Базы данных

Лекция 08 – Berkeley DB. Транзакции

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

2024.11.11

Оглавление

Преимущества транзакций....3
Примечание о системных сбоях....5
Требования к приложениям....7
Многопоточные и многопроцессные приложения....9
Восстанавливаемость....10
Настройка производительности....11
Включение транзакций....12
Окружения....14
Именование файлов....15
Поддержка обработки ошибок....21
Совместно используемые области памяти....22
Вопросы безопасности....24
Открытие транзакционного окружения и базы данных....27
Основы транзакций....33
DB_ENV->txn_begin()....34
DB_TXN->commit() — завершить транзакцию....40
DB_TXN->abort() — аварийное завершение транзакции....41
Фиксация транзакции....47
Неустойчивые транзакции....50
Прерывание транзакции....51
Автоматическая фиксация....52
Вложенные транзакции....57
Транзакционные курсоры....59

Преимущества транзакций

Транзакции обеспечивают защиту данных от сбоев приложения или системы.
Транзакции berkeley DB предлагают приложению полную поддержку ACID:

Атомарность

Несколько операций с базой данных рассматриваются как неделимое действие.
После фиксации все операции записи, выполненные под защитой транзакции, сохраняются в базах данных. Кроме того, в случае прерывания транзакции все операции записи, выполненные во время транзакции, отменяются. В этом случае база данных остается в том состоянии, в котором она была до начала транзакции, независимо от количества или типа операций записи, которые могли быть выполнены в ходе транзакции.
ВАЖНО!!! — транзакции БД могут охватывать один или несколько дескрипторов базы данных.

Согласованность

Базы данных никогда не увидят частично завершенную транзакцию. Это верно даже в случае сбоя приложения во время выполнения транзакций. Если приложение или система выходят из строя, то либо все изменения базы данных появляются при следующем запуске приложения, либо ни одно из них не появляется.
Другими словами, какие бы требования к согласованности ни предъявляло приложение, они никогда не будут нарушены DB. Если, например, приложение требует, чтобы каждая запись включала идентификатор сотрудника, и код добавляет этот идентификатор в записи БД, то DB никогда не нарушит это требование согласованности. Идентификатор останется в записях базы данных до тех пор, пока само приложение не решит его удалить.

Изоляция

Во время выполнения транзакции базы данных будут отображаться для транзакции так, как будто нет других операций, происходящих вне транзакции. То есть операции, заключенные в транзакцию, всегда будут иметь чистый и согласованный вид баз данных.

Им никогда не придется видеть обновления, которые в настоящее время выполняются под защитой другой транзакции.

ВАЖНО!!! — гарантии изоляции могут быть ослаблены по сравнению с настройками по умолчанию.

Долговечность

После фиксации в базах данных изменения сохранятся даже в случае сбоя приложения или системы. Следует обратить внимание, что, как и в случае изоляции, гарантия долговечности может быть ослаблена.

Примечание о системных сбоях

Транзакции защищают данные от «сбоя системы или приложения». Это верно до определенной степени – не все сбои одинаковы, и ни один механизм защиты данных не может защитить от всех мыслимых способов, которыми может умереть вычислительная система.

В целом, когда речь идет о защите от сбоев, это означает, что транзакции обеспечивают защиту от наиболее вероятных виновников сбоев системы и приложений.

Если изменения данных были зафиксированы на диске, эти изменения должны сохраняться, даже если приложение или ОС впоследствии выйдут из строя.

Мало того, если приложение или ОС выйдут из строя в середине фиксации транзакции (или прервутся), данные на диске должны быть либо в согласованном состоянии, либо должно быть достаточно данных, чтобы привести базы данных в согласованное состояние (например, с помощью процедуры восстановления).

При этом можно потерять все данные, которые фиксировались во время сбоя, но базы данных в остальном затронуты не будут.

Примечание

Следует помнить, что многие диски имеют кэш записи на диск, а в некоторых системах он включен по умолчанию. Это означает, что транзакция может быть зафиксирована, и для приложения данные могут казаться находящимися на диске, но на самом деле данные могут находиться в это время только в кэше записи.

Это означает, что если кэш записи на диск включен и для него нет резервного питания, после сбоя ОС данные могут быть потеряны, даже если используется режим максимальной надежности.

Для максимальной надежности следует отключить кэш записи на диск или использовать кэш записи на диск с резервным питанием.

Если диск выйдет из строя, то транзакционные преимущества, описанные в этой книге, будут только настолько хороши, насколько хороши резервные копии, которые были сделаны. Распределяя данные и файлы журналов по отдельным дискам, можно минимизировать риск потери данных из-за отказа диска, но даже в этом случае можно создать сценарий, когда даже этой защиты будет недостаточно (например, пожар в машинном зале), и придется обратиться к резервным копиям для защиты.

Наконец, следуя примерам программирования, показанным в этой книге, можно написать свой код так, чтобы защитить свои данные в случае сбоя этого кода. Однако ни один программный API не может никого защитить от логических сбоев в прикладном коде; транзакции не могут защитить от простой записи в базы данных не того, что надо.

Требования к приложениям

Чтобы использовать транзакции, приложение должно удовлетворять определенным требованиям, выходящим за рамки требований к нетранзакционным защищенным приложениям. Это:

Использование откружений[1]

Окружения необязательны для нетранзакционных приложений, но они необходимы для транзакционных.

Включение подсистемы транзакций

Чтобы использовать транзакции, необюходимо явно включить транзакционную подсистему для приложения, и это должно быть сделано во время первого создания окружения.

Включение подсистемы ведения журнала

Подсистема ведения журнала необходима для восстановления, но ее использование также означает, что приложению может потребоваться немного больше административных усилий, чем при неиспользовании ведения журнала.

Дескрипторы DB_TXN

Чтобы получить гарантию атомарности, предлагаемую транзакционной подсистемой (то есть объединить несколько операций в одну единицу работы), приложение должно использовать дескрипторы транзакций. Эти дескрипторы получаются из объектов `DB_ENV`. Обычно они должны быть недолговечными, и их использование достаточно просто.

Чтобы завершить транзакцию и сохранить выполненную ею работу, следует вызвать ее метод `commit( )`.

---

1) Утмшкщтьутеы

Чтобы завершить транзакцию и отменить ее работу, следует вызвать ее метод `abort()`.

Кроме того, можно использовать автоматическую фиксацию, если необходимо транзакционно защитить одну операцию записи. Автоматическая фиксация позволяет использовать транзакцию без получения явного дескриптора транзакции.

Требования к открытию базы данных

Помимо использования окружений и инициализации правильных подсистем, приложение должно защищать транзакции при открытии базы данных и любых ассоциациях вторичных индексов, если последующие операции с базами данных должны быть защищены транзакциями. Открытие базы данных и ассоциация вторичных индексов обычно защищаются транзакциями с помощью автоматической фиксации .

