Базы данных

Лекция 05 – Berkeley DB. Курсоры.

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

2024.10.21

Оглавление

Использование курсоров

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

Используя курсоры, можно извлекать, размещать и удалять записи базы данных.

Если база данных допускает дублирование записей, то курсоры — это самый простой способ получить доступ ко всему, кроме первой записи для данного ключа.

Курсор базы данных относится к одной паре ключ/данные в базе данных.

Он поддерживает обход базы данных и является единственным способом доступа к отдельным повторяющимся элементам данных.

Курсоры используются для работы с коллекциями записей, для перебора базы данных и для сохранения дескрипторов отдельных записей, чтобы их можно было изменять после прочтения.

С курсором связан объект типа DBC, который создается библитотекой при открытии курсора с помощью метода **DB->cursor()**. Приложение предоставляет указатель на объект типа DBC, в котором возвращается указатель на созданный и открытый DBC.

При возврате из данного метоад курсор не инициализирован – позиционирование курсора происходит как часть первой операции с курсором.

После открытия курсора базы данных записи можно получать **DBC->get()**, сохранять **DBC->put()** и удалять **DBC->del()**.

Дополнительные операции, поддерживаемые дескриптором курсора, включают дублирование **DBC->dup()**, соединение по принципу равенства **DB->join()** и подсчет повторяющихся элементов данных **DBC->count()**.

Курсоры в конечном итоге закрываются с помощью **DBC->close()**.

Открытие и закрытие курсоров

Курсоры управляются с помощью структуры **DBC**.
Чтобы использовать курсор, необходимо его открыть с помощью метода **DB->cursor()**.

```
#include<db.h>

int DB->cursor(DB         *db,      // дескриптор объекта базы данных
               DB_TXN     *txnid,   // транзакционный контекст, если не NULL
               DBC       **cursorp, // сюда будет помещен указатель на курсор
               u_int32_t   flags);  //
```

DB->cursor() возвращает созданный курсор базы данных.
Курсоры могут использоваться с потоками, но только последовательно, при этом приложение должно самостоятельно сериализовать доступ к дескриптору курсора.
DB->cursor() возвращает ненулевое значение ошибки в случае неудачи и 0 в случае успеха.

Пример

```
#include <db.h>
...
DB  *my_database;
DBC *cursorp;

// Открытие БД опущено для пущей простоты примера

// Получение курсора
my_database->cursor(my_database, NULL, &cursorp, 0);
```

Флаги

DB_CURSOR_BULK

Настраивает курсор для оптимизации для массовых операций.

Это позволяет избежать поиска, если между операциями курсора существует высокая степень локальности. В настоящее время этот флаг действует только при использовании метода доступа **btree**. Для других методов доступа этот флаг игнорируется.

Каждая последующая операция над курсором, настроенным с помощью этого флага, пытается продолжить работу на той же странице базы данных, что и предыдущая операция. Если требуется другая страница, осуществляется возврат к поиску.

DB_WRITECURSOR

Указывает, что курсор будет использоваться для обновления базы данных. Базовое окружение базы данных должно быть открыто с использованием флага **DB_INIT_CDB.**

Флаг DB_ENV: DB_INIT_CDB – инициализирует блокировку. В данном режиме Berkeley DB обеспечивает режим множественных операций чтения либо одной операции записи.

Изоляция транзакций

Транзакции могут быть изолированы друг от друга в разной степени.

Изоляция третьей степени – полностью сериализуемые транзакции. Они обеспечивают наибольшую изоляцию. Это означает, что на протяжении всего срока транзакции всякий раз, когда поток управления читает элемент данных, этот элемент данных не будет изменяться (при условии, конечно, что поток управления сам не изменяет его).

По умолчанию Berkeley DB обеспечивает именно полную сериализуемость всякий раз, когда чтение базы данных оборачивается транзакциями.

Изоляция моментальных снимков (Snapshot Isolation) — это второй механизм обеспечения сериализуемости, в котором предпочтение отдается процессам чтения, а не процессам обновления.

Отказ от блокировок чтения обычно значительно повышает пропускную способность для многих приложений.

Изоляция второй степени – обеспечивается только стабильность курсора, то есть гарантируется, что курсоры видят зафиксированные данные, которые не изменяются, пока к ним этот курсор обращается, но могут измениться до завершения транзакции чтения.

Изоляция первой степени – поддержка чтения незафиксированных данных – то есть, операции чтения могут получить данные, которые были изменены, но еще не зафиксированы другой транзакцией.

DB_READ_COMMITTED

Настраивает транзакционный курсор на изоляцию степени 2.

Это гарантирует стабильность текущего элемента данных, считываемого этим курсором, но позволяет изменять или удалять эти данные до фиксации транзакции для этого курсора.

DB_READ_UNCOMMITTED

Настраивает транзакционный курсор на изоляцию степени 1.

Операции чтения, выполняемые курсором, могут возвращать измененные, но еще не зафиксированные данные.

Молча игнорируется, если при открытии основной базы данных не был указан флаг **DB_READ_UNCOMMITTED**[1].

---

1) DB_READ_UNCOMMITTED – при открытии БД этот флаг обеспечивает поддержку транзакционных операций чтения с изоляцией степени 1. Операции чтения в базе данных могут запросить возврат измененных, но еще не зафиксированных данных. Этот флаг должен быть указан для всех дескрипторов БД, используемых для выполнения «грязного» чтения или обновления базы данных, иначе запросы на «грязное чтение» могут не обрабатываться, и чтение может быть заблокировано.

DB_TXN_SNAPSHOT

Настраивает транзакционный курсор для работы с изоляцией моментальных снимков только для чтения. Для баз данных с установленным флагом **DB_MULTIVERSION**[2] значения данных будут считываться такими, какие они были при открытии курсора, без блокировки чтения.

Этот флаг неявно начинает транзакцию, которая фиксируется при закрытии курсора.

Этот флаг игнорируется, если в основной базе данных не был установлен **DB_MULTIVERSION** или если транзакция указана в параметре **txnid**.

