Базы данных

Лекция 09 – Berkeley DB. Transactions. Concurency.

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

2024.11.18

Оглавление

4. Параллелизм (Concurrency)

DB предлагает большую поддержку для многопоточных и многопроцессных приложений, даже если транзакции не используются.

Многие из дескрипторов DB являются потокобезопасными либо могут быть сделаны потокобезопасными, если во время создания дескриптора будет предоставлен соответствующий флаг. В этом случае DB предоставляет гибкую подсистему блокировки для управления базами данных в параллельном приложении.

Кроме того, DB предоставляет надежный механизм для обнаружения и реагирования на взаимоблокировки.

Некоторые термины

Поток управления

Это поток, который выполняет какие-то операции в приложении. В данном контексте этот поток будет выполнять операции с БД.

Этот термин также может означать отдельный процесс, который выполняет операции — BDB поддерживает многопроцессные операции в базах данных.

BDB не зависит от типа или стиля потоков, используемых в приложении.

Поэтому, если для выполнения параллельного доступа к базе данных используется несколько потоков (в отличие от нескольких процессов), можно использовать любой пакет потоков, который лучше всего подходит для вашей платформы и приложения.

Тем не менее, рекомендуется использовать pthreads, поскольку они имеют наилучшие шансы на поддержку в большом диапазоне платформ.

Блокировка[1] – Когда поток управления получает доступ к общему ресурсу, говорят, что он блокирует этот ресурс.

BDB поддерживает как исключительные, так и неисключительные блокировки[2].

Свободнопоточный[3] – Структуры данных и объекты являются *свободнопоточными*, если они могут совместно использоваться потоками управления без какой-либо явной блокировки со стороны приложения. Некоторые книги, библиотеки и языки программирования могут использовать термин потокобезопасность для структур данных или объектов, которые имеют эту характеристику. Эти два термина означают одно и то же.

Заблокировано[4] – Когда поток не может получить блокировку, потому что какой-то другой поток уже удерживает блокировку на этом объекте, попытка получить блокировку считается заблокированной.

Взаимная блокировка[5] – Происходит, когда два или более потоков управления пытаются получить доступ к конфликтующему ресурсу таким образом, что ни один из потоков больше не может двигаться дальше.

Например, если поток A заблокирован в ожидании ресурса, удерживаемого потоком B, и в то же время поток B заблокирован в ожидании ресурса, удерживаемого потоком A, то ни один поток не может двигаться дальше. В этой ситуации поток A и поток B называются взаимоблокированными.

---

1) Lock
2) exclusive and non-exclusive locks
3) Free-threaded
4) Blocked
5) Deadlock

Какие дескрипторы БД являются потокобезопасными

Ниже описывается, в какой степени и при каких условиях отдельные дескрипторы являются потокобезопасными.

DB_ENV

Потокобезопасные, если методу open( ) окружения предоставляется флаг DB_THREAD.

DB

Потокобезопасные, если методу open( ) базы данных предоставляется флаг DB_THREAD или если база данных открыта с использованием потокобезопасного дескриптора окружения.

DBC

Курсоры не являются потокобезопасными. Однако они могут использоваться несколькими потоками управления, если приложение сериализует доступ к дескриптору.

DB_TXN

Доступ должен быть сериализован между потоками управления.

Блокировки, блоки и взаимоблокировки

Важно понять, как работает блокировка в параллельном приложении, прежде чем продолжить описание механизмов параллелизма, которые предоставляет DB.

Блокировка и взаимоблокировка имеют важные последствия для производительности приложения. Следовательно, в этом разделе дается фундаментальное описание этих концепций и того, как они влияют на операции DB.

Блокировки

Когда один поток управления хочет получить доступ к объекту, он запрашивает блокировку этого объекта. Эта блокировка позволяет DB предоставлять приложению гарантии транзакционной изоляции:

- ни один другой поток управления не может прочитать этот объект (в случае исключительной блокировки) и
- ни один другой поток управления не может изменить этот объект (в случае исключительной или неисключительной блокировки).

Ресурсы блокировки

Когда происходит блокировка, концептуально используются три ресурса:

1. Блокировщик[6];
2. Блокировка;
3. Заблокированный объект.

---

6) Locker

Блокировщик – это то, что удерживает блокировку. В транзакционном приложении блокировщик — это дескриптор транзакции. Для нетранзакционных операций блокировщик — это курсор или дескриптор БД.

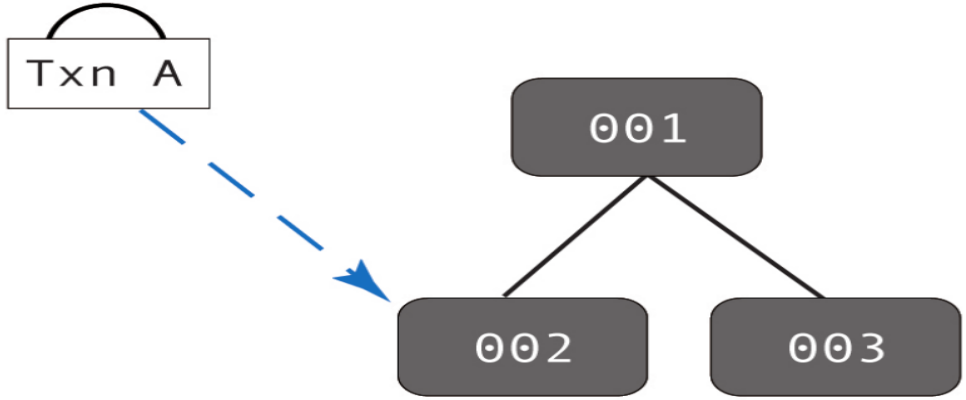
Блокировка — это фактическая структура данных, с помощью которой менеджер блокировок BDB блокирует объект.

Заблокированный объект – это именно то, что приложение хочет заблокировать.

В приложении БД заблокированный объект обычно представляет собой страницу базы данных, которая, в свою очередь, содержит несколько записей базы данных (ключ и данные). Однако для баз данных Queue блокируются отдельные записи базы данных.

Количество блокировщиков, блокировок и заблокированных объектов которое разрешено поддерживать приложению, можно настроить.

На следующем рисунке показан дескриптор транзакции Txn A, который удерживает блокировку на странице базы данных 002. На этом рисунке Txn A — это блокировщик, а заблокированный объект — страница 002. В этой операции используется только одна блокировка.



Типы блокировок

Приложения БД поддерживают как исключительные, так и неисключительные блокировки.

Исключительные блокировки предоставляются, когда блокировщик хочет записать в объект. По этой причине исключительные блокировки также иногда называют блокировками записи.

Исключительная блокировка не позволяет любому другому блокировщику получить какую-либо блокировку на объекте.

Это обеспечивает изоляцию, гарантируя, что никакой другой блокировщик не сможет наблюдать или изменять исключительно заблокированный объект, пока блокировщик не закончит запись в этот объект.

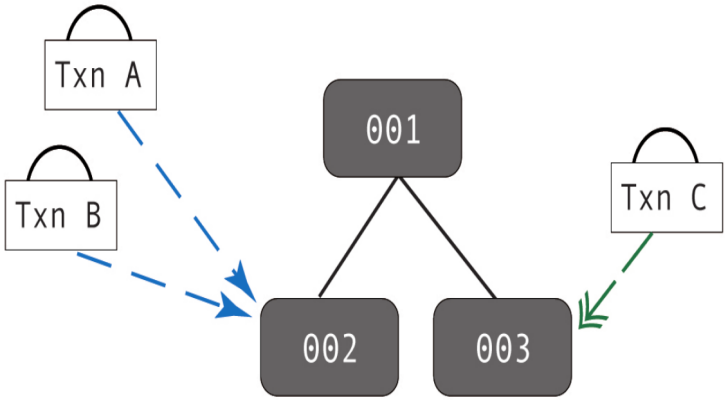
Неисключительные (совместные) блокировки предоставляются для доступа только для чтения.

По этой причине неисключительные блокировки также иногда называют блокировками чтения. Поскольку несколько блокировщиков могут одновременно удерживать блокировки чтения на одном и том же объекте, блокировки чтения также иногда называют общими или совместными блокировками.

Совместная блокировка не позволяет любому другому блокировщику изменять заблокированный объект, пока блокировщик все еще читает объект.

