (strlen(user.familiar_name) + strlen(user.surname) + 2);
databuff = malloc(buffsize);
memset(databuff, 0, buffsize);

// копируем все в буфер
memcpy(databuff, &(user.id), sizeof(int));
bufflen = sizeof(int);

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

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

```
// Теперь сохраним буфер

// Очищаем оба DBTs перед использованием
memset(&key, 0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));

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

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

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

free(sname);
free(databuff);
```

Пример получения хранимой структуры

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

typedef struct my_struct_s {
    char *familiar_name;
    char *surname;
    int id;
} my_struct_t;

...

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

// Открытие базы данных для ясности опущено

// Очищаем оба DBTs перед использованием
memset(&key, 0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));

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

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

```
// Некоторые компиляторы не допускают арифметических операций с указателями для void *
// поэтому следует вместо void * использовать char *.

buffer = data.data;

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

IFF — подход и использование пластичных массивов

Последний элемент структуры, содержащей более чем один именованный элемент может иметь незавершенный тип массива — этот элемент называется *членом пластичного массива*.

В большинстве случаев член пластичного массива игнорируется.

В частности, размер структуры остается такой, как если бы элемент пластичного массива отсутствовал, за исключением того, что такой член может иметь большее завершающее заполнение, чем подразумевает его отсутствие.

Формат IFF — Interchange File Format.


```
struct chunk_s {
    int16_t   type;   // TYPE_ID  (+/- Little/Big Endian)
    uint16_t  size;   // DATA_SZ
    char      data[]; // DATA
}
```

Алгоритм конкурентного доступ к записи

```
    REC <-- get(Record)           // читаем запись
Again:
    REC_SAV <-- REC               // сохраним исходную запись

    /* делаем что-нибудь с записью и желаем ее сохранить */

    if (REC модифицирована) {
        lock(Record)              // блокируем запись для модификации
        REC_NEW <-- get(Record)   // и перечитываем
        if (REC_NEW != REC_SAV) { // если кто-то изменил запись после
            unlock(Record)        // получения ее нами, освобождаем запись и
            REC <--  REC_NEW      // если требуется,
            goto Again            // повторим все с ее новым содержимым
        }
        put(REC, Record)          // сохраняем новое содержимое
        unlock(Record)            // освобождаем запись
    }
```