---

2) DB_MULTIVERSION – открывает базу данных с поддержкой многоверсионного управления параллелизмом. Это приведет к тому, что обновления базы данных будут выполняться по протоколу копирования при записи, который необходим для поддержки изоляции моментальных снимков. Флаг DB_MULTIVERSION требует, чтобы база данных была защищена транзакциями во время ее открытия и не поддерживается форматом очереди.

Ошибки

Метод **DB->cursor()** может завершиться неудачей и вернуть одну из следующих ненулевых ошибок:

DB_REP_HANDLE_DEAD
Имеет отношение к репликации. В процессе синхронизации клиента с мастером, зафиксированные транзакции могут быть отменены. Это аннулирует все дескрипторы базы данных и курсоров, открытые в среде репликации. Как только это произойдет, попытка использовать такой дескриптор вернет **DB_REP_HANDLE_DEAD**.

Приложению в этом случае, чтобы продолжить обработку, следует отказаться от дескриптора и открыть новый.

DB_REP_LOCKOUT
Операция была заблокирована синхронизацией клиента и мастера.

EINVAL
Было указано неверное значение флага или параметр.

После завершения работы с курсором, его необходимо закрыть с пом. метода **DBC->close()**.

Закрытие базы данных, когда курсоры все еще открыты, особенно если эти курсоры записывают в базу, может иметь непредсказуемые результаты.

Рекомендуется закрывать все дескрипторы курсора после их использования, чтобы обеспечить параллелизм и освободить ресурсы, такие как блокировки страниц.

```
#include <db.h>

...

DB  *my_database;
DBC *cursorp;

// Открытие базы данных и курсора опущено для пущей простоты примера

if (cursorp != NULL) {              // Имеет смысл проверить
    cursorp->close(cursorp);        // Сначала закрываем курсор
}

if (my_database != NULL) {          // И только потом базу данных
    my_database->close(my_database, 0);
}
```

Получение записей с помощью курсора

Чтобы просмотреть записи базы данных, от первой записи до последней, нужно просто открыть курсор, после чего использовать метод **DBC->get( )**.

```
#include<db.h>

int DBcursor->get(DBC *DBcursor,
                  DBT *key,    // ключ из базы данных
                  DBT *data,   // даные из базы данных
                  u_int32_t flags);

int DBcursor->pget(DBC *DBcursor,
                   DBT *key,   // ключ из вторичной базы данных (индекса)
                   DBT *pkey,  // ключ из первичной базы данных
                   DBT *data,  // даные из первичной базы данных
                   u_int32_t flags);
```

DBcursor->get() извлекает пары ключ/данные из базы данных.

Адрес и длина ключа возвращаются в объекте DBT, на который ссылается **key** (за исключением случая флага **DB_SET**, в котором ключевой объект не изменяется), а адрес и длина данных возвращаются в объекте DBT, на который ссылается **data**.

При вызове курсора, открытого в базе данных, которая была преобразована во вторичный индекс с помощью метода **DB->associate( )**, **DBcursor->get( )** и **Dbcursor->pget( )** возвращают ключ из вторичного индекса и элемент данных из первичной базы данных. Кроме того, **DBcursor->pget( )** возвращает ключ из первичной базы данных.

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

В процессе последовательного сканирования базы данных изменения будут отражаться следующим образом – записи, вставленные позади курсора, возвращаться не будут, а записи, вставленные перед курсором, будут.

Если не указано иное, **DBcursor->get()** возвращает ненулевое значение ошибки в случае неудачи и 0 в случае успеха.

Если **DBcursor->get()** по какой-либо причине завершится неудачно, состояние курсора останется неизменным.

Основные флаги

DB_FIRST (DB_LAST)

Инициализация курсора.

Курсор устанавливается для ссылки на первую (последнюю) пару **key/data** базы данных, и эта пара возвращается.

Если первый (последний) ключ имеет повторяющиеся значения, возвращается первый (последний) элемент данных в наборе дубликатов.

Если база данных представляет собой базу данных Queue или Recno, **DBcursor->get()** с флагом **DB_FIRST** (**DB_LAST**) будет игнорировать любые существующие ключи, которые никогда не создавались приложением явно или были созданы, а затем удалены.

Если установлен флаг **DB_FIRST (DB_LAST)** и база данных пуста, **DBcursor->get()** вернет **DB_NOTFOUND**.

DB_NEXT (DB_PREV)

Если курсор еще не инициализирован, **DB_NEXT (DB_PREV)** идентичен **DB_FIRST (DB_LAST)**.

В противном случае курсор перемещается на следующую (предыдущую) пару **key/data** базы данных, и эта пара возвращается.

При наличии повторяющихся значений ключа значение ключа может не изменяться.

Если база данных представляет собой базу данных Queue или Recno, **DBcursor->get()** с флагом **DB_NEXT (DB_PREV)** пропустит все ключи, которые существуют, но никогда не были явно созданы приложением, или те, которые были созданы и позже удалены.

Если установлен **DB_NEXT/DB_PREV** и курсор уже находится на последней (первой) записи в базе данных, **DBcursor->get()** вернет **DB_NOTFOUND**.

DB_CURRENT

Возвращает пару ключ/значение, на которую ссылается курсор.

Если установлен флаг **DB_CURRENT** и пара ключ/значение курсора была удалена, **DBcursor->get()** вернет ошибку **DB_KEYEMPTY.**

Примеры

Использование флага DB_NEXT для итерирования от первой записи до последней

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

...