Обнаружение взаимоблокировок[2]

Обычно транзакционные приложения при доступе к базе данных используют несколько потоков управления. Каждый раз, когда несколько потоков используются на одном ресурсе, возникает потенциальная возможность конфликта блокировок. В свою очередь, конфликт блокировок может привести к взаимоблокировкам. Поэтому транзакционные приложения должны часто включать код для обнаружения и реагирования на взаимоблокировки.

Следует обратить внимание, что это требование не является специфическим для транзакций — безусловно, можно писать параллельные нетранзакционные приложения БД. Кроме того, не каждое транзакционное приложение использует параллелизм, и поэтому не каждое транзакционное приложение должно управлять взаимоблокировками. Тем не менее, управление взаимоблокировками является настолько частой характеристикой транзакционных приложений, что оно обсуждается в этой книге.

---

2) Deadlock detection

Многопоточные и многопроцессные приложения

BDB разработана для поддержки многопоточных и многопроцессных приложений, однако их использование означает, что необходимо уделять особое внимание вопросам параллелизма. Транзакции помогают параллелизму приложения, предоставляя различные уровни изоляции для потоков управления. Кроме того, DB предоставляет механизмы, которые позволяют обнаруживать и реагировать на взаимоблокировки.

Изоляция означает, что изменения базы данных, внесенные одной транзакцией, обычно не будут видны читателям из другой транзакции до тех пор, пока первая не зафиксирует свои изменения. Разные потоки используют разные дескрипторы транзакций, поэтому этот механизм обычно используется для обеспечения изоляции между операциями базы данных, выполняемыми разными потоками.

Следует обратить внимание, что DB поддерживает разные уровни изоляции. Например, можно настроить приложение на просмотр незафиксированных чтений, что означает, что одна транзакция может видеть данные, которые были изменены, но еще не зафиксированы другой транзакцией. Это может означать, что транзакция считывает данные, «загрязненные» другой транзакцией, но которые впоследствии могут измениться до того, как эта другая транзакция зафиксирует свои изменения. С другой стороны, снижение требований к изоляции означает, что приложение может получить улучшенную пропускную способность из-за снижения конкуренции за блокировку.

Восстанавливаемость

Важной частью транзакционных гарантий DB является долговечность.

Долговечность означает, что после фиксации транзакции изменения базы данных, выполненные под ее защитой, не будут потеряны из-за сбоя системы.

Для обеспечения гарантии транзакционной долговечности DB использует систему опережающей записи в журналы. Каждая операция, выполненная в базах данных, описывается в журнале до ее выполнения в этих базах данных. Это делается для того, чтобы гарантировать, что операцию можно будет восстановить в случае несвоевременного сбоя приложения или системы.

Помимо регистрации, еще одним важным аспектом долговечности является восстанавливаемость. То есть резервное копирование и восстановление. DB поддерживает обычное восстановление, которое выполняется для подмножества файлов журнала.

Это обычная процедура, используемая всякий раз, когда впервые окружение открывается при запуске приложения, и она предназначена для обеспечения того, чтобы база данных находилась в согласованном состоянии.

DB также поддерживает архивное резервное копирование и восстановление в случае катастрофического сбоя, такого как потеря физического диска.

В этой книге описывается несколько различных процедур резервного копирования, которые можно использовать для защиты данных на диске. Эти процедуры варьируются от простых стратегий автономного резервного копирования до горячих отказов.

Горячие отказы обеспечивают не только механизм резервного копирования, но и способ восстановления после фатального сбоя оборудования.

В этой книге также описываются процедуры восстановления, которые следует использовать для каждой из стратегий резервного копирования, которые можно использовать.

Настройка производительности

С точки зрения производительности использование транзакций не является бесплатным. В зависимости от того, как они настроены, транзакционные фиксации обычно требуют, чтобы приложение выполняло дисковый ввод-вывод, который не транзакционное приложение не выполняет. Кроме того, для многопоточных и многопроцессных приложений использование транзакций может привести к увеличению конкуренции за блокировку из-за дополнительных требований к блокировке, обусловленных гарантиями изоляции транзакций.

Поэтому существует отдельный компонент настройки производительности для транзакционных приложений, который не применим для не транзакционных приложений (хотя некоторые соображения по настройке существуют независимо от того, использует ли приложение транзакции).

Включение транзакций

Чтобы использовать транзакции с приложением, необходимо их включить. Для этого следует:

Использовать окружение.

Включить транзакции для окружения. Это делается предоставлением флага DB_INIT_TXN методу `DB_ENV->open()`.

Следует обратить внимание, что инициализация транзакционной подсистемы подразумевает, что также инициализируется и подсистема ведения журнала.

Также следует обратить внимание, что если при первом создании окружения транзакции не инициализируются, то использовать транзакции для этого окружения после этого невозможно. Это связано с тем, что DB выделяет определенные структуры, необходимые для транзакционной блокировки, которые недоступны, если окружение создано без транзакционной поддержки.

Инициализировать кэш в памяти, передав флаг DB_INIT_MPOOL в `DB_ENV->open()`.

Инициализировать подсистему блокировки. Она обеспечивает блокировку для параллельных приложений. Она также используется для обнаружения взаимоблокировок.

Подсистема блокировки инициализируется флагом DB_INIT_LOCK в `DB_ENV->open()`.

Инициализировать подсистему ведения журнала. Хотя для транзакционных приложений это включено по умолчанию, предлагается для удобства чтения кода явно инициализировать ее в любом случае. Подсистема ведения журнала — это то, что обеспечивает транзакционному приложению гарантию долговечности, и она необходима для целей восстановления. Подсистема ведения журнала инициализируется флагом DB_INIT_LOG в методе `DB_ENV- >open()`.

Включить транзакции в базах данных. Если используется базовый API, следует включить транзакции в своих базах данных. Это делается путем инкапсуляции открытой базы данных в транзакцию.

Следует обратить внимание, что общепринятой практикой является использование автоматической фиксации для защиты открытой базы данных.

Чтобы использовать автоматическую фиксацию, включить транзакции все равно необходимо, однако явно использовать транзакцию при открытии базы данных не нужно.

Окружения

Для простых приложений БД окружения необязательны. Однако для защиты транзакций операций базы данных окружение использовать необходимо.

Окружение представляет собой инкапсуляцию одной или нескольких баз данных и любых связанных файлов журналов и областей общей памяти. Они используются для поддержки многопоточных и многопроцессорных приложений, позволяя различным потокам управления совместно использовать кэш в памяти, таблицы блокировки, подсистему журналирования и пространство имен файлов. Благодаря совместному использованию этих вещей параллельное приложение становится более эффективным, чем если бы каждый поток управления управлял этими ресурсами самостоятельно.

