Базы данных

Лекция 06 – Berkeley DB. Вторичные базы данных.

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

2024.10.28

Оглавление

Вторичные базы данных

Обычно поиск записей базе данных выполняется с помощью ключей записи. Однако используемый ключ не всегда будет содержать информацию, необходимую для предоставления быстрого доступа к данным, которые необходимо получить.

Например, база данных содержит записи, относящиеся к пользователям.

Ключ может быть строкой, представляющей собой уникальный идентификатор человека, например идентификатор пользователя.

Данные каждой записи, скорее всего, будут содержать сложный объект, содержащий сведения о людях, такие как имена, адреса, номера телефонов и т. д.

Хотя приложение может запрашивать человека по идентификатору пользователя (то есть по информации, хранящейся в ключе), иногда бывает необходимо найти кого-то по имени.

Вместо того, чтобы перебирать все записи в вашей базе данных, проверяя каждую по очереди на наличие имени данного человека, создаются индексы на основе имен, после чего в этом индексе ищется нужное имя.

Это можно сделать, используя вторичные базы данных.

В BDB база данных, содержащая данные, называется *первичной базой данных*.

База данных, предоставляющая альтернативный набор ключей для доступа к этим данным, называется *вторичной базой данных*.

Во вторичной базе данных ключи — это альтернативный (или вторичный) индекс, а данные соответствуют ключу первичной записи.

Вторичная база данных создается следующим образом:

- создается база данных;
- открывается;
- связывается (associated) с первичной базой данных (то есть с базой данных, для которой создается индекс).

В рамках связывания вторичной базы данных с первичной необходимо предоставить обратный вызов, который используется для создания ключей вторичной базы данных. Обычно этот обратный вызов создает ключ на основе данных, найденных в ключе или в данных первичной записи базы данных.

1) Добавление или удаление записей в первичной базе данных приводит к тому, что BDB по мере необходимости обновляет вторичную базу.

2) Изменение данных записи в первичной базе данных может привести к тому, что BDB изменит запись во вторичной базе данных, в зависимости от того, вызывает ли это изменение модификацию ключа во вторичной базе данных.

!! ВАЖНО !!! Напрямую писать во вторичную базу данных невозможно – попытка записи во вторичную базу данных приводит к возврату ненулевого результата.

3) Чтобы изменить данные, на которые ссылается вторичная запись, нужно изменить первичную базу данных.

4) Исключением из этого правила является то, что для вторичной базы данных разрешены операции удаления.

5) Записи вторичной базы данных обновляются/создаются только в том случае, если функция обратного вызова, указанная для создания ключа возвращает 0.

6) Если возвращается значение, отличное от 0, то во вторичную базу данных ключ не добавляется, а в случае обновления записи любой существующий ключ будет удален.

Чтобы указать, что запись не должна индексироваться функция обратного вызова может использовать либо DB_DONOTINDEX, или какой-нибудь код ошибки за пределами пространства имен BDB.

7) При чтении записи из вторичной базы данных возвращаются данные и, возможно, ключ из соответствующей записи в первичной базе данных.

Открытие и закрытие вторичных баз данных

Открытие и закрытие вторичной базы данных выполняется так же, как для любой обычной базы данных. Разница в том, что:

1) вторичную базу данных необходимо связать с первичной базой данных, используя метод `DB->associate( )`.

2) при закрытии баз данных, прежде чем закрывать первичные, рекомендуется убедиться, что вторичные базы данных уже закрыты. Это особенно важно, если закрытие базы данных не является однопоточным.

При связывании вторичной базы данных с первичной, необходимо предоставить функцию обратного вызова, которая используется для создания ключей вторичной базы данных.

```
#include <db.h>

int DB->associate(DB      *primary,   // Первичная БД
                  DB_TXN  *txnid,     // Транзакционный контекст
                  DB      *secondary, // Вторичная БД
                  int    (*callback)(DB        *secondary,
                                     const DBT *key,  // Primary Key
                                     const DBT *data, // Primary Data
                                     DBT       *result),
                  u_int32_t flags);   // Флаги
```

DB->associate()

```
#include <db.h>

int DB->associate(DB      *primary,   // Дескриптор первичной БД
                  DB_TXN  *txnid,     // Транзакционный контекст
                  DB      *secondary, // Вторичная БД
                  int    (*callback)(DB        *secondary, // вторичная БД
                                     const DBT *key,       // ключ первичной БД
                                     const DBT *data,    // данные первичной БД
                                           DBT *result),   // ключ вторичной БД
                  u_int32_t flags);   // Флаги
```

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

Дескриптор DB, из которого вызывается метод associate( ), является первичной базой данных.