DB  *my_database;
DBC *cursorp;
DBT key, data;
int ret;

// Открытие БД опущено

// Получаем курсор
my_database->cursor(my_database, NULL, &cursorp, 0);

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

// Итерируем по БД, извлекая каждую запись в цикле
while (0 == (ret = cursorp->get(cursorp,
                                &key,
                                &data,
                                DB_NEXT))) {
        // С DBT делаем что-нибудь полезное
}
```

```
if (ret != DB_NOTFOUND) {
        // А это ошибка и ее нужно обработать ...
}


...
// Курсор должен быть закрыт
if (cursorp != NULL) {
    cursorp->close(cursorp);
}
...
// до того, как будет закрыта его база данных
if (my_database != NULL) {
    my_database->close(my_database, 0);
}
```

Использование флага DB_PREV для итерирование по базе от последней записи к первой

```
#include <db.h>
#include <string.h>
...
DB  *my_database;
DBC *cursorp;
DBT key, data;
int ret;

// Открытие БД опущено
// Получаем курсор
my_database->cursor(my_database, NULL, &cursorp, 0);

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

// Итерируем по БД, извлекая каждую запись в цикле
while (0 == (ret = cursorp->get(cursorp,
                                &key,
                                &data,
                                DB_PREV))) {
        // Что-нибудь полезное с DBT делаем
}
if (ret != DB_NOTFOUND) {
        // А это ошибка и ее нужно обработать ...
}
```

Поиск записей

Можно использовать курсоры для поиска записей в базе.

Можно выполнять поиск только по ключу или искать одновременно по ключу и данным.

Также можно выполнять поиск по частичным совпадениям, если база данных поддерживает отсортированные наборы дубликатов.

Во всех случаях параметры ключа и данных этих методов заполняются значениями ключа и данных записи базы данных, на которую позиционируется курсор в результате поиска.

Если поиск не удался, то состояние курсора остается неизменным и возвращается **DB_NOTFOUND**.

Чтобы использовать курсор для поиска записи, следует использовать **DBC->get()**.

При использовании этого метода можно указать несколько флагов.

Если курсор проверяет только ключевую часть записей, в имени флагов курсора используется **SET** — в этом случае курсор устанавливается на запись, ключ которой соответствует значению, предоставленному курсору.

DB_SET — Перемещает курсор на первую запись в базе данных с указанным ключом.

DB_SET_RANGE — Идентичен **DB_SET**, но только в том случае, когда не используется тип доступа BTree. В этом случае курсор перемещается на первую запись в базе данных, ключ которой больше или равен указанному ключу. Это сравнение определяется функцией сравнения, которую необходимо предоставить базе данных.

Если функция сравнения не предусмотрена, то используется лексикографическая сортировка по умолчанию.

Когда курсор использует флаги, в имени которого присутствует **GET**, курсор позиционируется как на ключ, так и на значения данных.

Независимо от ключевого слова, которое используется для получения записи с курсором, **DBT** ключа и данных курсора заполняются данными, извлеченными из записи, на которой в данный момент расположен курсор.

DB_SET

Перемещает курсор к указанной паре ключ/данные базы данных и возвращает данные, связанные с данным ключом.

Если установлен **DB_SET**, но соответствующие ключи не найдены, **DBcursor->get()** вернет **DB_NOTFOUND**.

При наличии повторяющихся значений ключа **DBcursor->get()** вернет первый элемент данных для данного ключа.

DB_SET_RANGE

Перемещает курсор на указанную пару ключ/данные базы данных.

В случае метода доступа Btree возвращается ключ вместе с элементом данных, а возвращаемая пара **key/data** представляет собой наименьший ключ, больший или равный указанному ключу, как определено функцией сравнения Btree.

Это позволяет реализовать частичные ключевые совпадения и поиск по диапазону.

Например, предположим, что у нас есть записи базы данных, в которых в качестве ключей используются следующие строки:

Алабама
Аляска
Аризона

Затем при предоставлении ключа поиска **Аляска** курсор перемещается ко второму ключу.
Если представлен ключ **Ал**, курсор переместится к первому ключу (Алабама).
Если представлен ключ **Аляс**, курсор переместится ко второму ключу (Аляска).
Если представлен ключ **Ар** , курсор переместится к последнему ключу (Аризона).

DB_GET_BOTH

Курсор устанавливается на пару ключ/данные если и **key**, и **data** одновременно соответствуют значениям, указанным в параметрах **key** и **data**.

Во всем остальном этот флаг идентичен флагу **DB_SET**.

При использовании с **DBcursor->pget()** для дескриптора вторичного индекса и вторичный, и первичный ключи должны соответствовать элементу вторичного и первичного ключа в базе данных.

DB_GET_BOTH_RANGE

Перемещает курсор на указанную пару ключ/данные базы данных и возвращает данные.

Параметр **key** должен точно совпадать с ключом в базе данных, параметр **data** – не обязательно.

Извлеченный элемент данных — это элемент в повторяющемся наборе, который имеет наименьшее значение, которое *больше или равно* значению, предоставленному параметром **data** в порядке действующей функции сравнения.

Если указан этот флаг для базы данных, настроенной без поддержки сортировки дубликатов, поведение идентично флагу **DB_GET_BOTH**. Во всем остальном этот флаг идентичен флагу **DB_GET_BOTH**.

Например, база данных использует BTree и в ней есть записи, в которых используются следующие пары ключ/данные:

Alabama/Athens
Alabama/Florence
Alaska/Anchorage
Alaska/Fairbanks
Arizona/Avondale
Arizona/Florence

Ключ поиска	Данные	Положение курсора
Alaska	Fa	Alaska/Fairbanks
Arizona	Fl	Arizona/Florence
Alaska	An	Alaska/Anchorage

Пример кода

Пусть база данных содержит отсортированные повторяющиеся записи пар ключ/данные в виде строк – «штат»/«город», тогда следующий фрагмент кода можно использовать для установки курсора на любую запись в базе данных и вывода значений ее ключа и данных:

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

...

DBC  *cursorp;
DBT   key, data;
DB   *dbp;
int   ret;
char *search_data = "Fa";
char *search_key  = "Alaska";

// Открытие базы данных опущено

// Получаем курсор
dbp->cursor(dbp, NULL, &cursorp, 0);

// Устанавливаем обв DBT
key.data  = search_key;              // Alaska
key.size  = strlen(search_key) + 1;
data.data = search_data;             // Fa
data.size = strlen(search_data) + 1;
```

```
// Позиционируем курсор на первую запись в базе данных, ключ которой
// соответствует ключу поиска и чьи данные начинаются с данных поиска
ret = cursorp->get(cursorp, &key, &data, DB_GET_BOTH_RANGE);
if (!ret) {
    // Что-то делаем с полученными данными
} else {
    // Ошибка, тоже обрабатываем
}

// Закрываем курсор
if (cursorp != NULL) {
    cursorp->close(cursorp);
}
...
// Прежде, чем закрываем БД
if (dbp != NULL) {
    dbp->close(dbp, 0);
}
```

Работа с повторяющимися записями (резюме)

1) Запись является дубликатом другой записи, если две записи имеют один и тот же ключ.
2) Для таких повторяющихся записей уникальной является только часть записи – данные.
3) Повторяющиеся записи поддерживаются только для методов доступа **BTree** и **Hash**.