По умолчанию все базы данных баз данных поддерживаются файлами на диске. В дополнение к этим файлам транзакционные приложения баз данных создают журналы, которые также по умолчанию хранятся на диске (они могут быть дополнительно подкреплены с помощью общей памяти). Наконец, транзакционные приложения баз данных также создают и используют области общей памяти, которые также обычно поддерживаются файловой системой. Но, как и базы данных и журналы, области могут поддерживаться строго в памяти, если это требуется приложению.

Предупреждение

Использование окружений с некоторыми журналируемыми файловыми системами может привести к повреждению файла журнала. Это может произойти, если операционная система некорректно завершает работу при создании файла журнала.

Именование файлов

Для работы приложение БД должно иметь возможность находить свои файлы базы данных, файлы журналов и файлы областей. Если они хранятся в файловой системе, то необходимо сообщить БД, где они находятся (существует ряд механизмов, которые позволяют определить местоположение этих файлов — см. ниже).

В противном случае они находятся в текущем рабочем каталоге по умолчанию.

Указание домашнего каталога окружения

Домашний каталог окружения используется для определения того, где находятся файлы базы данных. Его местоположение определяется с помощью одного из следующих механизмов в следующем порядке приоритета:

1) Если не указано, где следует разместить домашний каталог окружения, то используется текущий рабочий каталог.

2) Если домашний каталог указан в методе `DB_ENV->open()`, то для домашнего каталога окружения всегда используется это местоположение.

3) Если домашний каталог не указан в `DB_ENV->open()`, но указаны флаги DB_USE_ENVIRON или DB_USE_ENVIRON_ROOT для метода DB_ENV->open(), то используется каталог, указанный переменной среды DB_HOME.

Оба флага позволяют указать путь к домашнему каталогу окружения с помощью переменной среды DB_HOME. Однако DB_USE_ENVIRON_ROOT учитывается только в том случае, если процесс запущен с правами root или администратора.

Указание местоположений файлов

По умолчанию все файлы DB создаются относительно домашнего каталога окружения. Например, предположим, что домашний каталог окружения находится в `/export/myAppHome`. Также предположим, что база данных имеет имя `data/myDatabase.db`. В этом случае база данных размещается в: `/export/myAppHome/data/myDatabase.db`.

При этом всегда имеется приоритет абсолютного пути. Это означает, что если указано абсолютное имя файла при указании имени базы данных, то этот файл не размещается относительно домашнего каталога окружения. Вместо этого он размещается в точном месте, которое указано для имени файла.

В системах UNIX абсолютный путь — это имя, начинающееся с косой черты ('/'). В системах Windows абсолютный путь — это имя, начинающееся с одного из следующих символов:

1) Обратный слеш ('\').

2) Любая буква алфавита, за которой следует двоеточие (':'), за которым следует обратный слеш ('\').

Примечание

Старайтесь не использовать абсолютные имена путей для файлов окружения.

В определенных сценариях восстановления абсолютные имена путей могут сделать окружение невосстанавливаемым.

Это происходит, если попытаться восстановить окружение в системе, которая не поддерживает абсолютный путь, который был использован.

Определение конкретных местоположений файлов

Как описано в предыдущих разделах, DB будет размещать все свои файлы в домашнем каталоге окружения или относительно него. Также можно разместить определенный файл базы данных в определенном месте, используя для его имени абсолютный путь. В этой ситуации домашний каталог окружения не учитывается при именовании файла.

Часто желательно размещать файлы базы данных, журнала и области на отдельных дисках. Распределяя ввод-вывод по нескольким дискам, можно повысить параллелизм и улучшить пропускную способность.

Кроме того, размещение файлов журнала и файлов баз данных на отдельных дисках, повышает надежность приложения, предоставляя ему больше шансов пережить сбой диска.

Файлы БД могут размещаться в разных местах, согласно следующим механизмам:

Тип файла	Как переопределить расположение

Файлы баз данных	Можно создать файлы базы данных в каталоге, отличном от домашнего каталога окружения, используя метод `DB_ENV->add_data_dir()`. Указанный каталог должен существовать. Если указан относительный путь, то каталог будет расположен относительно домашнего каталога окружения. Этот метод изменяет каталог, используемый для файлов базы данных, созданных и управляемых одним дескриптором окружения; он не настраивает всё окружения. Этот метод не может быть вызван после открытия окружения. Также можно задать местоположение данных по умолчанию, которое будет использоваться всем окружением, с помощью параметра `add_data_dir` в файле DB_CONFIG окружения. Параметр add_data_dir переопределяет любое значение, установленное методом `DB_ENV->add_data_dir()`.

Файлы журнала	Можно создать файлы журнала в каталоге, отличном от домашнего каталога окружения, с помощью метода DB_ENV->set_lg_dir(). Указанный здесь каталог должен существовать. Если указан относительный путь, то каталог будет расположен относительно домашнего каталога окружения. Метод изменяет каталог, используемый для файлов базы данных, созданных и управляемых одним дескриптором окружения; он не настраивает всё окружение. Этот метод нельзя вызывать после открытия окружения. Можно задать местоположение файла журнала по умолчанию, которое будет использоваться всем окружением, с помощью параметра set_lg_dir в файле DB_CONFIG окружения. Следует обратить внимание, что параметр set_lg_dir переопределяет любое значение, заданное методом DB_ENV->set_lg_dir().
Временные файлы	Можно создать временные файлы, требуемые окружением, в каталоге, отличном от домашнего каталога окружения, с помощью DB_ENV->set_tmp_dir(). Указанный здесь каталог должен существовать. Если указан относительный путь, то каталог будет расположен относительно домашнего каталога окружения. Также можно задать местоположение временного файла, используя параметр set_tmp_dir в файле DB_CONFIG окружения. Параметр set_tmp_dir переопределяет любое значение, установленное методом DB_ENV->set_tmp_dir().

Файлы метаданных	Можно создать постоянные файлы метаданных, необходимые приложениям репликации, в каталоге, отличном от домашнего каталога окружения, с помощью метода DB_ENV->set_metadata_dir(). Указанный здесь каталог должен существовать. Если указан относительный путь, то каталог будет расположен относительно домашнего каталога окружения. Можно задать местоположение каталога метаданных с помощью параметра set_metadata_dir в файле DB_CONFIG окружения. Параметр set_metadata_dir переопределяет любое значение, установленное методом DB_ENV->set_metadata_dir().
Файлы областей	Файлы областей, если поддерживаются файловой системой, всегда размещаются в домашнем каталоге окружения.

Следует обратить внимание, что файл DB_CONFIG должен находиться в домашнем каталоге окружения. Параметры указываются в нем по одному параметру на строку. После каждого параметра следует пробел, за которым следует значение параметра. Например:

```
add_data_dir /export1/db/env_data_files
```

Поддержка обработки ошибок

Для упрощения обработки ошибок и помощи в отладке приложений окружения предлагают несколько полезных методов. Многие из этих методов идентичны методам обработки ошибок, доступным для структуры DB. В частности, это:

`set_errcall()` – определяет функцию, которая вызывается, когда БД выдает сообщение об ошибке. Префикс ошибки и сообщение передаются в этот обратный вызов.