Именно таким образом транзакционные курсоры могут выполнять повторяющиеся чтения – по умолчанию транзакция курсора удерживает блокировку чтения на любом объекте, который проверил курсор, до тех пор, пока транзакция не будет зафиксирована или прервана. Можно избежать этих блокировок чтения, используя изоляцию моментального снимка.

На следующем рисунке Txn A и Txn B удерживают блокировку чтения на странице 002, в то время как Txn C удерживает блокировку записи на странице 003:



Продолжительность существования блокировки

Блокировщик удерживает свои блокировки до тех пор, пока блокировка больше не понадобится. Это означает следующее:

1. Транзакция удерживает все блокировки, которые она получает, пока транзакция не будет зафиксирована или отменена.

2. Все нетранзакционные операции удерживают блокировки до тех пор, пока операция не будет завершена.

Для курсор-операций блокировка удерживается до тех пор, пока курсор не будет перемещен в новую позицию или закрыт.

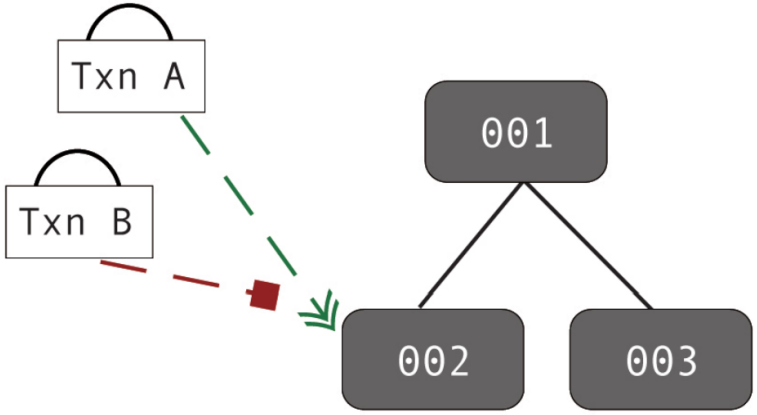
Блоки

Поток управления блокируется, когда он пытается получить блокировку, но эта попытка отклоняется, потому что какой-то другой поток управления удерживает конфликтующую блокировку.

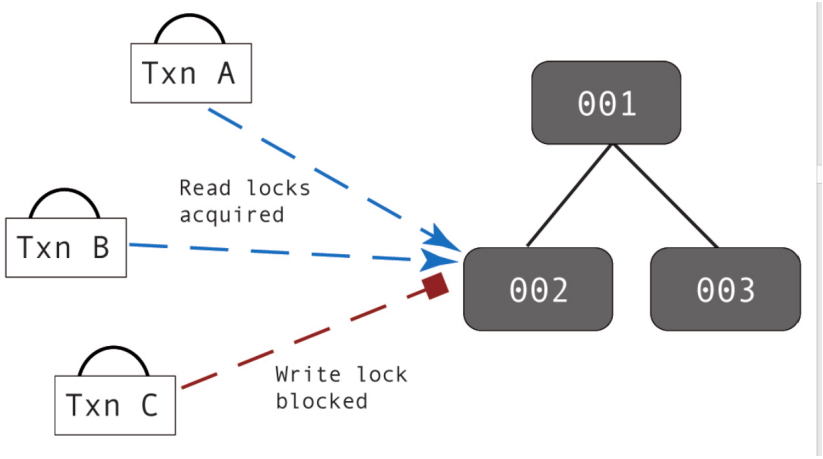
После заблокирования поток управления временно не может двигаться вперед, пока не будет получена запрошенная блокировка или операция, запрашивающая блокировку, не будет отменена.

Когда мы говорим о блокировании, строго говоря, поток не является тем, что пытается получить блокировку. Скорее, получить блокировку пытается какой-то объект внутри потока, например, курсор. Однако, как только блокировщик пытается получить блокировку, приостановливается весь поток управления до тех пор, пока запрос на блокировку не будет каким-то образом разрешен.

Например, если Txn A удерживает блокировку записи (исключительную блокировку) на объекте 002, то если Txn B попытается получить блокировку чтения или записи на этом объекте, поток управления, в котором выполняется Txn B, блокируется:



Однако если Txn A удерживает только блокировку чтения (совместную блокировку) на объекте 002, то будут блокироваться только те дескрипторы, которые пытаются получить блокировку записи на этом объекте.



Примечание

Предыдущее описание описывает поведение DB по умолчанию, когда она не может получить блокировку.

Можно настроить транзакции DB так, чтобы они не блокировались. Вместо этого, если блокировка недоступна, приложение немедленно уведомляется о ситуации взаимоблокировки.

Блокировка и производительность приложений

Многопоточные и многопроцессорные приложения обычно работают лучше, чем простые однопоточные приложения, потому что приложение может выполнять одну часть своей рабочей нагрузки (например, обновление записи базы данных), ожидая завершения какой-либо другой длительной операции (например, выполнение дискового или сетевого ввода-вывода).

Это улучшение производительности особенно заметно, если используется оборудование, предлагающее несколько ЦП, поскольку в таком случае потоки и процессы могут работать одновременно.

При этом параллельные приложения могут видеть снижение пропускной способности рабочей нагрузки, если их потоки управления сталкиваются с большим количеством конфликтов блокировки. То есть, если потоки блокируются при запросах блокировки, то это представляет собой потерю производительности для вашего приложения.

На предыдущей диаграмме Txn C не может получить запрошенную блокировку записи, потому что Txn A и Txn B уже удерживают блокировки чтения на запрошенном объекте. В этом случае поток, в котором работает Txn C, приостановится до тех пор, пока Txn C не получит блокировку записи или пока операция, запрашивающая блокировку, не будет отменена. Тот факт, что поток Txn C временно остановил весь прогресс вперед, представляет собой потерю производительности для вашего приложения.

Более того, любые блокировки чтения, запрошенные, пока Txn C ожидает своей блокировки записи, также будут блокироваться до тех пор, пока Txn C не получит и впоследствии не снимет свою блокировку записи.

Кк избежать блокировок

Сокращение числа конфликтов блокировки является важной частью настройки производительности параллельного приложения БД.

Приложения, которые имеют несколько потоков управления, получающих исключительные (записывающие) блокировки, подвержены проблемам с конкуренцией.

Более того, по мере увеличения количества блокировщиков и времени удержания блокировки увеличивается вероятность того, что приложение столкнется с конкуренцией блокировок.

При проектировании приложения, чтобы уменьшить конфликты блокировок, по возможности следует делать следующее:

1) Сокращать время, в течение которого приложение удерживает блокировки.

Более короткие транзакции приведут к более короткому времени жизни блокировок, что, в свою очередь, поможет уменьшить конфликты.

Кроме того, по умолчанию транзакционные курсоры удерживают блокировки чтения до тех пор, пока транзакция не будет завершена. По этой причине следует минимизировать время, в течение которого транзакционные курсоры держатся открытыми, или уменьшить уровни изоляции.

2) Если возможно, необходимо получать доступ к часто используемым (чтение или запись) элементам ближе к концу транзакции. Это сокращает время, в течение которого часто используемая страница блокируется транзакцией.

3)Уменьшать гарантии изоляции приложения.

Уменьшая гарантии изоляции, сокращаются ситуации, в которых блокировка может блокировать другую блокировку. Чтобы предотвратить блокирование чтения блокировкой записи можно для операций чтения использовать незафиксированные чтения.

Кроме того, для курсоров можно использовать изоляцию степени 2 (read commited), которая заставляет курсор снимать блокировки чтения сразу после завершения чтения записи (в отличие от удержания блокировок чтения до завершения транзакции).

Однако, следует помнить, что снижение гарантий изоляции может иметь неблагоприятные последствия для приложения. Перед тем как принять решение об уменьшении изоляции, следует внимательно изучить требования к изоляции приложения.

4) Для потоков только для чтения следует использовать изоляцию моментального снимка (snapshot isolation).

Изоляция моментального снимка заставляет транзакцию делать копию страницы, на которой она удерживает блокировку. Когда читатель делает копию страницы, для исходной страницы все еще могут быть получены блокировки записи. Это полностью устраняет конкуренцию чтения-записи.

5) Следует внимательно рассмотреть паттерны доступа к данным.

В зависимости от характера приложения, это может быть тем, с чем ничего не возможно сделать. Однако, если возможно организовать потоки таким образом, чтобы они работали только с неперекрывающимися частями базы данных, то можете существенно уменьшить конкуренцию за блокировки, поскольку потоки будут редко (если вообще будут) блокироваться на блокировках друг друга.