Если база данных поддерживает повторяющиеся записи, то потенциально она может содержать несколько записей с одним и тем же ключом.

По умолчанию обычные операции получения из базы данных возвращают только первую такую запись из набора дубликатов. Поэтому доступ к последующим повторяющимся записям следует осуществлять с помощью курсора.

При работе с базами данных, поддерживающими повторяющиеся записи, используются следующие флаги метода **`DBC->get()`**:

DB_NEXT, DB_PREV – выдает следующую/предыдущую запись, независимо от того, является ли она дубликатом текущей записи.

DB_GET_BOTH_RANGE – используется для поиска с помощью курсора определенной записи, независимо от того, является ли она дубликатом.

DB_NEXT_NODUP, **DB_PREV_NODUP** – выдает следующую/предыдущую неповторяющуюся запись в базе данных.

Флаг позволяет пропустить все дубликаты в наборе повторяющихся записей. Если вызывается **`DBC->get()`** с **`DB_PREV_NODUP`**, то курсор позиционируется на последнюю запись для предыдущего ключа в базе данных.

Например, если в базе данных есть следующие записи:

```
Alabama / Athens
Alabama / Florence  <==
Alaska  / Anchorage
Alaska  / Fairbanks <--
Arizona / Avondale
Arizona / Florence
```

и курсор позиционируется на **Alaska/Fairbanks**, после чего вызывается **DBC->get()** с **DB_PREV_NODUP**, то курсор позиционируется на **Alabama/Florence**.

Аналогично, если вызывается **DBC->get()** с **DB_NEXT_NODUP**, то курсор позиционируется на первую запись, соответствующую следующему ключу в базе данных.

Если в базе данных нет следующего/предыдущего ключа, то возвращается **DB_NOTFOUND**, а курсор остается на месте.

DB_NEXT_DUP – выдает следующую запись, которая использует текущий ключ.

Если же курсор находится на последней записи в наборе дубликатов и вызван **DBC->get()** с **DB_NEXT_DUP**, то возвращается **DB_NOTFOUND**, а курсор остается на месте.

Следующий фрагмент кода устанавливает курсор на ключ и выводит соответствующую запись и все ее дубликаты.

```
#include <db.h>
#include <string.h>
...
DB *dbp;
DBC *cursorp;
DBT key, data;
```

```
int ret;
char *search_key = "Al";

// Открытие базы данных опущено

// Получаем курсор
dbp->cursor(dbp, NULL, &cursorp, 0);

// Устанавливаем наши DBT
key.data = search_key;
key.size = strlen(search_key) + 1;

// Позиционируем курсор на первую запись в базе данных, ключ и дата которой
// начинаются с нужной строки

ret = cursorp->get(cursorp, &key, &data, DB_SET);
while (ret != DB_NOTFOUND) { // Если нашлась, циклим по дубликатам
    printf("key: %s, data: %s\n", (char *)key.data, (char *)data.data);
    ret = cursorp->get(cursorp, &key, &data, DB_NEXT_DUP);
}

// Закрываем курсор
if (cursorp != NULL) {
    cursorp->close(cursorp);
}
...
// И только потом закрываем БД
if (dbp != NULL) {
    dbp->close(dbp, 0);
}
```

Путем побитового включающего ИЛИ в параметре **flags** метода **get()** могут быть установлены следующие флаги:

DB_READ_COMMITTED

Настраивает транзакционную операцию **get** для изоляции степени 2 (чтение не повторяющееся).

DB_READ_UNCOMMITTED

Элементы базы данных, прочитанные во время транзакционного вызова, будут иметь изоляцию степени 1, включая измененные, но еще не зафиксированные данные. Однако, молча игнорируется, если при открытии базовой базы данных не был указан флаг **DB_READ_UNCOMMITTED**.

DB_RMW (Read-Modify-Write)

Вместо блокировок чтения при чтении метод получает блокировки записи.

Установка этого флага может устранить взаимоблокировку во время цикла «чтение-изменение-запись» путем получения блокировки записи во время части чтения цикла, так что другой поток управления получит блокировку чтения для того же элемента в своем собственном цикле «чтение-изменение-запись», что не приведет к взаимоблокировке.

DB_MULTIPLE

Запрос возврата нескольких элементов данных в параметре **data**.

В случае баз данных Btree или Hash дублирующиеся элементы данных для текущего ключа, начиная с текущей позиции курсора, помещаются в буфер.

Последующие вызовы с указанными флагами **DB_NEXT_DUP** и **DB_MULTIPLE** будут возвращать дополнительные повторяющиеся элементы данных, связанные с текущим ключом, или **DB_NOTFOUND**, если нет дополнительных повторяющихся элементов данных для возврата.

Последующие вызовы с указанными флагами **DB_NEXT** и **DB_MULTIPLE** вернут дополнительные повторяющиеся элементы данных, связанные с текущим ключом, или, если дополнительных повторяющихся элементов данных нет, вернут следующий ключ и его элементы данных.

Если же в базе данных нет дополнительных ключей, вернут **DB_NOTFOUND**.

DB_MULTIPLE_KEY

Возвращает несколько пар ключ/данные в параметре **data**.

Пары ключей и данных, начиная с текущей позиции курсора, помещаются в буфер.

Последующие вызовы с указанными флагами **DB_NEXT** и **DB_MULTIPLE_KEY** будут возвращать дополнительные пары ключ/данные или **DB_NOTFOUND**, если нет дополнительных ключей и элементов данных для возврата.

В случае баз данных Btree, Hash или Heap несколько пар ключей и данных можно перебирать с помощью макроса **DB_MULTIPLE_KEY_NEXT**.

В случае баз данных Queue или Recno несколько номеров записей и пар данных можно перебирать с помощью макроса **DB_MULTIPLE_RECNO_NEXT**.

Буфер, на который ссылается параметр данных, должен быть предоставлен из пользовательской памяти (**DB_DBT_USERMEM**).