Приложение должно правильно отображать эту информацию.

Это рекомендуемый способ получения сообщений об ошибках из БД.

`set_errfile()` – устанавливает указатель на структуру FILE *, которая будет использоваться при отображении сообщений об ошибках, выдаваемых библиотекой DB.

`set_errpfx()` – устанавливает префикс, используемый для любых сообщений об ошибках, выдаваемых библиотекой DB.

`err()` – выдает сообщение об ошибке на основе кода ошибки DB и текста сообщения, который предоставляется. Сообщение об ошибке отправляется в функцию обратного вызова, которрая определена `set_errcall()`. Если этот метод не использовался, то сообщение об ошибке отправляется в файл, определенный `set_errfile()`. Если ни один из этих методов не использовался, то сообщение об ошибке отправляется в стандартный поток ошибок.

Сообщение об ошибке состоит из строки префикса (как определено `set_errprefix()`), необязательного сообщения в формате `printf()`, сообщения об ошибке DB, связанного с предоставленным кодом ошибки, и завершающего символа новой строки.

`errx()` -- ведет себя идентично err(), за исключением того, что вы не указываете код ошибки DB, поэтому текст сообщения DB не отображается.

Кроме того, можно использовать функцию `db_strerror()`.

Совместно используемые области памяти[3]

Подсистемы, которые включаютя для окружения (в нашем случае транзакции, ведение журнала, блокировка и пул памяти), описываются одной или несколькими областями.

Области содержат всю информацию о состоянии, которая должна совместно использоваться потоками и/или процессами, использующими окружение.

Области могут поддерживаться файловой системой, кучей памяти или системной совместно используемой памятью.

Области, поддерживаемые файлами

По умолчанию совместно используемые области памяти создаются как файлы в домашнем каталоге среды (не в каталоге данных окружения). Если доступен для отображения этих файлов в адресное пространство приложения интерфейс POSIX mmap(), он используется. Если mmap() недоступен, то вместо него используются интерфейсы UNIX shmget() (опять же, если они доступны).

В этом случае по умолчанию файлы областей называются `__db.###` (например, `__db.001`, `__db.002` и т. д.).

Области, поддерживаемые памятью кучи

Если для поддержки совместно используемых областей используется память кучи, то для окружения можно открыть только один дескриптор.

Это означает, что к окружению не могут получить доступ несколько процессов. В этом случае области управляются только в памяти и не записываются в файловую систему.

---

3) Shared Memory Regions

Чтобы указать, что для файлов областей должна использоваться память кучи, для метода DB_ENV->open() следует указать флаг DB_PRIVATE. Также можно установить этот флаг с помощью параметра set_open_flags в файле DB_CONFIG.

Области, поддерживаемые системной памятью

Можно для регионов вместо файлов, отображенных в память, использовать системную память. Это делается предоставлением методу `DB_ENV->open()` флага DB_SYSTEM_MEM .

Когда файлы областей поддерживаются системной памятью, DB создает один файл в домашнем каталоге. Этот файл содержит информацию, необходимую для идентификации системной общей памяти, используемой окружением. Создавая этот файл, DB позволяет нескольким процессам использовать окружение совместно.

Системная память, которая используется, зависит от архитектуры. Например, в системах, поддерживающих интерфейсы общей памяти в стиле X/Open, таких как системы UNIX, используются `shmget(2)` и связанные с ними интерфейсы System V IPC. Кроме того, системную память используют системы VxWorks.

В этих случаях начальный идентификатор сегмента должен быть указан приложением, чтобы гарантировать, что приложения не перезапишут окружения друг друга, так что количество созданных сегментов не будет расти без ограничений.

На платформах Windows использование системной памяти для файлов областей проблематично, поскольку операционная система использует подсчет ссылок для автоматической очистки общих объектов в файле подкачки. Кроме того, разрешения на доступ к общим объектам по умолчанию отличаются от разрешений на доступ к файлам, что может вызвать проблемы, когда к окружению обращаются несколько процессов, работающих от имени разных пользователей.

Вопросы безопасности

При использовании окружений следует учитывать некоторые вопросы безопасности:

Разрешения окружения базы данных[4]

Каталог, используемый для окружения, должен иметь свои специфические разрешения, установленные для обеспечения того, чтобы файлы в окружении не были доступны пользователям без соответствующих прав.

Приложения, которые добавляют права пользователям (например, приложения UNIX setuid или setgid), должны быть тщательно проверены, чтобы не допустить незаконного использования добавленных прав, таких как общий доступ к файлам в каталоге окружения.

Переменные среды

Установка флагов DB_USE_ENVIRON или DB_USE_ENVIRON_ROOT, чтобы при именовании файлов можно было использовать переменные среды, может быть опасным.

Установка этих флагов в приложениях БД с дополнительными разрешениями (например, приложения UNIX setuid или setgid) может потенциально позволить пользователям читать и записывать базы данных, к которым у них обычно нет доступа.

Например, предположим, что вы пишете приложение БД, которое запускает setuid. Это означает, что когда приложение запускается, оно делает это под идентификатором пользователя, отличным от идентификатора вызывающей стороны приложения.

Особенно проблематично, если приложение предоставляет пользователю более высокие привилегии, чем те, которые пользователь мог бы иметь обычно.

---

4) Database environment permissions

Теперь, если для окружения установлены флаги DB_USE_ENVIRON_ROOT или DB_USE_ENVIRON, то окружение, которое использует приложение, может быть изменено с помощью переменной среды DB_HOME. В этом сценарии, если uid, используемый приложением, имеет достаточно широкие привилегии, то вызывающий приложение может читать и/или записывать базы данных, принадлежащие другому пользователю, просто установив свою переменную среды DB_HOME на окружение, используемое другим пользователем.

Следует обратить внимание, что этот сценарий не обязательно должен быть вредоносным; приложение может использовать неправильное окружение, просто непреднамеренно указав неправильный путь к DB_HOME.

Как всегда, следует использовать setuid экономно, если вообще использовать. Но если setuid используется, то для открываемого окружения следует воздержаться от указания флагов DB_USE_ENVIRON или DB_USE_ENVIRON_ROOT.

И, конечно, если необходимо использовать setuid, то следует убедиться, что используется самый слабый uid из возможных — желательно тот, который используется только самим приложением.

Разрешения на доступ к файлам

По умолчанию DB всегда создает файлы базы данных и журнала, доступные для чтения и записи владельцу и группе (то есть S_IRUSR, S_IWUSR, S_IRGRP и S_IWGRP; или восьмеричный режим 0660 в исторических системах UNIX). Групповое владение созданными файлами основано на системных и каталоговых значениях по умолчанию и дополнительно не указывается DB.