Примечание

Можно настроить транзакции БД так, чтобы они никогда не блокировались ожидая получения блокировки. Вместо этого они могут уведомить приложение о взаимоблокировке.

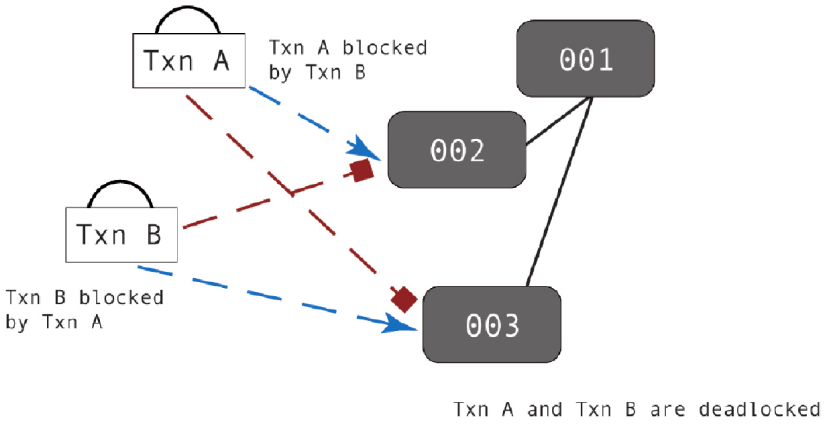
Такое поведение может быть настроено.

Взаимоблокировки

Взаимоблокировка происходит, когда два или более потоков управления блокируются, каждый из которых ожидает ресурс, удерживаемый другим потоком.

Когда это происходит, нет никакой возможности для потоков когда-либо продолжить работу, если только какой-либо внешний агент не предпримет действия для выхода из взаимоблокировки.

Например, если Txn A блокируется Txn B, в то же время Txn B блокируется Txn A, то потоки управления, содержащие Txn A и Txn B, блокируются; ни один поток не может продолжить работу, потому что ни один поток никогда не снимет блокировку, которая блокирует другой поток.



Когда два потока управления взаимоблокируются, единственным решением является наличие внешнего по отношению к двум потокам механизма, способного распознавать взаимоблокировку и уведомлять по крайней мере один поток о том, что он находится в ситуации взаимоблокировки.

После уведомления, чтобы разрешить взаимоблокировку, поток управления должен отказаться от предпринятой операции.

Подсистема блокировки DB предлагает механизм уведомления о взаимоблокировке.

Следует обратить внимание, что когда один блокировщик в потоке управления блокируется, ожидая блокировки, удерживаемой другим блокировщиком в том же потоке управления, поток считается самоблокирующимся.

Как избежать взаимоблокировки

То, что делается для избежания конфликта блокировок, также помогает уменьшить и взаимоблокировки.

Помимо этого, чтобы избежать взаимоблокировок, можно сделать следующее:

1) Никогда не следует иметь более одной активной транзакции в потоке одновременно. Распространенной причиной такого поведения является то, что поток использует автоматическую фиксацию для одной операции, в то время как в это же самое время в этом потоке используетсяявная транзакция.

2) Убедитесь, что все потоки получают доступ к данным в том же порядке, что и все другие потоки. Пока потоки блокируют страницы базы данных в том же базовом порядке, нет возможности взаимоблокировки (однако потоки все еще могут блокироваться).

Следует иметь в виду, что если используются вторичные базы данных (индексы), невозможно получить блокировки в согласованном порядке, поскольку невозможно предсказать порядок получения блокировок во вторичных базах данных.

Если параллельное приложение использут вторичные базы данных, нужно быть готовыми обрабатывать взаимоблокировки.

3) Если используется BTrees, в которых постоянно добавляются и затем удаляются данные, следует отключить механизм Btree reverse split (обратное расщепление Btree???).

4) Следует объявить блокировку чтения/изменения/записи для тех ситуаций, когда вы читаете запись, готовясь к изменению и последующей записи. В этом случае BDB назначает для операции чтения блокировку записи. В результате никакой другой поток управления не может использовать совместную блокировку чтения (что может вызвать конфликт при записи), но это также означает, что потоку записи не придется ждать получения блокировки записи, когда он будет готов записать измененные данные обратно в базу данных.

Подсистема блокировки

Чтобы разрешить параллельные операции, DB предоставляет подсистему блокировки. Эта подсистема обеспечивает меж- и внутрипроцессные механизмы параллелизма.

Она широко используется параллельными приложениями DB, но ее также можно использовать для ресурсов, не относящихся к DB.

Здесь описывается подсистема блокировки, используемая для защиты ресурсов DB.

Настройка подсистемы блокировки

Подсистема блокировки инициализируется при указании DB_INIT_LOCK в методе `DB_ENV->open()`.

Перед открытием окружения для подсистемы блокировки можно настроить различные значения. Все ограничения можно настроить только до открытия окружения.

Кроме того, эти методы настраивают все окружение, а не только какой-то отдельный дескриптор окружения.

Ограничения, которые можно настроить, следующие:

1) Количество блокировщиков, поддерживаемых окружением.

По умолчанию поддерживается 1000 блокировщиков.

Это значение используется окружением при ее открытии для оценки объема пространства, которое она должна выделить для различных внутренних структур данных.

Для настройки этого значения, используется метод `DB_ENV->set_memory_init()`, который позволяет настроить структуру DB_MEM_LOCKER.

В качестве альтернативы этому методу можете настроить это значение с помощью параметра `set_lk_max_lockers` файла DB_CONFIG.

2) Количество блокировок, поддерживаемых средой.

По умолчанию поддерживается 1000 блокировок.

Чтобы настроить это значение, используется `DB_ENV->set_memory_init()` для настройки структуры DB_MEM_LOCK.

В качестве альтернативы этому методу вы можете настроить это значение с помощью параметра `set_lk_max_locks` файла DB_CONFIG.

3) Количество заблокированных объектов, поддерживаемых средой.

По умолчанию можно заблокировать 1000 объектов.

Чтобы настроить это значение, используется метод `DB_ENV->set_memory_init()` для настройки структуры DB_MEM_LOCKOBJECT.

В качестве альтернативы этому методу вы можете настроить это значение с помощью параметра `set_lk_max_objects` файла DB_CONFIG.

Пример

Настройки количества блокировок, которые может использовать окружение:

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

int main(void) {

    int         ret, ret_c;
    u_int32_t   env_flags;
    DB_ENV     *envp = NULL;
    const char *db_home_dir = "/tmp/myEnvironment";
```

```
// Открываем окружение. Создаем дескриптор.
ret = db_env_create(&envp, 0);
if (ret != 0) {
    fprintf(stderr, "Error creating environment handle: %s\n",
    db_strerror(ret));
    return (EXIT_FAILURE);
}
// Флаги для окружения
env_flags = DB_CREATE |     // Создать окружение, если не существует
            DB_INIT_LOCK |  // Инициализация блокировок
            DB_INIT_LOG |   // Инициализация журналирования
            DB_INIT_MPOOL | // Инициализация кеширования в памяти
            DB_THREAD |     // Free-thread the env handle
            DB_INIT_TXN;    // Инициализация транзакций

// Настраиваем количество блокировок
ret = envp->set_memory_init(envp, DB_MEM_LOCK, 5000);
if (ret != 0) {
    fprintf(stderr, "Error configuring locks: %s\n",
    db_strerror(ret));
    goto err;
}
```

```
    // Открываем окружение
    ret = envp->open(envp, db_home_dir, env_flags, 0);
    if (ret != 0) {
        fprintf(stderr, "Error opening environment: %s\n",
        db_strerror(ret));
        goto err;
    }
err:
    // Закрываем окружение
    if (envp != NULL) {
        ret_c = envp->close(envp, 0);
        if (ret_c != 0) {
            fprintf(stderr, "environment close failed: %s\n",
            db_strerror(ret_c));
            ret = ret_c;
        }
    }
    return (ret == 0 ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

Настройка обнаружения взаимоблокировок

Чтобы база данных знала о возникновении взаимоблокировки, необходимо использовать некоторый механизм для обнаружения взаимоблокировок. Существует три способа обнаружения взаимоблокировок:

1) Разрешить базе данных внутренне обнаруживать взаимоблокировки по мере их возникновения. Для этого используется `DB_ENV->set_lk_detect()`.

Этот метод заставляет DB проходить по своей внутренней таблице блокировок в поисках взаимоблокировки всякий раз, когда блокируется запрос, и решать, какие запросы на блокировку отклоняются при обнаружении взаимоблокировок.