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

ВАЖНО: поскольку для работы вторичных индексов первичные ключи должны быть уникальными, первичную базу данных необходимо настроить без поддержки дубликатов. Т.е. ключ первичной БД должен быть первичным ключом – уникальным и не нулевым.

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

Параметры

primary – дескриптор первичной базы данных, которую необходимо индексировать.

txnid – если операция является частью транзакции, это дескриптор транзакции, в противном случае NULL.

secondary – параметр должен быть:
1) дескриптором открытой базы данных;
2) вновь созданной и пустой базы данных, которая будет использоваться для хранения вторичного индекса;
3) базой данных, которая ранее была связана с той же первичной базой данных и содержала вторичный индекс.
!!! ВАЖНО !!! , небезопасно связывать в качестве вторичной базы данных дескриптор, который используется другим потоком управления или имеет открытые курсоры.
Следует обратить внимание, что или вторичные ключи должны быть уникальными, или вторичная база данных должна быть настроена с поддержкой повторяющихся элементов данных.

callback – функция обратного вызова, которая создает набор вторичных ключей, соответствующих заданному первичному ключу и паре данных.
Параметр обратного вызова может иметь значение NULL, если дескрипторы первичной и вторичной базы данных были открыты с флагом **DB_RDONLY**.
flags – параметр должен быть установлен в 0 или путем побитового ИЛИ одного или нескольких значений:

DB_CREATE – если вторичная база данных пуста, выполняется проход по первичной базе данных и в пустой вторичной базе создается для нее индекс.

Эта операция потенциально очень дорогая.

Если вторичная база данных была открыта в среде, настроенной с использованием транзакций, все создание вторичного индекса выполняется в контексте одной транзакции.

!!! ВАЖНО !!! Не следует использовать вновь заполненную вторичную базу данных в другом потоке управления до тех пор, пока вызов `DB->associate()` не вернет успешный результат в первом потоке.

Также, если транзакции не используются, не следует изменять первичную базу данных, используемую для заполнения вторичной базы данных в другом потоке управления, пока вызов `DB->associate()` не вернет успешный результат в первом потоке. Если используются транзакции, Berkeley DB выполнит соответствующую блокировку, и приложению не потребуется выполнять какое-либо специальное упорядочивание операций.

DB_IMMUTABLE_KEY – указывает, что вторичный ключ является неизменяемым.

Этот флаг можно использовать для оптимизации обновлений, если вторичный ключ в первичной записи не будет изменяться после вставки первичной записи.

Чтобы вторичные ключи не изменялись, обычно не должна вызываться функция обратного вызова вторичной БД при обновлении записей в первичной. Такая оптимизация может значительно снизить накладные расходы на операции обновления, особенно, если функция обратного вызова является дорогостоящей.

!!! Следует указывать этот флаг только в том случае, если вторичный ключ для первичной записи никогда не меняется.

Если это правило будет нарушено, вторичный индекс испортится, то есть перестанет синхронизироваться с первичным.