Буфер должен быть по крайней мере такого же размера, как размер страницы основной базы данных, и выровнен для доступа к беззнаковым целым числам (`unsigned int`) и быть кратным 1024 байтам.

Если размер буфера недостаточен, то при возврате из вызова будет возвращена ошибка **DB_BUFFER_SMALL**, а в поле размера параметра данных будет установлен предполагаемый размер буфера.

Размер этот является приблизительным, поскольку точный необходимый размер может быть неизвестен до тех пор, пока не будут прочитаны все записи. Поэтому слкдует изначально предоставить относительно большой буфер, но приложения должны быть готовы изменять размер буфера по мере необходимости и неоднократно вызывать метод.

Флаги **DB_MULTIPLE** и **DB_MULTIPLE_KEY** можно использовать только с флагами:
DB_SET, DB_SET_RANGE, DB_SET_RECNO
DB_CURRENT
DB_FIRST
DB_NEXT, DB_NEXT_DUP, DB_NEXT_NODUP
DB_GET_BOTH, DB_GET_BOTH_RANGE

Флаг **DB_MULTIPLE** нельзя использовать при доступе к базам данных, преобразованным во вторичные индексы, с помощью метода **DB->associate( )**.

Размещение записей с помощью курсоров

Можно использовать курсоры и для помещения записей в базу данных.
Поведение БД при этом различается в зависимости от:
- флагов, которые используется при записи;
- используемого метода доступа;
- поддерживает ли база данных отсортированные дубликаты.

При помещении записи в базу данных с помощью курсора курсор устанавливается на вставленной записи.

Для помещения (записи) записей в базу данных используется метод **DBC->put()**.

```
#include <db.h>

int DBcursor->put(DBC *DBcursor,
                  DBT *key,
                  DBT *data,
                  u_int32_t flags);
```

Метод **DBcursor->put()** возвращает:
- 0 в случае успеха;
- ненулевое значение ошибки в случае неудачи.

Если **DBcursor->put()** по какой-либо причине завершится неудачно, состояние курсора останется неизменным.

Если **DBcursor->put()** завершается успешно и элемент вставляется в базу данных, курсор всегда располагается так, чтобы ссылаться на вновь вставленный элемент.

Флаги:

DB_AFTER

В случае методов доступа Btree и Hash вставляет элемент данных за записью, на которую ссылается курсор.

DB_BEFORE

В случае методов доступа Btree и Hash вставляет элемент данных сразу перед записью, на которую ссылается курсор.

DB_CURRENT

Заменяет данные в паре ключ/данные в записи, на который ссылается курсор.

Отсортированные дубликаты

Дубликаты записей могут храниться в отсортированном или несортированном порядке. Чтобы указать необходимость автоматически сортировать дубликаты записей, следует во время создания базы данных указать флаг **DB_DUPSORT**.

Если отсортированные дубликаты поддерживаются, то для определения местоположения дублированной записи в ее наборе дубликатов, используется предоставленная функция сортировки, указанная в методе **DB->set_dup_compare()**.

Если такая функция не указана, то используется лексикографическое сравнение по умолчанию.

Несортированные дубликаты

Из соображений производительности BTrees всегда должны содержать отсортированные записи. (BTrees, содержащие неотсортированные записи, потенциально должны тратить гораздо больше времени на поиск записи, чем BTree, содержащий отсортированные записи). При этом DB предоставляет поддержку для подавления автоматической сортировки дубликатов записей, поскольку может оказаться, что ваше приложение вставляет записи, которые уже находятся в отсортированном порядке.

То есть, если база данных настроена на поддержку неотсортированных дубликатов, то предполагается, что ваше приложение будет выполнять сортировку самостоятельно. В этом случае следует ожидать значительное снижение производительности. Каждый раз, когда записи помещаются в базу данных в порядке сортировки, неизвестном DB, приходится платить снижением производительности.

DB ведет себя при вставке записей в базу данных, которая поддерживает неотсортированные дубликаты, следующим образом:

- Если приложение просто добавляет дублирующую запись с помощью **DB->put()**, то запись вставляется в конец ее отсортированного набора дубликатов.

- Если для помещения дублирующейся записи в базу данных используется курсор, то новая запись помещается в набор дубликатов в соответствии с флагами, которые предоставляются в методе **DBC->put()**. Соответствующие флаги:

DB_AFTER

Данные, предоставленные при вызове **DBC->put()**, помещаются в базу данных как дублирующая запись. Ключ, используемый для этой операции, — это ключ, используемый для записи, на которую в данный момент ссылается курсор. Поэтому любой ключ, предоставленный при вызове **DBC->put()**, игнорируется.

Дублирующая запись вставляется в базу данных сразу после текущей позиции курсора в базе данных.

Этот флаг игнорируется, если для базы данных поддерживаются отсортированные дубликаты.

DB_BEFORE

Ведет себя так же, как **DB_AFTER**, за исключением того, что новая запись вставляется непосредственно перед текущим местоположением курсора в базе данных.

DB_KEYFIRST

Для баз данных, не поддерживающих дубликаты, этот метод ведет себя точно так же, как если бы была выполнена вставка по умолчанию.

Если же база данных поддерживает повторяющиеся записи и была предоставлена функция сортировки дубликатов, вставляемый элемент данных добавляется в отсортированное место.

Если ключ, предоставленный при вызове **DBC->put()**, в базе данных уже существует, и база данных настроена на использование дубликатов без сортировки, то новая запись вставляется как первая запись в соответствующий список дубликатов.

DB_KEYLAST

Действует идентично **DB_KEYFIRST**, за исключением того, что новая дублирующая запись вставляется как последняя запись в списке дубликатов.

DB_NODUPDATA

Если предоставленный ключ в базе данных уже существует, этот метод возвращает значение **DB_KEYEXIST**. Если ключ не существует, то порядок помещения записи в базу данных определяется порядком вставки, используемым базой данных:

- если в базе данных предусмотрена функция сравнения, запись вставляется в отсортированное место;

- в противном случае (если BTree) используется лексикографическая сортировка, при которой более короткие элементы упорядочиваются перед более длинными элементами.