Например, DB может решить отклонить запрос на блокировку для транзакции, которая имеет наибольшее количество блокировок, наименьшее количество блокировок, удерживает самую старую блокировку, удерживает наибольшее количество блокировок записи и т. д.

Политики обнаружителя блокировок

```
DB_LOCK_NORUN      0
DB_LOCK_DEFAULT    1   /* Default policy. */
DB_LOCK_EXPIRE     2   /* Only expire locks, no detection. */
DB_LOCK_MAXLOCKS 3   /* Select locker with max locks. */
DB_LOCK_MAXWRITE 4   /* Select locker with max writelocks. */
DB_LOCK_MINLOCKS  5   /* Select locker with min locks. */
DB_LOCK_MINWRITE  6   /* Select locker with min writelocks. */
DB_LOCK_OLDEST    7   /* Select oldest locker. */
DB_LOCK_RANDOM    8   /* Select random locker. */
DB_LOCK_YOUNGEST 9   /* Select youngest locker. */
```

Этот метод может быть вызван в любое время в течение жизненного цикла приложения, но обычно он используется до того, как вы окружение будет открыто.

Следует заметить, что указание, как DB должна решать, какой поток управления должен разорвать взаимоблокировку, в значительной степени зависит от характера приложения.

Часто требуется некоторое тестирование производительности для принятия такого решения. Тем не менее, транзакция, которая удерживает наибольшее количество блокировок, обычно выполняет наибольший объем работы. Соответственно, нет смысла в том, чтобы транзакция, которая выполнила большой объем работы, отказалась от своих усилий и начала все заново. Поэтому обычно для разрыва взаимоблокировки выбирается транзакция с минимальным количеством блокировок записи.

2) Для обнаружения взаимоблокировок следует использовать выделенный поток или внешний процесс.

Этот поток не должен выполнять никаких других операций с базой данных, кроме обнаружения взаимоблокировок. Для внешнего обнаружения блокировок можно использовать метод `DB_ENV->lock_detect()`, либо использовать утилиту командной строки `db_deadlock`.

Метод или команда заставляет DB проходить таблицу блокировок в поисках взаимоблокировок.

Приложения, которые выполняют обнаружение взаимоблокировок таким образом, обычно запускают обнаружение взаимоблокировок каждые несколько секунд или минуту.

Это означает, что приложению, возможно, придется ждать уведомления о взаимоблокировке, но при этом экономятся накладные расходы на обход таблицы блокировок каждый раз, когда запрос на блокировку блокируется.

3) Тайм-ауты блокировки.

Можно настроить подсистему блокировки таким образом, чтобы она разблокировала любую блокировку, которая не была снята в течение указанного периода времени.

Для этого следует использовать метод `DB_ENV->set_timeout()`.

Тайм-ауты блокировки проверяются только тогда, когда блокируется запрос на блокировку или когда иным образом выполняется обнаружение взаимоблокировки.

Таким образом, блокировка может быть разаблокирована по тайм-ауту, но все еще удерживаться в течение некоторого периода времени, пока у базы данных не появится причина проверить свои таблицы блокировки.

При этом чрезвычайно долгоживущие транзакции или операции, которые удерживают блокировки в течение длительного времени, могут быть ненадлежащим образом завершены до того, как транзакция или операция получит шанс завершиться.

Поэтому вам следует использовать этот механизм только в том случае, если известно, что приложение будет удерживать блокировки в течение очень коротких периодов времени.

Следующий вызов командной строки запускает обнаружение взаимоблокировки для среды, содержащейся в `/export/dbenv`. Для прерывания взаимоблокировки выбирается транзакция с самой молодой блокировкой

```
$ /usr/local/db_install/bin/db_deadlock -h /export/dbenv -a y
```

Пример

Настройка приложения таким образом, чтобы БД проверяла таблицу блокировок на наличие взаимоблокировок каждый раз, когда блокируется запрос на блокировку:

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

int main(void) {

    int        ret, ret_c;
    u_int32_t db_flags, env_flags;
    DB     *dbp  = NULL;
    DB_ENV *envp = NULL;
    DB_TXN *txn;

    const char *db_home_dir = "/tmp/myEnvironment";
    const char *file_name = "mydb.db";

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

```
    env_flags = DB_CREATE |     // Создать окружение, если не существует
                DB_INIT_LOCK |  // Инициализация блокировок
                DB_INIT_LOG |   // Инициализация журналирования
                DB_INIT_MPOOL | // Инициализация кеширования в памяти
                DB_THREAD |     // Free-thread the env handle
                DB_INIT_TXN;    // Инициализация транзакций
// Настройка базы данных для внутреннего обнаружения взаимоблокировок и выбора
// транзакции, выполнившей наименьший объем записи, чтобы разорвать
// взаимоблокировку в случае ее обнаружения.
    ret = envp->set_lk_detect(envp, DB_LOCK_MINWRITE);
    if (ret != 0) {
        fprintf(stderr, "Error setting lk detect: %s\n",
        db_strerror(ret));
        goto err;
    }
    ret = envp->open(envp, db_home_dir, env_flags, 0);
    if (ret != 0) {
        fprintf(stderr, "Error opening environment: %s\n",
        db_strerror(ret));
        goto err;
    }

// Далее открывается базы данных, выполняются операции с базами данных,
// выполняются реакции на взаимоблокировки (опущено для краткости).
    ...
```

Разрешение взаимоблокировок

Когда DB определяет, что произошла взаимоблокировка, она выбирает поток управления для разрешения взаимоблокировки, а затем возвращает DB_LOCK_DEADLOCK этому потоку.

Если обнаружена взаимоблокировка, поток должен:

1) Прекратить все операции чтения и записи.
2) Закрыть все открытые курсоры.
3) Прервать транзакцию.
4) При необходимости повторить операцию.

Если приложение повторяет операции взаимоблокировки, новая попытка должна быть сделана с использованием новой транзакции.

Примечание

Если поток заблокировался, он не может выполнять никаких дополнительных вызовов базы данных с использованием дескриптора, который заблокировался.

Пример

```
    retry:
    ret = envp->txn_begin(envp, NULL, &txn, 0);
    if (ret != 0) {
        envp->err(envp, ret, "txn_begin failed");
        return (EXIT_FAILURE);
    }
    ...
    /* key and data are Dbts. Their usage is omitted for brevity. */
    ...
    switch (ret = dbp->put(dbp, txn, &key, &data, 0)) {
    case 0:
        break;
    case DB_LOCK_DEADLOCK:     // Deadlock handling goes here
        (void)txn->abort(txn); // Abort the transaction

// retry_count – это счетчик, используемый для определения того, сколько раз
// операциz повторялась. Чтобы избежать потенциального бесконечного
// зацикливания, повторять более MAX_DEADLOCK_RETRIES раз не стоит.

        if (retry_count < MAX_DEADLOCK_RETRIES) {
            printf("Got DB_LOCK_DEADLOCK.\n");
            printf("Retrying write operation.\n");
            retry_count++;
            goto retry;
        }
```

```
        printf("Got DB_LOCK_DEADLOCK and out of retries.");
        printf("Giving up.\n");
        return (EXIT_FAILURE);
    default:
        /* If some random database error occurs, we just give up */
        envp->err(envp, ret, "db put failed");
        ret = txn->abort(txn);
        if (ret != 0) {
            envp->err(envp, ret, "txn abort failed");
            return (EXIT_FAILURE);
        }
    }
    /* If all goes well, commit the transaction */
    ret = txn->commit(txn, 0);
    if (ret != 0) {
        envp->err(envp, ret, "txn commit failed");
        return (EXIT_FAILURE);
    }
    return (EXIT_SUCCESS);
```

Установка приоритетов транзакций

Обычно, когда необходимо выбрать поток управления для разрешения взаимоблокировки, БД решает, какой поток это сделает. Заранее узнать, какой поток будет выбран для разрешения взаимоблокировки, нет возможности.

Однако могут быть ситуации, когда программист знает, что лучше разрушить взаимоблокировку одним потоком, чем другим.

Например, если есть фоновый поток, выполняющий действия по управлению данными, и другой поток, отвечающий на запросы пользователей, естественно желать, чтобы разрешение взаимоблокировки происходило в фоновом потоке, поскольку это менее всего повлияет на пропускную способность приложения.