Пример кода для открытия вторичной базы данных и связывания ее с первичной:

```
#include <db.h>

...

DB *dbp,  *sdbp;    // Дескрипторы первичной и вторичной БД
u_int32_t  flags;   // Флаг открытия первичной БД
int        ret;     // Код возврата

ret = db_create(&dbp, NULL, 0);    // Первичная БД
if (ret != 0) {
  error_processing(...);
}
ret = db_create(&sdbp, NULL, 0);   // Вторичная БД
if (ret != 0) {
  error_processing(...);
}

// Обычно имеет смысл поддерживать дубликаты для вторичных БД
ret = sdbp->set_flags(sdbp, DB_DUPSORT); // конфиг. перед открытием
if (ret != 0) {
  error_processing(...);
}
```

```
// Флаги открытия
flags = DB_CREATE;    // если не существует, следует создать

// Открываем первичную БД
ret = dbp->open(dbp,         // Указатель на дескриптор первичной БД
                NULL,        // Указатель на контекст транзакции
                "my_db.db", // имя файла, содержащего БД
                NULL,        // необязательное логическое имя БД
                DB_BTREE,    // метод доступа
                flags,       // флаги открытия
                0);          // Режим файла (по умолчанию)
if (ret != 0) {
  error_processing(...);
}
// Открываем вторичную БД
ret = sdbp->open(sdbp,           // Указатель на дескриптор вторичной БД
                 NULL,           // Указатель на контекст транзакции
                 "my_secdb.db", // имя файла, содержащего БД
                 NULL,           // необязательное логическое имя БД
                 DB_BTREE,       // метод доступа
                 flags,          // флаги открытия
                 0);             // Режим файла (по умолчанию)
if (ret != 0) {
  error_processing(...);
}
```

```
// Связываем вторичную БД с первичной
dbp->associate(dbp,             // первичная БД
               NULL,            // контекст транзакции
               sdbp,            // Вторичная БД
               get_sales_rep,   // функция обратного вызова
               0);              // флаги
```

Закрытие первичной и вторичной баз данных выполняется точно так же, как и любой другой:

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

Реализация экстракторов ключей

Каждой вторичной базе данных необходимо предоставить обратный вызов, который создает ключи из первичных записей. Этот обратный вызов предоставляется, когда выполняется связывание вторичной базы данных с первичной.

Можно создавать ключи, используя любые данные. Как правило, ключ основывается на некоторой информации, найденной в данных записи, но можно использовать информацию, найденную и в ключе первичной записи. То, как создаются ключи, полностью зависит от характера индекса, который необходимо поддерживать.

Экстрактор ключа реализуется с помощью функции, которая извлекает необходимую информацию из ключа или данных первичной записи. Эта функция должна соответствовать определенному прототипу и должна предоставляться как обратный вызов метода `associate()`.

```
int    (*callback)(DB        *secondary, // дескриптор вторичной БД
                   const DBT *key,       // ключ первичного файла
                   const DBT *data,      // данные первичного файла
                         DBT *result),   // ключ вторичного файла
```

Обратный вызов принимает четыре аргумента:

secondary – дескриптор вторичной базы данных;

key – DBT, ссылающийся на первичный ключ;

data – DBT, ссылающийся на элемент данных в первичной БД;

result – обнуленный DBT, в котором функция обратного вызова должна заполнить поля данных и размера, которые описывают вторичный ключ (или ключи).

DBT

```
typedef
struct __db_dbt {
    void      *data; // ключ/данные
    u_int32_t  size; // размер ключа/данных
    ...
    u_int32_t  flags;
} DBT;
```

Пример

Допустим, записи первичной БД содержат данные, использующие следующую структуру:

```
typedef struct vendor_s {
    char name[MAXFIELD];            // Наименование поставщика
    char street[MAXFIELD];          // Улица и номер
    char city[MAXFIELD];            // Город
    char state[3];                  // Двузначный код штата
    char zipcode[6];                // Почтовый индекс
    char phone_number[13];          // Телефон поставщика
    char sales_rep[MAXFIELD];       // Имя представителя поставщика
    char sales_rep_phone[MAXFIELD]; // И его телефон
} VENDOR;
```

Предположим, необходимо иметь возможность запрашивать первичную базу данных на основе имени торгового представителя.

В этом случае необходимо написать функцию, которая выглядит так:

```
#include <db.h>
...
int
get_sales_rep(DB        *sdbp,   // дескриптор вторичной БД
              const DBT *pkey,   // ключ записи первичной БД
              const DBT *pdata,  // данные записи первичной БД
              DBT       *skey)   // ключ записи вторичной БД
{
    VENDOR *vendor = pdata->data; // извлекаем структуру данных

    // Чистим DBT ключа вторичной БД
    memset(skey, 0, sizeof(DBT));
    // Устанавливаем в качестве вторичного ключа имя представителя.
    skey->data = vendor->sales_rep;
    skey->size = strlen(vendor->sales_rep) + 1;

    // Возвращаем 0 чтобы указать, что запись может быть создана
    return (0);
}
```

Чтобы использовать эту функцию, ее указывают в методе `associate( )` после того, как первичная и вторичная базы данных были созданы и открыты:

```
dbp->associate(dbp,              // дескриптор первичной БД
               NULL,             // контекст транзакции
               sdbp,             // дескриптор вторичной БД
               get_sales_rep, // ф-ция обратного вызова, создающая индекс
               0);               // флаги по умолчанию
```

!!! ВАЖНО !!! Berkeley DB не является реентерантной.
Функции обратного вызова не должны пытаться выполнять вызовы библиотеки (например, для снятия блокировок или закрытия открытых дескрипторов).