Временные файлы резервного копирования

Если создается неименованная база данных, а кэш слишком мал для хранения базы данных в памяти, Berkeley DB создаст временный физический файл, чтобы иметь возможность выгружать базу данных на диск по мере необходимости. В этом случае для указания местоположения этого временного файла могут использоваться переменные среды, такие как TMPDIR.

Хотя временные файлы резервного копирования создаются доступными для чтения и записи только владельцу (S_IRUSR и S_IWUSR или восьмеричный режим 0600 в исторических системах UNIX), некоторые файловые системы могут недостаточно защищать от несанкционированного доступа временные файлы, созданные в случайных каталогах.

Для обеспечения абсолютной безопасности приложения, хранящие конфиденциальные данные в неименованных базах данных, для указания временного каталога с известными разрешениями должны использовать метод `DB_ENV->set_tmp_dir()`.

Открытие транзакционного окружения и базы данных

Чтобы включить транзакции для окружения, необходимо инициализировать транзакционную подсистему. Это также инициализирует и подсистему журналирования.

Кроме того, необходимо инициализировать пул памяти (кэш в памяти).

Также необходимо инициализировать подсистему блокировки.

В следующем примере перед открытием окружения создается дескриптор окружения с помощью функции `db_env_create( )`:

```
#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";

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

```
    env_flags = DB_CREATE |     // Создаем окружение если еще не существует
                DB_INIT_TXN |   // инициализация транзакции
                DB_INIT_LOCK | // инициализация блокировок
                DB_INIT_LOG |   // инициализация журналирования
                DB_INIT_MPOOL; // инициализация кэша в памяти

    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:
    // Close the environment
    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);
}
```

Затем создается и открывается база данных, как и в нетранзакционной системе.

Единственное отличие заключается в том, что необходимо передать дескриптор окружения в функцию db_create( ) и нужно открыть базу данных для использования транзакций.

Обычно для этой цели используется автоматическая фиксация. Для этого в команду открытия базы данных передается DB_AUTO_COMMIT.

Перед закрытием окружения рекомендуется закрыть все базы данных.

```
#include <stdlib.h>
#include <stdio.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 |     // Создаем окружение если еще не существует
            DB_INIT_TXN |   // инициализация транзакции
            DB_INIT_LOCK |  // инициализация блокировок
            DB_INIT_LOG |   // инициализация журналирования
            DB_INIT_MPOOL; // инициализация кэша в памяти

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 БД
ret = db_create(&dbp, envp, 0); // дескриптор окружения
if (ret != 0) {
    envp->err(envp, ret, "Database creation failed");
    goto err;
}
db_flags = DB_CREATE |      // Создаем если не существует
           DB_AUTO_COMMIT; // Автофиксация
ret = dbp->open(dbp,        // Указатель на БД
                NULL,       // Указатель транзакции
                file_name, // Имя файла
                NULL,       // Лигическое имя БД
                DB_BTREE,  // Тип БД (btree)
```

```
                    db_flags,  // Флаги открытия
                    0);        // Режим сознания файла (по умолчанию)
    if (ret != 0) {
        envp->err(envp, ret, "Database '%s' open failed",
        file_name);
        goto err;
    }
err:
    // Закрываем БД
    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);
}
```

Примечание

Никогда не следует закрывать базу данных, в которой есть активные транзакции.

Следует убедиться, что перед закрытием базы данных все транзакции завершены (зафиксированы или отменены).

Основы транзакций

После включения транзакций для окружения и баз данных можно их использовать для защиты операций в БД.

Сначала получаем дескриптор транзакции, а затем используем его для любой операции базы данных, которую следует включить в данную транзакцию.

Дескриптор транзакции получаем с помощью метода **`DB_ENV->txn_begin()`**.

После завершения всех операций, которые следует включить в транзакцию, необходимо зафиксировать транзакцию с помощью метода **`DB_TXN->commit()`**.

Если по какой-либо причине необходимо отказаться от транзакции, ее следует отменить с помощью **`DB_TXN->abort()`**.

Любой дескриптор транзакции, который был зафиксирован или прерван, больше не может использоваться приложением.

Перед закрытием баз данных и окружения следует убедиться, что все дескрипторы транзакций либо зафиксированы, либо прерваны.

Примечание

При необходимости защитить транзакцией только одну операцию записи в БД, можно использовать автоматическую фиксацию. При использовании автоматической фиксации явный дескриптор транзакции не требуется.

DB_ENV->txn_begin()