В этих обстоятельствах можно определить, какой поток управления будет выбран для разрешения взаимоблокировок, установив приоритеты транзакций.

Когда две транзакции взаимоблокируются, DB прервет транзакцию с самым низким приоритетом.

По умолчанию каждой транзакции присваивается приоритет 100.

Однако, используя метод `DB_TXN->set_priority()`, можно установить свой приоритет для каждой транзакции.

Когда две или более транзакции имеют одинаковый самый низкий приоритет, взаимоблокировка разрывается на основе политики, предоставленной параметру `atype` метода `DB_ENV->lock_detect()`.

Приоритет транзакции может быть изменен в любое время после создания дескриптора транзакции и до того, как транзакция будет разрешена (зафиксирована или отменена).

Пример

```
#include <stdio.h>
#include <stdlib.h>
#include "db.h"
int main(void) {

    int       ret, ret_c;
    u_int32_t db_flags, env_flags;
    DB     *dbp;
    DB_ENV *envp;
    DBT     key, data;
    DB_TXN *txn;
    ...
// Open the environment and database as normal.
// Omitted for brevity
    ...
// Get the txn handle

    txn = NULL;
    ret = envp->txn_begin(envp, NULL, &txn, 0);
    if (ret != 0) {
        envp->err(envp, ret, "Transaction begin failed.");
        goto err;
    }
    ret = txn->set_priority(txn, 200);
```

```
    if (ret != 0) {
        envp->err(envp, ret, "Transaction set_priority failed.");
        goto err;
    }
// Perform the database write. If this fails, abort the transaction.
    ret = dbp->put(dbp, txn, &key, &data, 0);
    if (ret != 0) {
        envp->err(envp, ret, "Database put failed.");
        txn->abort(txn);
        goto err;
    }
// Commit the transaction.
    ret = txn->commit(txn, 0);
    if (ret != 0) {
        envp->err(envp, ret, "Transaction commit failed.");
        goto err;
    }
    Isolation
        err:
    ...
// Close the database and environment here, and exit the application.
// Omitted for brevity.
}
```

Изоляция

Гарантии изоляции являются важным аспектом защиты транзакций. Транзакции гарантируют, что данные, с которыми работает транзакция, не будут изменены какой-либо другой транзакцией. Более того, изменения, внесенные транзакцией, никогда не будут видны за пределами этой транзакции, пока изменения не будут зафиксированы.

Тем не менее, существуют различные степени изоляции, и можно ослабить гарантии изоляции в той или иной степени в зависимости от требований приложения.

Основная причина, по которой имеет смысл это сделать, — это производительность. Чем большая изоляция предоставлена транзакциям, тем больше блокировок будет выполнять приложение. С большим количеством блокировок увеличивается вероятность активных блокировок, что, в свою очередь, заставляет потоки останавливаться в ожидании получения своей блокировки. Таким образом, ослабляя гарантии изоляции, потенциально можно улучшить пропускную способность приложения.

Поддерживаемые степени изоляции

BDB поддерживает следующие уровни изоляции (в терминах ANSI):

READ UNCOMMITTED – незафиксированные чтения
READ COMMITTED – изоляция зафиксированного чтения
SERIALIZABLE – сериализуемая изоляция

READ UNCOMMITTED – незафиксированные чтения

Незафиксированные чтения означают, что транзакция никогда не перезапишет грязные данные другой транзакции.

Грязные данные — это данные, которые транзакция изменила, но еще не зафиксировала в базовом хранилище данных. Однако незафиксированные чтения позволяют транзакции видеть данные, загрязненные другой транзакцией. Кроме того, транзакция может читать данные, загрязненные другой транзакцией, но которые впоследствии могут быть прерваны этой другой транзакцией.

В этом последнем случае транзакция чтения может читать данные, которые на самом деле никогда не существовали в базе данных.

READ COMMITTED – изоляция зафиксированного чтения

Изоляция зафиксированного чтения означает, что соблюдается степень изоляции 1, за исключением того, что грязные данные никогда не читаются.

Кроме того, этот уровень изоляции гарантирует, что данные никогда не изменятся, пока к ним обращается курсор, но данные могут измениться до того, как курсор чтения будет закрыт. В случае транзакции с данным уровнем данные в текущей позиции курсора не изменятся, но после перемещения курсора предыдущие данные, на которые ссылаются, могут измениться.

Это означает, что читатели снимут блокировки чтения до того, как курсор будет закрыт, и, следовательно, до того, как транзакция завершится.

Этот уровень изоляции заставляет курсор работать точно так же, как и при отсутствии транзакции.

SERIALIZABLE – сериализуемая изоляция

Имеет место зафиксированное чтение, плюс данные, считанные транзакцией T, никогда не будут загрязнены другой транзакцией до завершения T.

Это означает, что блокировки чтения и записи не снимаются до завершения транзакции.

Кроме того, ни одна транзакция не увидит фантомов. Фантомы — это записи, возвращенные в результате поиска, но которые не были видны той же транзакцией, когда ранее использовались идентичные критерии поиска.

Это гарантия изоляции DB по умолчанию.

По умолчанию транзакции BDB и транзакционные курсоры предлагают сериализуемую изоляцию. Можно снизить уровень изоляции, настроив БД на использование изоляции незафиксированного чтения.

В дополнение к обычным степеням изоляции БД можно использовать изоляцию моментального снимка.

Этот тип изоляции позволяет избежать блокировок чтения, которые требуются в случае сериализуемой изоляции.

Чтение незафиксированных данных (READ UNCOMMITED)

Berkeley DB позволяет настроить приложение для чтения данных, которые были изменены, но еще не зафиксированы другой транзакцией; то есть грязных данных.

Если это сделать, можно увидеть выигрыш в производительности, поскольку приложение не будет блокироваться в ожидании блокировок записи.

С другой стороны, данные, которые считывает приложение, могут измениться до завершения транзакции.

При использовании с транзакциями незафиксированные чтения означают, что одна транзакция может видеть данные, измененные, но еще не зафиксированные другой транзакцией.

При использовании с транзакционными курсорами незафиксированные чтения означают, что любой читатель базы данных может видеть данные, измененные курсором, до того, как транзакция курсора будет зафиксирована.

Из-за этого незафиксированные чтения позволяют транзакции читать данные, которые впоследствии могут быть прерваны другой транзакцией. В этом случае транзакция чтения будет иметь прочитанные данные, которые никогда на самом деле не существовали в базе данных.

Чтобы настроить приложение для чтения незафиксированных данных, следует:

1) База данных открывается так, чтобы она допускала незафиксированные чтения. Это можно сделать, указав DB_READ_UNCOMMITTED при открытии базы данных.

2) Следует указать DB_READ_UNCOMMITTED при создании транзакции, открытии курсора или чтении записи из базы данных.

Следующий код открывает базу данных таким образом, что она поддерживает незафиксированные чтения, а затем создает транзакцию, которая заставляет все чтения, выполняемые в ней, использовать незафиксированные чтения.

!!! ВАЖНО !!! Просто открыть базу данных для поддержки незафиксированных чтений недостаточно – также необходимо объявить, что операции чтения будут выполняться с использованием режима незафиксированных чтений.

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

int main(void) {

    int ret, ret_c;
    u_int32_t db_flags, env_flags;
    DB     *dbp  = NULL;
    DB_ENV *envp = NULL;
    DB_TXN *txn;

    const char *db_home_dir = "/tmp/myEnvironment";
    const char *file_name = "mydb.db";
    const char *keystr = "thekey";
    const char *datastr = "thedata";
```

```
// Open the environment
    ret = db_env_create(&envp, 0);
    if (ret != 0) {
        fprintf(stderr, "Error creating environment handle: %s\n",
                db_strerror(ret));
        return (EXIT_FAILURE);
    }
    env_flags = DB_CREATE |     // Создать окружение, если не существует
                DB_INIT_LOCK |  // Инициализация блокировок
                DB_INIT_LOG |   // Инициализация журналирования
                DB_INIT_MPOOL | // Инициализация кеширования в памяти
                DB_THREAD |     // Free-thread the env handle
                DB_INIT_TXN;    // Инициализация транзакций

    ret = envp->open(envp, db_home_dir, env_flags, 0);
    if (ret != 0) {
        fprintf(stderr, "Error opening environment: %s\n",
                db_strerror(ret));
        goto err;
    }

// Инициализация дескриптора базы данных
    ret = db_create(&dbp, envp, 0);
    if (ret != 0) {
        envp->err(envp, ret, "Database creation failed");
        goto err;
    }
```