Повторный вход в Berkeley DB не гарантируется, и результаты не определены.

Флаги для функции обратного вызова

В поле `flags` результирующего DBT могут быть установлены следующие флаги:

DB_DBT_APPMALLOC – если функция обратного вызова выделяет память для поля данных результата (а не просто указать на первичный ключ или данные), в поле флагов DBT результата следует установить DB_DBT_APPMALLOC, что заставит Berkeley DB освободить память, когда функция завершится.

DB_DONOTINDEX – специальное значение.

Если какая-либо пара ключ/данные в первичном ключе выдает нулевой вторичный ключ и должна быть исключена из вторичного индекса, функция обратного вызова может дополнительно вернуть DB_DONOTINDEX. В противном случае функция обратного вызова должна возвращать 0 в случае успеха или в случае неудачи ошибку за пределами пространства имен Berkeley DB.

Код ошибки будет возвращен из вызова, который инициировал обратный вызов.

Если функция обратного вызова возвращает DB_DONOTINDEX для любых пар ключ/данные в первичной базе данных, вторичный индекс не будет содержать никаких ссылок на эти пары ключ/данные, а такие операции, как итерации курсора и запросы диапазона, будут отражать только соответствующее подмножество базы данных.

Если это нежелательно, приложение должно позаботиться, чтобы функция обратного вызова была четко определена для всех возможных значений и никогда не возвращает DB_DONOTINDEX.

Возврат DB_DONOTINDEX эквивалентен установке DB_DBT_MULTIPLE в DBT результата и установке поля `size` в ноль.

DB_DBT_MULTIPLE – устанавливается в поле `flags` DBT результата и позволяет вернуть несколько вторичных ключей.

```
typedef
struct __db_dbt {
    void      *data; // ключ/данные
    u_int32_t  size; // размер ключа/данных
    ...
    u_int32_t  flags;
} DBT;
```

В поле `size` будет указано количество вторичных ключей (ноль или более), а поле `data` будет указателем на массив из `size` DBT, содержащих набор вторичных ключей.

Если возвращается несколько вторичных ключей, ключи не могут повторяться.

Это значит, что для баз данных:

Recno и Queue в массиве не должно быть повторяющихся номеров записей;

Btree и Hash ключи не должны сравниваться одинаково с помощью функции сравнения вторичной базы данных.

Если ключи повторяются, операции могут завершиться неудачно, и вторичный ключ может стать несогласующимся с первичным.

Для любого из DBT в массиве возвращаемых DBT может быть установлен свой флаг DB_DBT_APPMALLOC, чтобы указать, что Berkeley DB должна освободить память, на которую ссылается поле данных этого конкретного DBT, когда работа с ней будет завершена.

Флаг DB_DBT_APPMALLOC можно комбинировать с DB_DBT_MULTIPLE в поле флагов DBT результата, чтобы указать, что Berkeley DB должна освободить целый массив по завершении дел со всеми ключами.

Работа с несколькими ключами

До сих пор мы обсуждали только индексы, как если бы между вторичным ключом и первичной записью базы данных существовала связь один к одному.

На самом деле, можно сгенерировать несколько ключей для любой данной записи.

Например, предположим, есть база данных, содержащая информацию о книгах.

Предположим, что иногда необходимо искать книги по авторам.

Поскольку иногда у книг есть несколько авторов, бывает необходимо вернуть несколько вторичных ключей для каждой книги, которая индексируется.

Для этого пишется экстрактор ключа, который возвращает DBT, элемент данных которого указывает на массив DBT.

Каждый такой член этого массива содержит один вторичный ключ.

Кроме того, DBT, возвращаемый экстрактором ключей, должен иметь поле размера, равное количеству элементов, содержащихся в массиве DBT.

Кроме того, поле флага для DBT, возвращаемого обратным вызовом, должно включать **DB_DBT_MULTIPLE**.