Этот флаг можно использовать только для методов доступа BTree и Hash и только в том случае, если база данных настроена на поддержку отсортированных дубликатов.

Настройка базы данных для поддержки дубликатов

Поддержка дубликатов может быть настроена только во время создания базы данных. Это делается путем указания соответствующих флагов в **DB->set_flags()** перед первым открытием базы данных:

DB_DUP

База данных поддерживает несортированные дубликаты записей.

DB_DUPSORT

База данных поддерживает сортированные дубликаты записей. Следует обратить внимание, что этот флаг также устанавливает и флаг **DB_DUP**.

Следующий фрагмент кода иллюстрирует, как настроить базу данных для поддержки сортированных дубликатов записей:

```
#include <db.h>
...
DB   *dbp;
FILE *error_file_pointer;
int   ret;
char *program_name = "my_prog";
char *file_name    = "mydb.db";

// Для краткости присвоения переменным опущены
// Аналогично и в отношении инициализации дескриптора базы данных

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

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

// Настройка базы данных для сортировки дубликатов

ret = dbp->set_flags(dbp, DB_DUPSORT);
if (ret != 0) {
    dbp->err(dbp, ret, "Попытка установить флаг DUPSORT не удалась.");
    dbp->close(dbp, 0);
    return(ret);
}

// А теперь открываем базу данных
ret = dbp->open(dbp,       // Указатель на базу данных
                NULL,      // Указатель на транзакцию
                file_name, // Имя файла
                NULL,      // Лочисеское имя БД не требуется
                DB_BTREE,  // Тип тоступа к базе данных
                DB_CREATE, // Флаги открытия
                0);        // Режим файла по умолчанию
if (ret != 0) {
    dbp->err(dbp, ret, "Database '%s' open failed.", file_name);
    dbp->close(dbp, 0);
    return(ret);
}
```

Пример

```
#include <db.h>
#include <string.h>
...
DB   *dbp;
DBC  *cursorp;
DBT   data1, data2, data3;
DBT   key1, key2;
char *key1str  = "Моя первая строка";
char *data1str = "Мои первые данные";
char *key2str  = "Вторая строка";
char *data2str = "Мои вторые данные";
char *data3str = "Мои третьи данные";
int   ret;

// Опускаем открытие БД

// Устанавливаем наши DBT
key1.data  = key1str;
key1.size  = strlen(key1str) + 1;
data1.data = data1str;
data1.size = strlen(data1str) + 1;

key2.data  = key2str;
key2.size  = strlen(key2str) + 1;
data2.data = data2str;
data2.size = strlen(data2str) + 1;
data3.data = data3str;
data3.size = strlen(data3str) + 1;
```

```
// Получаем курсор
dbp->cursor(dbp, NULL, &cursorp, 0);

// Предполагая, что база данных пуста, сначала помещаем пару
// «Моя первая строка»/«Мои первые данные» в первую позицию в базе данных.
ret = cursorp->put(cursorp, &key1, &data1, DB_KEYFIRST);

// Далее помещаем «Вторая строка»/«Мои вторые данные» в базу данных в соответствии
// с сортировкой ключей по ключу, используемому для существующей в данный момент
// записи базы данных. Эта запись окажется первой в базе данных.
ret = cursorp->put(cursorp, &key2, &data2, DB_KEYFIRST); // в порядке сортировки

// Если дубликаты не разрешены, существующая в данный момент запись, использующая
// key2, перезаписывается данными, в этом месте предоставленными. То есть запись
// "Вторая строка"/"Мои вторые данные" становится "Вторая строка"/"Мои третьи данные"
// Если дубликаты разрешены, то «Мои третьи данные» помещаются в список дубликатов в
//  соответствии с тем, как они сортируются по «Мои вторые данные».
ret = cursorp->put(cursorp, &key2, &data3, DB_KEYFIRST);
```

Если дубликаты не разрешены, запись будет перезаписана новыми данными

Если дубликаты разрешены, запись будет добавлена в начало списка дубликатов

Удаление записей с помощью курсоров

Чтобы удалить запись с помощью курсора, нужно поместите курсор на запись, которую необходимо удалить, после чего вызвать метод **DBC->del( )**.

Пример

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

...

DB   *dbp;
DBC  *cursorp;
DBT   key, data;
char *key1str = "My first string";
int   ret;

// Опускаем открытие БД

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

// Устанавливаем DBT
key.data = key1str;
key.size = strlen(key1str) + 1;
```

```
// Получаем курсор
dbp->cursor(dbp, NULL, &cursorp, 0);

// Итерируем по БД, удаляя каждую запись
while (0 == (ret = cursorp->get(cursorp, &key, &data, DB_SET))) {
    cursorp->del(cursorp, 0);
}

// Закрываем курсор
if (cursorp != NULL) {
    cursorp->close(cursorp);
}
...
// И только потом закрываем БД
if (dbp != NULL) {
    dbp->close(dbp, 0);
}
```

Замена записей с помощью курсоров

Для замены записей используется метод **DBC->put()** с флагом **DB_CURRENT**.

DB_CURRENT

Перезаписывает данные пары ключ/данные, на которую ссылается курсор, указанным элементом данных. Ключевой параметр игнорируется.

Если текущая запись курсора уже удалена, **DBcursor->put()** вернет **DB_NOTFOUND**.

Пример

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

...

DB   *dbp;
DBC  *cursorp;
DBT   key, data;
char *key1str          = "My first string";
char *replacement_data = "replace me";
int   ret;

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

```
// Устанавливаем DBT
key.data = key1str;
key.size = strlen(key1str) + 1;

// Опускаем открытие БД

// Получаем курсор
dbp->cursor(dbp, NULL, &cursorp, 0);

// Позиционируем
ret = cursorp->get(cursorp, &key, &data, DB_SET);
if (ret == 0) { // Если все в порядке
    data.data = replacement_data;
    data.size = strlen(replacement_data) + 1;
    cursorp->put(cursorp, &key, &data, DB_CURRENT);
}