```
    db_flags = DB_CREATE |          // Создать окружение, если не существует
               DB_AUTO_COMMIT |     // Разрешить автофиксацию
               DB_READ_UNCOMMITTED; // Разрешить незафиксированное чтение

    ret = dbp->open(dbp,        // Указатель на базу данных
                    NULL,       // Указатель на транзакцию
                    file_name, // Имя файла
                    NULL,       // Логическое имя базы данных
                    DB_BTREE,  // Тип базы данных (using btree)
                    db_flags,  // Флаги открытия
                    0);        // Режим файла (using defaults)
    if (ret != 0) {
        envp->err(envp, ret, "Database '%s' open failed",
                  file_name);
        goto err;
    }

// Получаем дескриптор транзакции
    txn = NULL;
    ret = envp->txn_begin(envp, NULL, &txn, DB_READ_UNCOMMITTED);
    if (ret != 0) {
        envp->err(envp, ret, "Transaction begin failed.");
        goto err;
    }
// Далее обычные операции, фиксируя и отменяя транзакции по мере необходимости,
// и тестирования для исключения взаимоблокировок (опущено для краткости).
...
```

Зафиксированные чтения (READ COMMITED)

Можно настроить транзакцию так, чтобы данные, считываемые транзакционным курсором, пока к ним тот обращается, были согласованными.

Однако после того, как курсор закончит чтение записи (то есть чтение записей со страницы, которую он в данный момент заблокировал), курсор снимает блокировку с этой записи или страницы.

Это означает, что данные, которые курсор прочитал и освободил, могут измениться до завершения транзакции курсора.

Например, предположим, что есть две транзакции, Ta и Tb.

Предположим далее, что у Ta есть курсор, который считывает запись R, не изменяя ее.

Tb не сможет записать запись R, потому что Ta будет удерживать блокировку чтения на ней. Но если транзакция настроена для зафиксированных чтений, Tb может изменять запись R до завершения Ta, если курсор чтения больше не обращается к записи или странице.

Когда приложение настраивается для этого уровня изоляции, можно добиться лучшей производительности, потому что транзакции удерживают меньше блокировок чтения.

Изоляция READ COMMITED наиболее полезна, когда у вас есть курсор, который читает и/или пишет записи в одном направлении, и которому никогда не нужно возвращаться, для повторного чтения тех же самых записей. В этом случае можно разрешить DB снимать блокировки чтения по мере их выполнения, а не удерживать их в течение всего срока действия транзакции.

Чтобы настроить приложение на использование подтвержденных чтений, выполните одно из следующих действий:

-1) Транзакция создается допускающей зафиксированные чтения.

Это можно сделать, указав DB_READ_COMMITTED при открытии транзакции.

2) При открытии курсора следует указать DB_READ_COMMITTED.

Пример

Следующий код создает транзакцию, которая допускает подтвержденные чтения:

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

int main(void) {

    int       ret, ret_c;
    u_int32_t db_flags, env_flags;

    DB     *dbp  = NULL;
    DB_ENV *envp = NULL;
    DB_TXN *txn  = NULL;

    const char *db_home_dir = "/tmp/myEnvironment";
    const char *file_name = "mydb.db";
```

```
    // Открываем окружение. Создаем дескриптор.
    ret = db_env_create(&envp, 0);
    if (ret != 0) {
        fprintf(stderr, "Error creating environment handle: %s\n",
        db_strerror(ret));
        return (EXIT_FAILURE);
    }

    env_flags = DB_CREATE |     // Создать окружение, если не существует
                DB_INIT_LOCK |  // Инициализация блокировок
                DB_INIT_LOG |   // Инициализация журналирования
                DB_INIT_MPOOL | // Инициализация кеширования в памяти
                DB_THREAD |     // Free-thread the env handle
                DB_INIT_TXN;    // Инициализация транзакций

    ret = envp->open(envp, db_home_dir, env_flags, 0);
    if (ret != 0) {
        fprintf(stderr, "Error opening environment: %s\n",
        db_strerror(ret));
        goto err;
    }
    // Инициализация дескриптора базы данных
    ret = db_create(&dbp, envp, 0);
    if (ret != 0) {
        envp->err(envp, ret, "Database creation failed");
        goto err;
    }
```

```
// Чтобы разрешить зафиксированные чтения не требуется указывать какие-либо
// флаги для базы данных (в отличие от незафиксированных чтений, для которых
// НУЖНО указывать соответствующий флаг при открытии базы данных).
    db_flags = DB_CREATE | DB_AUTO_COMMIT;
    ret = dbp->open(dbp,       // Указатель на базу данных
                    NULL,      // Указатель на транзакцию
                    file_name, // Имя файла
                    NULL,      // Логическое имя базы данных
                    DB_BTREE,  // Тип базы данных (using btree)
                    db_flags,  // Флаги открытия
                    0);        // Режим файла (using defaults)
    if (ret != 0) {
        envp->err(envp, ret, "Database '%s' open failed",
        file_name);
        goto err;
    }
    // Получаем дескриптор транзакции
    // Откываем транзакцию и включаем подтвержденные чтения. Все курсоры,
    // открытые с этим дескриптором транзакции, будут использовать изоляцию
    // подтвержденного чтения.
    ret = envp->txn_begin(envp, NULL, &txn, DB_READ_COMMITTED);
    if (ret != 0) {
        envp->err(envp, ret, "Transaction begin failed.");
        goto err;
    }
    // Далее как обычно, работаем ...
```

Использование изоляции моментальных снимков (Snapshot Isolation)

По умолчанию используется сериализуемая изоляция. Важным побочным эффектом этого уровня изоляции является то, что операции чтения получают блокировки чтения на страницах базы данных, а затем удерживают эти блокировки до тех пор, пока операция чтения не будет завершена.

При использовании транзакционных курсоров это означает, что блокировки чтения удерживаются до тех пор, пока транзакция не будет зафиксирована или прервана.

В этом случае со временем транзакционный курсор может постепенно заблокировать запись всех других транзакций в базу данных.

Избежать этого можно, используя изоляцию моментального снимка.

Изоляция моментального снимка использует управление многоверсионным параллелизмом для гарантии повторяющихся чтений. Это означает, что каждый раз, когда писатель захватывает на странице блокировку чтения, вместо этого создается копия страницы, и писатель работает с этой копией страницы. Это освобождает других писателей от блокировки из-за блокировки чтения, удерживаемой на странице.

Примечание

Изоляция моментального снимка настоятельно рекомендуется для потоков только для чтения в том случае, когда также запущены потоки записи, поскольку это устранит конкуренцию чтения-записи и значительно улучшит пропускную способность транзакций для потоков записи.

Однако для того, чтобы изоляция моментального снимка работала для потоков только для чтения, необходимо использовать транзакции для чтения базы данных.

Стоимость изоляции моментального снимка

Изоляция моментального снимка не обходится без затрат. Поскольку страницы дублируются перед обработкой, кэш будет заполняться быстрее. Это означает, что может потребоваться больший кэш, чтобы хранить весь рабочий набор в памяти.

Если кэш заполнится копиями страниц до того, как старые копии будут удалены, будут выполняться дополнительные операции ввода-вывода, поскольку страницы будут записываться во временные файлы для «заморозки» на диске. Это может существенно снизить пропускную способность, и этого следует избегать, если возможно, настраивая большой кэш и поддерживая короткие транзакции изоляции моментального снимка.

Можно оценить, насколько большим должен быть кэш, взяв контрольную точку, а затем вызвать метод `DB_ENV->log_archive()`. Требуемый объем кэша примерно вдвое превышает размер оставшихся файлов журнала (то есть файлов журнала, которые не могут быть заархивированы).

Требования к транзакциям изоляции моментальных снимков

Помимо увеличенного размера кэша, также может потребоваться увеличить количество транзакций, поддерживаемых приложением.

В худшем случае может потребоваться настроить приложение на одну дополнительную транзакцию для каждой страницы в кэше. Это связано с тем, что транзакции сохраняются до тех пор, пока последняя созданная ими страница не будет удалена из кэша.

Когда использовать изоляцию моментальных снимков

Изоляцию моментальных снимков лучше всего использовать, когда выполняются все или большинство из следующих условий:

1) Можно сделать большой кэш относительно размера рабочего набора данных.

2) Требуются повторяющиеся чтения.