Пример

```
int
my_callback(DB *dbp, const DBT *pkey, const DBT *pdata, DBT *skey) {

    DBT  *tmpdbt;
    char *tmpdata1, tmpdata2;

    // Этап извлечения данных из DBT pkey или pdata, которые будут
    // использоваться для создания вторичных ключей опущен.
    // Предположим, что данные временно сохраняются в двух переменных:
    // tmpdata1 и tmpdata2.
    // Создаем массив DBT, достаточно большой для того количества
    // ключей, которые желаем вернуть. В данном случае используется
    // массив размерности 2

    tmpdbt = malloc(sizeof(DBT) * 2);
    memset(tmpdbt, 0, sizeof(DBT) * 2);

    // Назначаем вторичные ключи каждому элементу массива.
    tmpdbt[0].data = tmpdata1;
    tmpdbt[0].size = (u_int32_t)strlen(tmpdbt[0].data) + 1;
    tmpdbt[1].data = tmpdata2;
    tmpdbt[1].size = (u_int32_t)strlen(tmpdbt[1].data) + 1;
```

```
    // Теперь устанавливаем флаги для возвращаемого DBT.
    // DB_DBT_MULTIPLE необходим для того, чтобы DB знала, что DBT
    // ссылается на массив.
    // Кроме того, мы устанавливаем DB_DBT_APPMALLOC, поскольку мы
    // динамически выделяем память для поля данных DBT.
    // DB_DBT_APPMALLOC заставляет DB освободить эту память, как только
    // она закончит работу с возвращенным DBT.

    skey->flags = DB_DBT_MULTIPLE | DB_DBT_APPMALLOC;

    skey->data = tmpdbt; // Указываем в поле данных DBT адрес массива
    skey->size = 2;      // и его размерность

    return (0);
}
```

Чтение вторичных баз данных

Как и для первичной базы данных, можно считывать записи из вторичной базы данных либо с помощью методов DB->get( ) или DB->pget( ), либо с помощью курсора, установленного на вторичной базе данных.

```
#include <db.h>

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

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

Основное различие между чтением вторичной базы данных и первичной заключается в том, что при чтении записи из вторичной базы данных данные этой записи методом не возвращаются. Вместо этого возвращаются первичный ключ и данные, соответствующие вторичному ключу.

!!! Как и в первичной базе данных, если вторичная база данных поддерживает повторяющиеся записи, то `DB->get()` и `DB->pget()` возвращают только первую запись, найденную в соответствующем наборе дубликатов.

Если необходимо увидеть все записи, относящиеся к определенному вторичному ключу, следует использовать курсор, открытый во вторичной базе данных.

Ниже пример – вторичная база данных содержит ключи, связанные с полным именем человека.

```
#include <db.h>
#include <string.h>
...
DB   *secondary_database;
DBT   key;          // Ключ поиска
DBT   pkey, pdata; // Для возврата первичного ключа и данных

char *search_name = "John Deer";

/* --- открытие первичной и вторичной баз данных опущено --- */

// очистка перед использованием
memset(&key,   0, sizeof(DBT));
memset(&pkey,  0, sizeof(DBT));
memset(&pdata, 0, sizeof(DBT));

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

// Возвращается ключ из вторичной базы данных
// и данные из связанной записи из первичной БД
secondary_database->get(my_secondary_database, NULL, &key, &pdata, 0);