```
#include <db.h>

int DB_ENV->txn_begin(DB_ENV *env,      // окружение
                      DB_TXN *parent,   // вложенность транзакции
                      DB_TXN **tid,     // идентификатор транзакции
                      u_int32_t flags); //
```

Метод `DB_ENV->txn_begin()` создает новую транзакцию в окружении и копирует указатель на DB_TXN, который уникально идентифицирует транзакцию в память, на которую ссылается `tid`.

Вызов методов `DB_TXN->abort()`, `DB_TXN->commit()` или `DB_TXN->discard()` аннулирует возвращенный дескриптор.

Примечание

Транзакции могут охватывать потоки только в том случае, если они делают это последовательно, то есть каждая транзакция должна быть активна только в одном потоке управления в один момент времени.

Это ограничение также действует для родителей вложенных транзакций – никакие два потомка не могут быть одновременно активны в более чем одном потоке управления в любой момент времени.

Примечание

Курсоры не могут охватывать транзакции, то есть каждый курсор должен быть открыт и закрыт в рамках одной транзакции.

Примечание

Родительская транзакция не может выполнять никаких операций Berkeley DB — за исключением `DB_ENV->txn_begin()`, `DB_TXN->abort()` и `DB_TXN->commit()` пока у нее есть активные дочерние транзакции (дочерние транзакции, которые еще не были зафиксированы или прерваны).

Метод `DB_ENV->txn_begin()` возвращает ненулевое значение ошибки при неудаче и 0 при успешном выполнении.

Параметры

parent — если не равен NULL, новая транзакция будет вложенной транзакцией, при этом транзакция, указанная `parent`, является ее родителем.

Транзакции могут быть вложены на любом уровне. При наличии распределенных транзакций и двухфазной фиксации только родительская транзакция, то есть транзакция без указанного родителя, должна передаваться в качестве параметра в `DB_TXN->prepare()`.

flags — должен быть установлен в 0 или путем побитового включающего ИЛИ вместе для одного или нескольких из следующих значений:

DB_READ_COMMITTED

Эта транзакция будет иметь изоляцию степени 2.

Уровень обеспечивает стабильность курсора, но не повторяющиеся чтения.

Элементы данных, которые были ранее прочитаны этой транзакцией, могут быть удалены или изменены другими транзакциями до завершения этой транзакции.

DB_READ_UNCOMMITTED

Эта транзакция будет иметь изоляцию степени 1.

Операции чтения, выполняемые транзакцией, могут считывать измененные, но еще не зафиксированные данные. Будет игнорироваться, если при открытии данной базы данных не был указан флаг DB_READ_UNCOMMITTED.

DB_TXN_SNAPSHOT

Транзакция будет выполняться в режиме изоляции моментального снимка.

Для баз данных с установленным флагом DB_MULTIVERSION значения данных будут считываться такими, какими они были в начале транзакции, без установки блокировок чтения. Будет игнорироваться для операций с базами данных, в которых DB_MULTIVERSION не установлен. Изоляция моментального снимка не поддерживается при репликации.

Если транзакция моментального снимка попытается обновить данные, которые были изменены после того, как транзакция моментального снимка их прочитала, из операций обновления будет возвращена ошибка DB_LOCK_DEADLOCK.

DB_TXN_SYNC

Синхронно очищать журнал, когда эта транзакция фиксируется или готовится.

Это означает, что транзакция будет демонстрировать все свойства ACID (атомарность, согласованность, изоляция и долговечность).

Если с помощью метода DB_ENV->set_flags() не был указан флаг DB_TXN_NOSYNC, это поведение для окружений Berkeley DB является поведением по умолчанию.

Любое значение, указанное для этого метода, переопределяет эту настройку.

DB_TXN_NOSYNC

Не очищать журнал синхронно, когда эта транзакция фиксируется или готовится.

Это означает, что транзакция будет демонстрировать свойства ACI (атомарность, согласованность и изоляция), но не D (долговечность). То есть целостность базы данных будет сохранена, но возможно, что эта транзакция может быть отменена во время восстановления. Такое поведение может быть установлено для окружения Berkeley DB с помощью метода `DB_ENV->set_flags()`. Любое значение, указанное для данного метода, переопределяет эту настройку.

DB_TXN_NOWAIT

Если для любой операции Berkeley DB, выполняемой в контексте этой транзакции, недоступна блокировка, операция вернет DB_LOCK_DEADLOCK (или DB_LOCK_NOTGRANTED, если среда базы данных была настроена с использованием флага DB_TIME_NOTGRANTED).

Это поведение может быть установлено для окружения Berkeley DB использованием метода `DB_ENV->set_flags()`.

DB_TXN_WAIT

Если для любой операции Berkeley DB, выполняемой в контексте этой транзакции, блокировка недоступна, следует таки дождаться блокировки.

Если с помощью метода DB_ENV->set_flags() не был указан флаг DB_TXN_NOWAIT, это поведение для окружений Berkeley DB является поведением по умолчанию.

Любое значение, указанное для этого метода, переопределяет эту настройку.

DB_TXN_WRITE_NOSYNC

Когда транзакция будет зафиксирована, записать, но не делать синхронный сброс журнала. Это означает, что транзакция будет демонстрировать свойства ACI (атомарность, согласованность и изоляция), но не D (долговечность).

То есть целостность базы данных будет сохранена, но если система выйдет из строя, возможно, некоторое количество последних зафиксированных транзакций может быть отменено во время восстановления. Количество транзакций, находящихся под риском отмены, регулируется тем, как часто система сбрасывает грязные буферы на диск и как часто сбрасывается или проверяется журнал. Это поведение может быть установлено для среды Berkeley DB с помощью метода DB_ENV->set_flags().

DB_TXN_BULK

Включить оптимизацию транзакционной массовой вставки.

Когда этот флаг установлен, транзакция избегает регистрации содержимого вставок на новых выделенных страницах базы данных.

В транзакции, которая вставляет большое количество новых записей, при выборе этой опции может быть значительной экономия ввода-вывода.

Пользователи этой опции должны знать о нескольких проблемах. Когда действует оптимизация, выделения страниц, которые расширяют файл базы данных, регистрируются как обычно — это позволяет корректно прерывать транзакции как в режиме онлайн, так и во время восстановления.

Во время фиксации страницы базы данных сбрасываются на диск, что устраняет необходимость в накате транзакции во время обычного восстановления. Однако существуют и другие операции восстановления, которые зависят от наката, и необходимо соблюдать осторожность, когда с ними взаимодействуют транзакции DB_TXN_BULK.

В частности, DB_TXN_BULK несовместим с репликацией и просто игнорируется, если репликация включена. Кроме того, процедуры горячего резервного копирования должны следовать определенному протоколу, введенному в Berkeley DB 11gR2.5.1, который заключается в установке флага DB_HOTBACKUP_IN_PROGRESS в окружении перед началом копирования файлов.

Важно отметить, что инкрементные горячие резервные копии могут быть признаны недействительными с помощью оптимизации массовой вставки.

Оптимизация массовой вставки эффективна только для транзакций верхнего уровня. Флаг DB_TXN_BULK игнорируется, если parent не равен NULL.

Ошибки

`DB_ENV->txn_begin()` может завершиться ошибкой и вернуть одну из следующих ненулевых ошибок:

ENOMEM – достигнуто максимальное количество одновременных транзакций.

DB_TXN->commit() — завершить транзакцию

```
#include <db.h>

int DB_TXN->commit(DB_TXN    *tid,    // идентификатор транзакции
                   u_int32_t  flags); // флаги
```

В случае вложенных транзакций, если транзакция является родительской транзакцией, фиксация родительской транзакции приводит к фиксации всех неразрешенных дочерних транзакций родителя.

В случае вложенных транзакций, если транзакция является дочерней транзакцией, ее блокировки не снимаются, а приобретаются ее родителем.

Хотя фиксация дочерней транзакции будет успешной, фактическое разрешение дочерней транзакции откладывается до тех пор, не будет зафиксирована или отменена пока родительская транзакция, то есть, если ее родительская транзакция зафиксирована, она будет зафиксирована, а если ее родительская транзакция прервется, будет прервана и эта.

Все курсоры, открытые в рамках транзакции, должны быть закрыты до фиксации транзакции. Если они не закрыты, они будут закрыты этой функцией.

Когда операция закрытия курсора завершается неудачей, метод возвращает ненулевое значение ошибки для первого случая такой ошибки, закрывает остальные курсоры, а затем прерывает транзакцию.

После вызова DB_TXN->commit(), независимо от его возврата, дескриптор DB_TXN более не будет доступен.

Если DB_TXN->commit() обнаруживает ошибку, транзакция и все дочерние транзакции транзакции прерываются.

DB_TXN->abort() — аварийное завершение транзакции