3) Будут использоваться транзакции, которые регулярно работают со всей базой данных, или, что более распространено, в базе данных есть данные, которые будут очень часто записываться более чем одной транзакцией.

4) Конфликты чтения/записи ограничивают пропускную способность приложения, или приложение полностью или в основном только читает, а конкуренция за мьютекс менеджера блокировок ограничивает проихводительность.

Как использовать изоляцию моментальных снимков

Следующим образом:

1) База данных открывается с поддержкой многоверсионности.

Это можно настроить это либо при открытии среды, либо при открытии базы данных, используя флаг DB_MULTIVERSION для настройки этой поддержки.

2) Курсор или транзакция настраивается для использования изоляции моментальных снимков.

Для этого следует передать флаг DB_TXN_SNAPSHOT при открытии курсора или создании транзакции. Если настройка выполнена для транзакции, то этот флаг не требуется при открытии курсора.

Самый простой способ воспользоваться изоляцией моментального снимка:

- транзакции обновления должны использоваться с полной блокировкой чтения/записи
- транзакции или курсоры только для чтения должны использоваться с изоляцией моментального снимка.

Это должно минимизировать блокировку транзакций изоляции моментального снимка и позволит избежать ошибок взаимоблокировки.

Если приложение имеет транзакции обновления, которые считывают много элементов и обновляют только небольшой набор (например, сканирование до тех пор, пока не будет найдена нужная запись, а затем ее изменение), производительность можно улучшить, запустив некоторые обновления в режиме изоляции моментального снимка.

Однако, это означает, что возникнет необходимость управлять ошибками взаимоблокировки.

Пример

Изоляция моментального снимка для транзакции:

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

int main(void) {

    int       ret, ret_c;
    u_int32_t db_flags, env_flags;
    DB     *dbp  = NULL;
    DB_ENV *envp = NULL;
```

```
const char *db_home_dir = "/tmp/myEnvironment";
const char *file_name   = "mydb.db";

// Открываем окружение
ret = db_env_create(&envp, 0);
if (ret != 0) {
    fprintf(stderr, "Error creating environment handle: %s\n",
    db_strerror(ret));
    return (EXIT_FAILURE);
}
/*
 * Поддержка snapshot isolation
 */
envp->set_flags(envp, DB_MULTIVERSION, 1);
env_flags = DB_CREATE |     // Создать окружение, если не существует
            DB_INIT_LOCK |  // Инициализация блокировок
            DB_INIT_LOG |   // Инициализация журналирования
            DB_INIT_MPOOL | // Инициализация кеширования в памяти
            DB_INIT_TXN;    // Инициализация транзакций

// Инициализация транзакций
ret = envp->open(envp, db_home_dir, env_flags, 0);
if (ret != 0) {
    fprintf(stderr, "Error opening environment: %s\n",
    db_strerror(ret));
    goto err;
}
```

```
    // Инициализация  дескриптора базы данных
    ret = db_create(&dbp, envp, 0);
    if (ret != 0) {
        envp->err(envp, ret, "Database creation failed");
        goto err;
    }

// Для поддержки изоляции моментального снимка ничего не нужно предоставлять.
// Предоставляет окружение и для базы данных тоже.

    db_flags = DB_CREATE | DB_AUTO_COMMIT;
    ret = dbp->open(dbp,       // Указатель на базу данных
                    NULL,      // Указатель на транзакцию
                    file_name, // Имя файла
                    NULL,      // Логическое имя базы данных
                    DB_BTREE,  // Тип базы данных (using btree)
                    db_flags,  // Флаги открытия
                    0);        // Режим файла (using defaults)
    if (ret != 0) {
        envp->err(envp, ret, "Database '%s' open failed",
        file_name);
        goto err;
    }
    ....
    ret = envp->txn_begin(envp, NULL, &txn, DB_TXN_SNAPSHOT);
    // mice gnawed for brevity
```

Транзакционные курсоры и параллельные приложения

При использовании транзакционных курсоров с параллельным приложением следует помнить, что в случае возникновения взаимоблокировки необходимо убедиться закрыть курсор, прежде чем прервать и повторить транзакцию.

Также следует помнить, что при использовании уровня изоляции по умолчанию, каждый раз, когда ваш курсор читает запись, он блокирует эту запись до тех пор, пока не будет завершена охватывающая транзакция. Это означает, что обход вашей базы данных с помощью транзакционного курсора увеличивает вероятность конфликта блокировок.

По этой причине, если необходимо регулярно обходить базу данных с помощью транзакционного курсора, следует рассмотреть возможность использования пониженного уровня изоляции, например read commited.

Использование курсоров с незафиксированными данными

Чтобы транзакция могла читать данные, измененные, но еще не зафиксированные другой транзакцией, можно ослабить уровень изоляции своей транзакции.

Это можно настроить при создании дескриптора транзакции, и после того, как это будет сделано, все курсоры, открытые внутри этой транзакции, будут автоматически использовать незафиксированные чтения.

Также можно это сделать при создании дескриптора курсора из сериализуемой транзакции. После этого только те курсоры, которые настроены для незафиксированных чтений, используют незафиксированные чтения.

В любом случае, прежде чем настраивать транзакции или курсоры для использования незафиксированных чтений, необходимо сначала настроить дескриптор базы данных для поддержки незафиксированных чтений.

В следующем примере показано, как настроить отдельный дескриптор курсора для чтения незафиксированных данных из сериализуемой (полная изоляция) транзакции.

Пример настройки транзакции для выполнения незафиксированных чтений в целом см. в разделе Чтение незафиксированных данных (слайд 35).

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

int main(void) {

    DB     *dbp  = NULL;
    DB_ENV *envp = NULL;
    DB_TXN *txn  = NULL;
    DBC    *cursorp;
    int ret, c_ret;
    char *replacementString = "new string";

    // Открываем окружение
    ret = db_env_create(&envp, 0);
    if (ret != 0) {
        fprintf(stderr, "Error creating environment handle: %s\n",
        db_strerror(ret));
        return (EXIT_FAILURE);
    }
```

```
    env_flags = DB_CREATE |     // Создать окружение, если не существует
                DB_INIT_LOCK |  // Инициализация блокировок
                DB_INIT_LOG |   // Инициализация журналирования
                DB_INIT_MPOOL | // Инициализация кеширования в памяти
                DB_INIT_TXN;    // Инициализация транзакций

    ret = envp->open(envp, db_home_dir, env_flags, 0);
    if (ret != 0) {
        fprintf(stderr, "Error opening environment: %s\n",
        db_strerror(ret));
        goto err;
    }

    // Инициализация дескриптора базы данных
    ret = db_create(&dbp, envp, 0);
    if (ret != 0) {
        envp->err(envp, ret, "Database creation failed");
        goto err;
    }

    db_flags = DB_CREATE |          // Создать, базу данных если не существует
               DB_AUTO_COMMIT |     // Разрешить автофиксацию
               DB_READ_UNCOMMITTED; // Разрешить чтение незафиксированных
                                    // данных
```

```
    ret = dbp->open(dbp,        // Указатель на базу данных
                    NULL,       // Указатель на транзакцию
                    file_name, // Имя файла
                    NULL,       // Логическое имя базы данных
                    DB_BTREE,  // Тип базы данных (using btree)
                    db_flags,  // Флаги открытия
                    0);         // Режим файла (using defaults)

    if (ret != 0) {
        envp->err(envp, ret, "Database '%s' open failed",
        file_name);
        goto err;
    }

    // Получаем дескриптор транзакции (транзакция 3-й степени)

    ret = envp->txn_begin(envp, NULL, &txn, 0);
    if (ret != 0) {
        envp->err(envp, ret, "Transaction begin failed.");
        goto err;
    }

    // Получаем курсор, предоставляя дескриптор транзакции txn и
    // заставляем курсор выполнять незафиксированные чтения
    ret = dbp->cursor(dbp, txn, &cursorp, DB_READ_UNCOMMITTED);
```

```
    if (ret != 0) {
        envp->err(envp, ret, "Cursor open failed.");
        txn->abort(txn);
        goto err;
    }
// Далее чтение и запись курсора как обычно, фиксируя и отменяя транзакции
// по мере необходимости, а также проверяя как обычно взаимоблокировки
    ...
```

Эксклюзивные дескрипторы базы данных

В некоторых случаях параллельные приложения могут выиграть от периодического предоставления исключительного доступа ко всей базе данных одному дескриптору базы данных. Это желательно, когда поток будет выполнять операцию, которая затрагивает все или большинство страниц в базе данных.