// Возвращается ключ из вторичной базы данных, а также ключ и данные
// из связанной записи в первичной БД
secondary_database->pget(my_secondary_database, NULL, &key, &pkey, &pdata, 0);
```

Удаление вторичных записей базы данных

Как правило, записи во вторичной базе данных напрямую не изменяют.

Чтобы изменить вторичную базу данных, необходимо изменить первичную базу данных и «позволить» DB управлять изменениями во вторичной.

Тем не менее, можно удалять записи вторичной базы данных напрямую, что приводит к удалению связанной пары первичный ключ/данные в первичной базе.

Это, в свою очередь, приводит к тому, что БД удаляет все записи вторичной базы данных, которые ссылаются на первичную запись.

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

ACHTUNG ! Если вторичная база данных содержит повторяющиеся записи, то удаление записи из набора дубликатов приведет к удалению всех дубликатов.

Удалить запись вторичной базы данных с помощью описанного механизма можно только в том случае, если первичная база данных открыта для записи.

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

...

DB   *dbp, *sdbp;     // Дескрипторы первичной и вторичной БД
DBT   key;            // DBT, используемый для удаления
int   ret;            // Возвращаемое значение из функций
```

```
char *search_name = "John Doe"; /* Name to delete */

// Создание первичной БД
ret = db_create(&dbp, NULL, 0);
if (ret != 0) {
  // Обработка ошибок
}

// Создание вторичной БД
ret = db_create(&sdbp, NULL, 0);
if (ret != 0) {
  // Обработка ошибок
}

// Обычно дубликаты для вторичных баз данных поддерживаются
ret = sdbp->set_flags(sdbp, DB_DUPSORT);
if (ret != 0) {
  // Обработка ошибок
}

// Открытие первичной БД
ret = dbp->open(dbp, NULL, "my_db.db", NULL, DB_BTREE, 0, 0);
if (ret != 0) {
  // Обработка ошибок
}
```

```
// Открытие вторичной БД
ret = sdbp->open(sdbp, NULL, "my_secdb.db", NULL, DB_BTREE, 0, 0);
if (ret != 0) {
  // Обработка ошибок
}

// Связываем вторичную БД с первичной
dbp->associate(dbp, NULL, sdbp, get_sales_rep, 0);

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

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

// Удаляем вторичную запись.
// Это приводит к удалению связанной первичной записи.
// Если какие-либо другие вторичные базы данных имеют вторичные записи,
// ссылающиеся на удаленную первичную запись, эти вторичные записи
// также удаляются.
 sdbp->del(sdbp, NULL, &key, 0);
```

Использование курсоров со вторичными базами данных

Курсоры во вторичных базах данных используются для перебора записей точно так же, как они используются для первичных.

Курсоры со вторичными базами данных можно использовать для поиска определенных записей, для поиска первой или последней записи, для получения следующей дублирующейся записи и т. д.

Однако при использовании курсоров со вторичными базами данных есть особенности:

1) любые возвращаемые данные — это данные, содержащиеся в первичной записи базы данных, на которую ссылается вторичная запись.

2) нельзя использовать `DB_GET_BOTH` и соответствующие флаги с `DB->get()` и вторичной базой данных. Вместо этого следует использовать `DB->pget()`. В этом случае, чтобы первичная запись была возвращена в результате вызова, первичный и вторичный ключи, заданные при вызове `DB->pget()`, должны совпадать со вторичным ключом и соответствующим ключом первичной записи.

Пример – поиск имени человека во вторичной базе данных и удаление всех вторичных и первичных записей, в которых используется это имя.

```
#include <db.h>
#include <string.h>
...
DB  *sdbp;          // дескриптор вторичной БД
DBC *cursorp;       // Курсор
DBT  key, data;     // DBT для удаления

char *search_name = "John Deer"; // Наименование для удаления

// Опускаем код открытия БД

// Получаем курсор для вторичной БД
sdbp->cursor(sdbp, NULL, &cursorp, 0);

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

key.data = search_name;
key.size = strlen(search_name) + 1;
// позиционируем курсор для удаления
while (cursorp->get(cursorp, &key, &data, DB_SET) == 0) {
    cursorp->del(cursorp, 0);
}
```

Соединения (Join)

Если есть две или более вторичных баз данных, связанных с первичной базой, можно получить первичные записи на основе пересечения нескольких вторичных записей.

Это делается это с помощью курсора соединения (**join cursor**).

Допустим, у нас есть структура, которая хранит информацию о продавцах бакалейных товаров.

Обычно такая структура содержит ограниченное число элементов данных, которые могут быть интересны с точки зрения запроса.

Однако, часто хранится информация о чем-то с гораздо большим количеством характеристик, например об автомобиле. В этом случае можно хранить такую информацию, как цвет, количество дверей, расход топлива, тип автомобиля, количество пассажиров, марку, модель и год выпуска...

В этом случае используется какое-то уникальное значение для первичных записей (для этой цели часто используют VIN автомобиля).

Затем создается структура, которая идентифицирует все характеристики автомобилей.