```
#include <db.h>

int DB_TXN->abort(DB_TXN *tid);
```

Журнал воспроизводится в обратном порядке, и все необходимые операции отмены выполняются с помощью функции `tx_recover()`, указанной в DB_ENV->set_app_dispatch().

Перед возвратом DB_TXN->abort() будут сняты все блокировки, удерживаемые транзакцией.

В случае вложенных транзакций прерывание родительской транзакции приводит к прерыванию всех дочерних (неразрешенных или нет) родительской транзакции.

Все курсоры, открытые в транзакции, должны быть закрыты до того, как транзакция будет прервана. Если они не закрыты, они будут закрыты этой функцией.

Если операция закрытия курсора завершается неудачно, закрываются остальные курсоры, а окружение базы данных устанавливается в состояние паники.

После вызова DB_TXN->abort(), независимо от его возврата, к дескриптору DB_TXN нельзя снова получить доступ.

Метод DB_TXN->abort() возвращает ненулевое значение ошибки при неудаче и 0 при успехе.

Следующий пример открывает среду с поддержкой транзакций и базу данных, получает дескриптор транзакции, а затем выполняет операцию записи под ее защитой.

В случае любого сбоя в операции записи транзакция прерывается, и база данных остается в состоянии, как будто изначально не было предпринято никаких операций.

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

int main(void) {

    int       ret;
    int       ret_c;
    u_int32_t db_flags;
    u_int32_t env_flags;

    DB       *dbp   = NULL;
    DB_ENV   *envp  = NULL;
    DB_TXN   *txn   = NULL;
    DBT       key;
    DBT       data;

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

```
    // 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 |    // Создаем окружение, если не существует
                DB_INIT_TXN |  // инициализация транзакций
                DB_INIT_LOCK | // инициализация блокировок
                DB_INIT_LOG |  // инициализация журналирования
                DB_INIT_MPOOL; // инициализация кеш-памяти
    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 БД
    ret = db_create(&dbp, envp, 0);
    if (ret != 0) {
        envp->err(envp, ret, "Database creation failed");
        goto err;
    }
```

```
    db_flags = DB_CREATE |
               DB_AUTO_COMMIT;
    //
    // Открываем базу данных. При этом используем автоматическую фиксацию
    // при открытии, чтобы база данных могла поддерживать транзакции.
    //
    ret = dbp->open(dbp,       // указатель на БД
                    NULL,      // указатель на TXN
                    file_name, // имя файла
                    NULL,      // логическое имя БД
                    DB_BTREE,  // тип БД (btree)
                    db_flags,  // флаги открытия
                    0);        // режим создания по умолчанию
    if (ret != 0) {
        envp->err(envp, ret, "Database '%s' open failed",
        file_name);
        goto err;
    }

    // Готовим структуры DBT
    memset(&key, 0, sizeof(DBT));
    memset(&data, 0, sizeof(DBT));
    key.data  = &keystr;
    key.size  = strlen(keystr) + 1;
    data.data = &datastr;
    data.size = strlen(datastr) + 1;
```

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

// Выполним запись в базу данных.
// Если это не удалось, прерываем транзакцию.
ret = dbp->put(dbp, txn, &key, &data, 0);
if (ret != 0) {
    envp->err(envp, ret, "Database put failed.");
    txn->abort(txn);
    goto err;
}

// Фиксируем транзакцию.
ret = txn->commit(txn, 0);
if (ret != 0) {
    envp->err(envp, ret, "Transaction commit failed.");
    goto err;
}
```

```
err:
    // Закрываем БД
    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);
}
```

Фиксация транзакции

Чтобы полностью понять, что происходит при фиксации транзакции, сначала следует немного разобраться в том, что DB делает с подсистемой ведения журнала.

Ведение журнала приводит к тому, что все операции записи в базу данных идентифицируются в журналах, и по умолчанию эти журналы поддерживаются файлами на диске.

Эти журналы используются для восстановления баз данных в случае сбоя системы или приложения, поэтому, выполняя ведение журнала, DB обеспечивает целостность данных.

Более того, DB выполняет опережающую запись в журнал.

Это означает, что информация записывается в журналы до того, как будет изменена фактическая база данных. Это означает, что вся активность записи, выполненная под защитой транзакции, отмечается в журнале до того, как транзакция будет зафиксирована.

Однако следует иметь в виду, что база данных хранит журналы в памяти.

Если журналы сохраняются на диске, информация журнала в конечном итоге будет записана в файлы, но пока транзакция продолжается, данные журнала могут храниться только в памяти.

При фиксации транзакции, происходит следующее:

- Запись о фиксации записывается в журнал. Это значит, что изменения, внесенные транзакцией, с этого момента являются постоянными. По умолчанию эта запись выполняется синхронно на диск, поэтому запись о фиксации поступает в файлы журнала до того, как будут выполнены какие-либо другие действия.

- Любая информация из журнала, хранящаяся в памяти, (по умолчанию) синхронно записывается на диск. Это требование может быть смягчено в зависимости от типа выполняемой фиксации. Если журналы ведутся полностью в памяти, этот шаг не будет выполнен.

Систему журналирования для использования в памяти можно настроить.

- Все блокировки, удерживаемые транзакцией, снимаются.

Это означает, что операции чтения, выполняемые другими транзакциями или потоками управления, теперь могут видеть изменения, не используя незафиксированные чтения.

Чтобы зафиксировать транзакцию следует вызывать DB_TXN->commit().

ВАЖНО!!! — Фиксация транзакции не обязательно приводит к тому, что данные, измененные в кэше памяти, будут записаны в файлы, поддерживающие базы данных на диске.

«Загрязненные» страницы базы данных записываются по ряду причин, но транзакционная фиксация не является одной из них.

Ниже перечислено то, что может привести к записи загрязненной страницы базы данных в файл поддерживающей базы данных:

Контрольные точки

Контрольные точки приводят к тому, что все грязные страницы, которые в данный момент существуют в кэше, записываются на диск, после чего запись контрольной точки записывается в журналы.

Контрольные точки можно запускать явно.

Кэш заполнен

Если кэш в памяти заполнен, то чтобы освободить место для других страниц, которые необходимо использовать приложению, грязные страницы могут быть записаны на диск.

ВАЖНО!!! — Если грязные страницы записаны в файлы базы данных, то любые записи журнала, описывающие, как эти страницы были загрязнены, записываются на диск до того, как будут записаны страницы базы данных.

ВАЖНО!!! — Поскольку фиксация транзакции привела к принудительному сохранению на диске изменений базы данных, записанных в журналах, изменения являются «постоянными» по умолчанию, то есть их можно восстановить в случае сбоя приложения или системы. Однако время восстановления ограничено тем, сколько данных было изменено с момента последней контрольной точки, поэтому для приложений, которые выполняют много записей, вам может потребоваться инициировать контрольные точки с некоторой частотой.

После фиксации транзакции дескриптор транзакции, который использовался для транзакции, больше недействителен.

Чтобы выполнять действия с базой данных под управлением новой транзакции, необходимо получить новый дескриптор транзакции.

Неустойчивые транзакции[5]

По умолчанию фиксации транзакций являются устойчивыми, поскольку они вызывают синхронную запись изменений, выполненных в рамках транзакции, в файлы журналов на диске. Однако можно использовать и неустойчивые транзакции.