Чтобы настроить дескриптор на исключительный доступ к базе данных, необходимо дать ему одну блокировку записи для всей базы данных. Это приводит к блокировке всех других потоков при их попытке получить блокировку чтения или записи для любой части этой базы данных.

Исключительная блокировка позволяет улучшить пропускную способность, поскольку дескриптор не будет пытаться получить какие-либо дополнительные блокировки после того, как он получит исключительную блокировку записи. Он также никогда не будет заблокирован в ожидании блокировки, и нет возможности цикла взаимоблокировка-повторная попытка. Чтобы настроить дескриптор базы данных с исключительной блокировкой, перед тем, как открыть дескриптор базы данных, используется метод `DB->set_lk_exclusive()`.

DB->set_lk_exclusive()

```
#include <db.h>

int DB->set_lk_exclusive(DB *db,             // дескриптор базы данных
                         int nowait_onoff); // режим вызова Wait/No Wait
```

Метод дает дескриптору исключительный доступ к базе данных, поскольку блокировка записи заблокирует все остальные потоки управления как для чтения, так и для записи.

Этот метод следует использовать для повышения производительности и пропускной способности базы данных для потока, который управляет этим дескриптором.

При настройке с помощью этого метода операции в базе данных не получают блокировок страниц, когда они выполняют операции чтения и/или записи. Кроме того, исключительная блокировка означает, что операции, выполняемые в дескрипторе базы данных, никогда не будут блокироваться в ожидании блокировки из-за действий другого потока.

Приложение также будет защищено от взаимоблокировок.

С другой стороны, использование этого метода означает, что пока дескриптор не закрыт можно иметь только один поток, обращающийся к базе данных,. Для некоторых приложений невозможность другим потокам одновременно работать с базой данных приведет к снижению производительности.

Для данного дескриптора одновременно может быть активна только одна транзакция.

Чтение/изменение/запись

Если запись извлекается из базы данных с целью ее изменения или удаления, следует объявить цикл «чтения-изменения-записи» во время чтения записи.

В результате во время чтения будет получена блокировка записи (вместо блокировки чтения). Это поможет предотвратить взаимоблокировки, не давая другой транзакции получить блокировку чтения для той же записи во время выполнения цикла чтения-изменения-записи.

Объявление цикла чтения-изменения-записи может фактически увеличить количество блокировок, которые видит приложение, поскольку читатели немедленно блокируются по записи, так как блокировки записи не могут быть общими.

По этой причине циклы чтения-изменения-записи следует использовать только в том случае, если в приложении происходит большое количество взаимоблокировок.

Чтобы объявить цикл чтения/изменения/записи при выполнении операции чтения, в метод получения базы данных или курсора следует передать флаг DB_RMW. Например:

```
retry:
// Получим транзакцию
ret = envp->txn_begin(envp, NULL, &txn, 0);
if (ret != 0) {
    envp->err(envp, ret, "txn_begin failed");
    return (EXIT_FAILURE);
}
...
// ключ и данные представлены как Dbt. Их использование опущено
...
```

```
// Получим данные. Объявим цикл read/modify/write
ret = dbp->get(dbp, txn, &key, &data, DB_RMW);
...
// Модифицируем ключ и данные как нам нужно (здесь не показано)
...
// Put the data. Note that you do not have to provide any additional flags here
due to the read/modify/write cycle. Simply put the data and perform your
deadlock detection as normal.

ret = dbp->put(dbp, txn, &key, &data, 0);
switch (ret) {
// Обнаружение взаимоблокировки опущено для краткости изложения
....
```

Без ожидания получения блокировки

Обычно, когда транзакция БД блокируется по запросу блокировки, прежде чем ее поток управления сможет продолжить работу, она будет ждать, пока запрошенная блокировка не станет доступной. Однако можно настроить дескриптор транзакции таким образом, чтобы он вместо ожидания сообщал о взаимоблокировке.

Это можно сделать для каждой транзакции отдельно, указав DB_TXN_NOWAIT в методе `DB_ENV->txn_begin( )`. Например:

```
// Получим транзакцию
DB_TXN *txn = NULL;
ret = envp->txn_begin(envp, NULL, &txn, DB_TXN_NOWAIT);
if (ret != 0) {
    envp->err(envp, ret, "txn_begin failed");
    return (EXIT_FAILURE);
}
....
// Обнаружение взаимоблокировки опущено для краткости изложения
```

Обратные разделения Btree (Reverse BTree Splits)

Если приложение использует метод доступа Btree и многократно удаляет и добавляет записи в базу данных, можно уменьшить количество конфликтов блокировок, отключив обратные разделения Btree.

По мере того, как страницы в базе данных опустошаются, DB пытается удалить пустые страницы, чтобы сохранить базу данных как можно меньше и минимизировать время поиска. Более того, когда страница в базе данных заполняется, DB, конечно же, добавляет дополнительные страницы, чтобы создать место для посткпающих данных.

Добавление и удаление страниц в базе данных требует, чтобы поток записи заблокировал родительскую страницу. Следовательно, по мере уменьшения количества страниц в базе данных, приложение будет видеть все больше конфликтов блокировки – максимальный уровень параллелизма в базе данных из двух страниц намного меньше, чем в базе данных из 100 страниц, поскольку в этом случае меньше страниц, которые могут быть заблокированы.

Таким образом, если не допускать сокращения базы данных до минимального количества страниц, можно улучшить пропускную способность параллельного доступа приложения. Однако следует отметить, что следует делать это только в том случае, если ваше приложение имеет тенденцию удалять и затем добавлять одни и те же (в смысле значения ключа) данные. В противном случае отключение Reverse BTree Splits может увеличить времена поиска в базе данных.

Reverse BTree Splits отключаются с помощью флага DB_REVSPLITOFF, указанного в методе DB->set_flags():

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

int main(void) {

    int       ret, ret_c;
    u_int32_t db_flags, env_flags;
    DB     *dbp  = NULL;
    DB_ENV *envp = NULL;

    const char *db_home_dir = "/tmp/myEnvironment";
    const char *file_name   = "mydb.db";

    // Открываем окружение
    ret = db_env_create(&envp, 0);
    if (ret != 0) {
        fprintf(stderr, "Error creating environment handle: %s\n",
        db_strerror(ret));
        return (EXIT_FAILURE);
    }
```

```
    env_flags = DB_CREATE |     // Создать окружение, если не существует
                DB_INIT_LOCK |  // Инициализация блокировок
                DB_INIT_LOG |   // Инициализация журналирования
                DB_INIT_MPOOL | // Инициализация кеширования в памяти
                DB_THREAD |     // Free-thread the env handle
                DB_INIT_TXN;    // Инициализация транзакций

    ret = envp->open(envp, db_home_dir, env_flags, 0);
    if (ret != 0) {
        fprintf(stderr, "Error opening environment: %s\n",
        db_strerror(ret));
        goto err;
    }

  // Инициализация дескриптора базы данных
    ret = db_create(&dbp, envp, 0);
    if (ret != 0) {
        envp->err(envp, ret, "Database creation failed");
        goto err;
    }

    // Отключаем btree reverse split
    ret = db->set_flags(&db, DB_REVSPLITOFF);
    if (ret != 0) {
        envp->err(envp, ret, "Turning off Btree reverse split failed");
        goto err;
    }
```

```
    db_flags = DB_CREATE | DB_AUTO_COMMIT;
    ret = dbp->open(dbp,        // Указатель на базу данных
                    NULL,       // Указатель на транзакцию
                    file_name, // Имя файла
                    NULL,       // Логическое имя базы данных
                    DB_BTREE,  // Тип базы данных (using btree)
                    db_flags,  // Флаги открытия
                    0);        // Режим файла (using defaults)

    if (ret != 0) {
        envp->err(envp, ret, "Database '%s' open failed",
        file_name);
        goto err;
    }
err:
    // Закрываем базу данных
    if (dbp != NULL) {
        ret_c = dbp->close(dbp, 0);
        if (ret_c != 0) {
            envp->err(envp, ret_c, "Database close failed.");
            ret = ret_c
        }
    }
```

```
    // Закрываем окружение
    if (envp != NULL) {
        ret_c = envp->close(envp, 0);
        if (ret_c != 0) {
            fprintf(stderr, "environment close failed: %s\n",
            db_strerror(ret_c));
            ret = ret_c;
        }
    }
    return (ret == 0 ? EXIT_SUCCESS : EXIT_FAILURE);
}
```