Чтобы запросить эти данные, можно создать несколько вторичных баз данных, по одной для каждой характеристики, которую есть желание запросить. Например, можно создать вторичную БД для цвета, еще одну для количества дверей, еще одну для количества пассажиров и так далее.

При этом понадобится уникальная функция извлечения ключей для каждой такой вторичной базы данных.

Все это делается с использованием концепций и методов, описанных ниже.

После того, первичная база данных создана, и созданы все интересующие вторичные базы данных, появляется возможность извлекать записи об автомобилях на основе одной характеристики.

Например, можно найти все автомобили красного цвета.

Или можно найти все автомобили, которые имеют четыре двери.

Или все автомобили, которые являются SUV...

Следующим наиболее естественным шагом является формирование составных запросов или ***объединений***. Например, найти все автомобили красного цвета производства Toyota и являющиеся минивэнами. Это можно сделать с помощью join-курсора.

Использование join-курсоров

Чтобы использовать курсор соединения, следует:

1) открыть два или более курсоров для вторичных баз данных, связанных с одной и той же первичной базой.

2) поместить каждый такой курсор на значение вторичного ключа, которое интересует. Например:

- курсор для базы данных цветов расположить на красных записях;
- курсор для базы данных моделей — на записях SUV;
- курсор базы данных марок — на Toyota.

3) создать массив курсоров и поместить в него каждый из курсоров, которые участвуют в join-запросе.

Массив должен заканчиваться NULL.

4) получить join-курсор, используя метод `DB->join()`. Этому методу необходимо передать массив вторичных курсоров, которые были открыты и установлены на предыдущих шагах.

5) итерировать набор совпадающих записей до тех пор, пока код возврата не станет отличным от 0.

6) закрыть join-курсор.

7) по завершению следует закрыть все курсоры.

Пример

```
#include <db.h>
#include <string.h>
...
DB  *automotiveDB;      // Первичная база данных
DB  *automotiveColorDB; // Вторичная БД автомобильных цветов
DB  *automotiveMakeDB;  // Вторичная БД производителей
DB  *automotiveTypeDB;  // Вторичная БД типа авто

DBC *color_curs, *make_curs, *type_curs, *join_curs; // Курсоры
DBC *carray[4];                                      // массив курсоров 3 + 0
DBT  key, data;
int  ret;

// что ищем
char *the_color = "red";
char *the_type  = "SUV";
char *the_make  = "Toyota";

// Открытие первичной и вторичной ДБ опущены

// инициализируем все указатели и структуры
color_curs = NULL;
make_curs  = NULL;
type_curs  = NULL;
join_curs  = NULL;
```

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

// (1) Открываем три курсора
ret = automotiveColorDB->cursor(automotiveColorDB, NULL, &color_curs, 0);
if (ret != 0) {
  // Обработка ошибок
}
ret = automotiveMakeDB->cursor(automotiveMakeDB, NULL, &make_curs, 0);
if (ret) != 0) {
  // Обработка ошибок
}
ret = automotiveTypeDB->cursor(automotiveTypeDB, NULL, &type_curs, 0);
if (ret) != 0) {
  // Обработка ошибок
}

// (2) Позиционируем курсоры
key.data = the_color;
key.size = strlen(the_color) + 1;

ret = color_curs->get(color_curs, &key, &data, DB_SET);
if (ret) != 0) {
  // Обработка ошибок
}
```

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

ret = make_curs->get(make_curs, &key, &data, DB_SET);
if (ret) != 0) {
  // Обработка ошибок
}

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

ret = type_curs->get(type_curs, &key, &data, DB_SET);
if (ret) != 0) {
  // Обработка ошибок
}

// (3) Формируем массив курсоров
carray[0] = color_curs;
carray[1] = make_curs;
carray[2] = type_curs;
carray[3] = NULL;

// (4) Создаем соединение
ret = automotiveDB->join(automotiveDB, carray, &join_curs, 0);
if (ret) != 0) {
  // Обработка ошибок
}
```

```
// (5) Итерируем соединенный курсор
ret = join_curs->get(join_curs, &key, &data, 0);
while (ret) == 0) {
   // делаем что-либо полезное
}

// (6) Если вышли из цикла, потому что закончились записи,
//     то он завершился успешно.
if (ret == DB_NOTFOUND) {
    // Закрываем все курсоры, базы и выходим со статусом (0).
}
```

DB->set_flags()

```
#include <db.h>

int DB->set_flags(DB *db, u_int32_t flags);
```

Настраивает базу данных.