// Закрываем курсор
if (cursorp != NULL) {
    cursorp->close(cursorp);
}
...
// И только потом закрываем БД
if (dbp != NULL) {
    dbp->close(dbp, 0);
}
```

!!! Важно !!!

Используя этот метод невозможно изменить ключ записи —
при замене записи ключевая компонента всегда игнорируется.

При замене части данных записи, если заменятся запись, являющаяся членом набора отсортированных дубликатов, замена будет успешной, только в том случае если новая запись сортируется так же, как и старая запись.

Это означает, что если заменяется запись, входящая в набор отсортированных дубликатов, и если используется лексикографическая сортировка по умолчанию, то в случае нарушения порядка сортировки замена выполнена не будет. Однако, если предоставлена пользовательская процедура сортировки, которая, например, сортирует на основе всего нескольких байтов из элемента данных, то потенциально можно не нарушить вышеуказанные ограничения и выполнить прямую замену.

В обстоятельствах, если необходимо заменить данные, содержащиеся в дублирующейся записи, а пользовательская процедура сортировки не используется, следует удалить запись и создать новую с нужным ключом и данными.

Степени изоляции

Транзакции могут быть изолированы друг от друга в разной степени.

Сериализуемые транзакции (изоляция степени 3) обеспечивают наибольшую изоляцию и означает, что на протяжении всего срока транзакции всякий раз, когда поток управления читает элемент данных, этот элемент данных не будет изменяться (при условии, конечно, что поток управления сам не изменяет его).

По умолчанию Berkeley DB обеспечивает именно сериализуемость всякий раз, когда чтение базы данных оборачивается транзакциями.

Большинству приложений не требуется заключать все операции чтения в транзакции, и, когда это возможно, следует избегать транзакционно защищенных операций чтения с сериализуемой изоляцией, поскольку они могут вызвать проблемы с производительностью.

Например, сериализуемый курсор, последовательно считывающий каждую пару ключ/данные в базе данных, получит блокировку чтения на большинстве страниц базы данных и, таким образом, будет постепенно блокировать все операции записи в базах данных до тех пор, пока транзакция не зафиксируется или не прервется.

Следует обратить внимание, что если в приложении присутствуют транзакции обновления, операции чтения по-прежнему должны использовать блокировку и должны быть готовы к повторению любой операции (возможно, закрытия и повторного открытия курсора), которая завершается неудачей с возвращаемым значением DB_LOCK_DEADLOCK.

Приложения, которым требуется повторяющееся чтение, — это приложения, которым требуется возможность многократного доступа к элементу данных, зная, что он не изменится (например, операция, изменяющая элемент данных на основе его существующего значения).

Изоляция моментальных снимков (Snapshot Isolation) — это второй механизм обеспечения сериализуемости, в котором предпочтение отдается читателю, а не средствам обновления.

При использовании многоверсионного управления параллелизмом (MVCC) вместо того, чтобы требовать от читателей блокировки чтения при просмотре страницы, именно программы обновлений выполняют большую часть работы по поддержке изоляции степени 3. Это делает операции обновления более затратными, поскольку им приходится выделять место для новых версий страниц в кэше и делать копии, но отказ от блокировок чтения может значительно повысить пропускную способность для многих приложений.

Изоляция моментальных снимков подробно обсуждается ниже.

Транзакция может требовать только стабильности курсора, то есть гарантии того, что курсоры видят зафиксированные данные, которые не изменяются, пока к ним этот курсор обращается, но могут измениться до завершения транзакции чтения. Это также называется изоляцией ***второй степени***.

Berkeley DB обеспечивает этот уровень изоляции, когда транзакция запускается с флагом **DB_READ_COMMITTED**.

Этот флаг также можно указать при открытии курсора в полностью изолированной транзакции.

Berkeley DB дополнительно поддерживает чтение незафиксированных данных – то есть операции чтения могут запрашивать данные, которые были изменены, но еще не зафиксированы другой транзакцией. Это также называется изоляцией ***первой степени***.

Это делается путем указания флага **DB_READ_UNCOMMITTED** при открытии базовой базы данных, а затем указания флага **DB_READ_UNCOMMITTED** при старте транзакции, открытии курсора или выполнении операции чтения. Преимущество использования **DB_READ_UNCOMMITTED** заключается в том, что операции чтения не будут блокироваться, когда другая транзакция удерживает блокировку записи запрошенных данных. Недостатком является то, что операции чтения могут возвращать данные, которые исчезнут, если транзакция, удерживающая блокировку записи, прервется.

Изоляция моментальных снимков (Snapshot Isolation)

Изоляция моментальных снимков происходит, когда к базам данных с поддержкой многоверсионного управления параллелизмом (MVCC) обращаются транзакции **DB_TXN_SNAPSHOT**. Приложение сначала включает **DB_MULTIVERSION** при открытии баз данных, для которых требуется изоляция моментальных снимков.

Приложение завершает настройку изоляции моментальных снимков, включая **DB_TXN_SNAPSHOT** при начале транзакции или курсора.

Обычно изоляция моментальных снимков настраивается для каждой базы данных и транзакции, и только там, где это необходимо. Тем не менее, иногда полезно установить изоляцию моментального снимка для всего дескриптора среды:

```
dbenv->set_flags(dbenv, DB_MULTIVERSION, 1);
dbenv->set_flags(dbenv, DB_TXN_SNAPSHOT, 1);
```

Это может быть полезно для клиентов репликации, которые разгружают запросы на чтение с главного сайта.

При настройке среды для изоляции моментальных снимков важно понимать, что наличие нескольких версий страниц в кеше означает, что рабочий набор будет занимать большую часть кеша. В результате изоляция моментальных снимков лучше всего подходит для использования с кэшами большего размера.

Если кэш заполняется копиями страниц до того, как старые копии могут быть удалены, произойдет дополнительный ввод-вывод, поскольку страницы записываются во временные файлы «заморозки». Это может существенно снизить пропускную способность, и по возможности этого следует избегать, настраивая большой кэш и сокращая транзакции изоляции моментальных снимков.

Объем кэша, необходимый для предотвращения замораживания буферов, можно оценить, выполнив контрольную точку, за которой следует вызов **DB_ENV->log_archive()**. Требуемый объем кэша примерно вдвое превышает размер оставшихся журналов.

Окружение также следует настроить для достаточного количества транзакций с помощью **DB_ENV->set_tx_max()**. Максимальное количество транзакций должно включать все транзакции, выполняемые приложением одновременно, а также все курсоры, настроенные для изоляции моментальных снимков. Кроме того, транзакции сохраняются до тех пор, пока последняя созданная ими страница не будет удалена из кеша, поэтому в крайнем случае для каждой страницы в кеше может потребоваться дополнительная транзакция. Обратите внимание, что размеры кэша менее 500 МБ увеличиваются на 25 %, поэтому при расчете количества страниц это необходимо учитывать.

Так когда же приложениям следует использовать изоляцию моментальных снимков?

- имеется большой кэш относительно размера обновлений, выполняемых одновременными транзакциями;
- конкуренция операций чтения/записи ограничивает пропускную способность приложения;
- приложение полностью или по большей части только читает.

Самый простой способ воспользоваться изоляцией моментальных снимков для запросов состоит в сохранении транзакции обновления, используя полную блокировку чтения/записи, и установлении флага **DB_TXN_SNAPSHOT** для транзакций или курсоров, доступных только для чтения.

Это должно свести к минимуму блокировку транзакций изоляции моментальных снимков и позволит избежать возникновения новых ошибок **DB_LOCK_DEADLOCK**.

Если в приложении есть транзакции обновления, которые считывают множество элементов и обновляют только небольшой набор (например, сканирование до тех пор, пока не будет найдена нужная запись, а затем ее изменение), пропускную способность можно повысить, запустив некоторые обновления в режиме изоляции моментальных снимков.

DB_NEXT_NODUP (DB_PREV_NODUP)

Если курсор еще не инициализирован, **DB_NEXT_NODUP** (**DB_PREV_NODUP**) идентичен **DB_FIRST** (**DB_LAST**).

В противном случае курсор перемещается на следующий (предыдущий) *неповторяющийся* ключ базы данных, и эта пара ключ/данные возвращается.

Если база данных представляет собой базу данных Queue или Recno, **DBcursor->get()** с флагом **DB_NEXT_NODUP** (**DB_PREV_NODUP**) будет игнорировать любые ключи, которые существуют, но никогда не были явно созданы приложением, или те, которые были созданы и позже удалены.

Если установлен **DB_NEXT_NODUP** (**DB_PREV_NODUP**) и после (перед) позиции курсора в базе данных не существует неповторяющихся пар ключ/данные, **DBcursor->get()** вернет **DB_NOTFOUND**.

При использовании базы данных кучи этот флаг идентичен флагу **DB_NEXT** (**DB_PREV**).

Создание функций сравнения

Функция сравнения ключей BTree устанавливается с помощью **DB->set_bt_compare()**.
Функция сравнения дубликатов BTree устанавливается с помощью **DB->set_dup_compare()**.

```
#include <db.h>
                                // база данных
int DB->set_bt_compare(DB *db, int (*bt_compare_fcn)(DB *db,          // база данных
                                                     const DBT *dbt1, // Appl key
                                                     const DBT *dbt2, // Btree key
                                                     size_t *locp));  // NULL
```

Устанавливает функцию сравнения ключей Btree. Она вызывается всякий раз, когда необходимо сравнить ключ, указанный приложением, с ключом, который в данный момент хранится в дереве.

Если функция сравнения не указана, ключи сравниваются лексически, причем более короткие ключи располагаются перед более длинными ключами.

Метод **DB->set_bt_compare()** настраивает операции, выполняемые с использованием указанного дескриптора базы данных, а не все операции, выполняемые в основной базе данных.

DB->set_bt_compare() возвращает ненулевое значение ошибки при неудаче и 0 при успехе.

1) Нельзя использовать эти методы после открытия базы данных.

2) Если база данных уже существует при ее открытии, функции, предоставлямые этими методами, должна быть такими же, как те, что «исторически» использовались для создания базы данных, иначе может произойти повреждение.

Функция **bt_compare_fcn** — это функция сравнения Btree, определяемая приложением. Она имеет следующую сигнатуру:

```
int (*function)(DB *db, const DBT *key1, const DBT *key2, size_t *locp)
```

db – вложенный дескриптор базы данных.
dbt1 – DBT, представляющий ключ, предоставленный приложением.
dbt2 – DBT, представляющий ключ текущего дерева.
locp – параметр locp в настоящее время не используется и должен быть установлен в значение NULL, иначе может произойти повреждение.

Если считается, что **key1** < **key2**, то функция должна возвращать отрицательное значение.
Если считается, что **key1** = **key2**, то функция должна возвращать 0.
Если считается, что **key1** > **key2**, то функция должна возвращать значение больше 0.

Пример процедуры, которая используется для сортировки целочисленных ключей в базе данных:

```
int compare_int(DB *dbp, const DBT *a, const DBT *b, size_t *locp) {

    int ai, bi;
    locp = NULL;
    memcpy(&ai, a->data, sizeof(int));
    memcpy(&bi, b->data, sizeof(int));
    return (ai – bi);
}
```

Следует обратить внимание, что данные должны быть сначала скопированы в память, которая соответствующим образом выровнена, поскольку Berkeley DB не гарантирует никакого выравнивания данных, в том числе для процедур сравнения. При написании процедур сравнения следует помнить, что базы данных, созданные на машинах с разной архитектурой, могут иметь разный порядок байтов целых чисел, который код приложения должен компенсировать.

Заставляем базу данных использовать функцию сравнения

```
#include <db.h>
#include <string.h>
...
DB  *dbp;
int  ret;

// Создаем базу данных
ret = db_create(&dbp, NULL, 0);
if (ret != 0) {
    fprintf(stderr, "%s: %s\n", "my_program",
    db_strerror(ret));
    return(-1);
}
// Настройка функции сравнения btree для этой базы данных
dbp->set_bt_compare(dbp, compare_int);

// теперь можно эту базу данных открыть
...
```

Функция, предоставлямая **set_dup_compare()**, работает точно так же, за исключением того, что параметры DBT содержат элементы данных записи вместо ключей.

```
int DB->set_dup_compare(DB *db, int (*dup_compare_fcn)(DB *db,
                                                       const DBT *dbt1,
                                                       const DBT *dbt2,
                                                       size_t *locp));
```