Вызов DB->set_flags( ) является аддитивным – возможности очистить флаги нет.

DB->set_flags( ) нельзя вызывать после вызова метода DB->open( ).

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

Общие флаги

DB_CHKSUM – проверка контрольной суммы страниц, считанных в кэш из резервного хранилища файлов (SHA1 или общий алгоритм).

Если на момент вызова DB->open( ) база данных уже существует , флаг DB_CHKSUM будет игнорироваться.

DB_ENCRYPT

Шифрует базу данных, используя криптографический пароль, указанный в методах DB_ENV->set_encrypt( ) или DB->set_encrypt( ).

Если база данных уже существует на момент вызова DB->open( ), флаг DB_ENCRYPT должен быть таким же, как у существующей базы данных, иначе будет возвращена ошибка.

Зашифрованные базы данных не переносятся между компьютерами с разным порядком байтов, то есть зашифрованные базы данных, созданные на машинах с прямым порядком байтов, не могут быть прочитаны на машинах с прямым порядком байтов, и наоборот.

DB_TXN_NOT_DURABLE

Если установлен этот флаг, Berkeley DB для этой базы данных не будет записывать в журнал.

Это означает, что обновления этой базы данных используют свойства ACI (атомарность, согласованность и изоляция), но не D (долговечность), то есть целостность базы данных будет сохранена, но в случае сбоя приложения или системы целостность не сохранится.

В этом случае файл базы данных должен быть верифицирован и/или восстановлен из резервной копии после сбоя.

Чтобы обеспечить целостность после завершения работы приложения, дескрипторы базы данных должны быть закрыты без указания DB_NOSYNC, или все изменения базы данных должны быть удалены из кэша среды с помощью `DB_ENV->txn_checkpoint()` или `DB_ENV->memp_sync()`. Для всех дескрипторов базы данных в одном физическом файле должно быть установлено значение DB_TXN_NOT_DURABLE.

Вызов `DB->set_flags()` с флагами DB_ENCRYPT, DB_TXN_NOT_DURABLE, DB_CHKSUM, влияет только на указанный дескриптор DB, а также на любые другие дескрипторы Berkeley DB, открытые в области действия этого дескриптора.

Флаги BTree

DB_DUP и DB_DUPSORT

Дубликаты записей могут храниться в отсортированном или несортированном порядке.

DB_DUPSORT – сортировать дубликаты

Разрешить дублирование элементов данных в базе данных – вставка, если ключ вставляемой пары ключ/данные уже существует в базе данных, будет успешной.

Порядок дубликатов в базе данных определяется функцией сравнения дубликатов `set_dup_compare()`. Если приложение не указывает функцию сравнения, будет использоваться лексическое сравнение по умолчанию.

Указание одновременно DB_DUPSORT и DB_RECNUM является ошибкой.

DB_DUP

Разрешить дублирование элементов данных в базе данных – вставка, когда ключ вставляемой пары ключ/данные уже существует в базе данных, будет успешной.

Порядок дубликатов, если не указан иным образом с помощью операции курсора или функции сортировки дубликатов в базе данных, определяется порядком вставки.

Флаг DB_DUPSORT предпочтительнее DB_DUP по соображениям производительности.

Флаг DB_DUP следует использовать только приложениям, которым требуется упорядочивать повторяющиеся элементы данных вручную.

Вызов `DB->set_flags()` с флагом DB_DUP влияет на базу данных, включая все потоки управления, обращающиеся к базе данных.

Если база данных уже существует на момент вызова `DB->open()`, флаг `DB_DUP` должен быть таким же, как у существующей базы данных, иначе будет возвращена ошибка.

Указание одновременно DB_DUP и DB_RECNUM является ошибкой.

DB_RECNUM

Поддержка поиска в Btree с использованием номеров записей.

При вставке или удалении записи в базах данных Btree изменяются номера логических записей и это создает определенные проблемы с доступом к страницам, хранящим записи.

Как во время вставки, так и во время удаления, вся база данных должна быть заблокирована, что фактически делает базу данных для этих операций однопоточной. Указание DB_RECNUM может привести к серьезному снижению производительности некоторых приложений и наборов данных.