Неустойчивые транзакции могут понадобиться из соображений производительности.

Например, можно использовать транзакции просто для гарантии изоляции. В этом случае может не понадобиться гарантия устойчивости, и поэтому можно предотвратить дисковый ввод-вывод, который обычно сопровождает фиксацию транзакции.

Есть несколько способов снять гарантию долговечности для транзакций:

1) Можно указать DB_TXN_NOSYNC с помощью метода `DB_ENV->set_flags()`.

Это заставит DB записывать какие-либо данные журнала на диск при фиксации транзакции не синхронно. То есть изменения полностью сохраняются в кэше в памяти, а информация журнала не помещается принудительно в файловую систему для долгосрочного хранения. Тем не менее, данные журнала в конечном итоге попадут в файловую систему (при условии отсутствия сбоев приложения или ОС) как часть управления DB своими буферами журнала и/или кэшем.

Эта форма фиксации обеспечивает слабую гарантию долговечности, поскольку из-за сбоя приложения или ОС может произойти потеря данных.

2) Можно указать DB_TXN_WRITE_NOSYNC с помощью метода DB_ENV->set_flags().

Это приводит к синхронной записи данных журнала в буферы файловой системы ОС при фиксации транзакции. В конечном итоге данные будут записаны на диск, но это происходит, когда операционная система решает запланировать действие – фиксация транзакции может успешно завершиться до того, как ОС выполнит этот дисковый ввод-вывод.

---

5) Non-Durable Transactions

Данная форма фиксации защищает от сбоев приложения, но не от сбоев ОС.

Метод предлагает меньше возможностей для потери данных, чем DB_TXN_NOSYNC.

Данное поведение указывается на основе дескриптора окружения. Чтобы приложение демонстрировало согласованное поведение, необходимо указать этот флаг для всех дескрипторов окружения, используемых в приложении.

3) Поддерживать журналы полностью в памяти. В этом случае журналы никогда не записываются на диск. В результате теряются все гарантии долговечности.

Прерывание транзакции

Когда транзакция прерывается, все изменения базы данных, выполненные под ее защитой, отменяются, все блокировки, которые в данный момент удерживаются транзакцией, снимаются.

В этом случае данные просто остаются в том состоянии, в котором они были до того, как транзакция начала выполнять изменения данных.

После прерывания транзакции дескриптор транзакции, который использовался для транзакции, больше недействителен.

Для выполнения действий с базой данных под управлением новой транзакции необходимо получить новый дескриптор транзакции.

Чтобы прервать транзакцию, следует вызвать DB_TXN->abort().

Автоматическая фиксация[6]

Хотя транзакции часто используются для обеспечения атомарности нескольких операций с базой данных, иногда необходимо выполнить всего лишь одну операцию с базой данных под управлением транзакции. Вместо того, чтобы получать транзакцию, выполнять одну операцию записи, а затем либо фиксировать транзакцию, либо отменять, можно автоматически сгруппировать эту последовательность событий с помощью **auto commit**.

Чтобы использовать auto commit:

1) Следует открыть окружение и базы данных таким образом, чтобы они поддерживали транзакции.

Автоматическая фиксация может быть открыта для окружения или базы данных. Чтобы использовать автоматическую фиксацию для среды или базы данных, следует указать DB_AUTO_COMMIT в методе DB_ENV->set_flags() или DB->open().

Если автоматическая фиксация указывается при открытии среды, указывать автоматическую фиксацию для базы данных нет необходимости.

2) Предоставлять дескриптор транзакции для метода, который выполняет операцию записи в базу данных, не нужно.

ВАЖНО!!! — автоматическая фиксация недоступна для курсоров.

Если необходимо, чтобы операции курсора были транзакционно защищены, курсоры всегда должны открываться с помощью транзакции.

Примечание

Никогда не следует иметь в потоке более одной активной транзакции одновременно. Это особенно проблематично, если смешивается явная транзакция с автофиксаций.

---

6) Auto Commit

В следующем примере используется автоматическая фиксация для выполнения операции записи в базу данных:

```
#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;
    DBT key, data;

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

    // 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);
    }
```

```
    // Собираем флаги
    env_flags = DB_CREATE |
                DB_INIT_TXN |  // Инициализация транзакций
                DB_INIT_LOCK | // Инициализация блокировок
                DB_INIT_LOG |  // Инициализация журналирования
                DB_INIT_MPOOL; // Инициализация кеш-памяти
    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 БД
    ret = db_create(&dbp, envp, 0);
    if (ret != 0) {
        envp->err(envp, ret, "Database creation failed");
        goto err;
    }
    db_flags = DB_CREATE | DB_AUTO_COMMIT;
    // Открываем БД и используем автофиксацию
    ret = dbp->open(dbp,       // указатель на БД
                    NULL,      // указатель на TXN
                    file_name, // имя файла
                    NULL,      // логическое имя БД
                    DB_BTREE,  // тип БД(btree)
                    db_flags,  // флаги открытия
                    0);        // режим файла по умолчанию
```

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

    // Готовим структуры DBT
    memset(&key, 0, sizeof(DBT));
    memset(&data, 0, sizeof(DBT));
    key.data  = &keystr;
    key.size  = strlen(keystr) + 1;
    data.data = &datastr;
    data.size = strlen(datastr) + 1;

    // Выполняем запись в БД.
    // Дескриптор TXN не предоставляется, поскольку автофиксация.
    ret = dbp->put(dbp, NULL, &key, &data, 0);
    if (ret != 0) {
        envp->err(envp, ret, "Database put failed.");
        goto err;
    }

err:
    // Закрываем БД
    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);
}
```

Вложенные транзакции[7]

Вложенная транзакция используется для предоставления транзакционной гарантии для подмножества операций, выполняемых в рамках более крупной транзакции.

Это позволяет фиксировать и отменять подмножество операций независимо от более крупной транзакции.

Правила использования вложенной транзакции следующие:

1) Пока вложенная (дочерняя) транзакция активна, родительская транзакция не может выполнять никаких операций, кроме фиксации или отмены или создания дополнительных дочерних транзакций;

2) Фиксация вложенной транзакции не влияет на состояние родительской транзакции. Родительская транзакция по-прежнему остается не зафиксированной. Однако родительская транзакция теперь может видеть любые изменения, внесенные дочерней транзакцией. Эти изменения, конечно, по-прежнему скрыты от всех других транзакций, пока родительская транзакция также не зафиксируется.

3) Прерывание вложенной транзакции не влияет на состояние родительской транзакции. Единственным результатом прерывания является то, что ни родительская, ни любые другие транзакции не увидят никаких изменений базы данных, выполненных под защитой вложенной транзакции.

4) Если родительская транзакция фиксируется или прерывается, пока у нее есть активные дочерние транзакции, дочерние транзакции разрешаются так же, как и родительская.

То есть, если родительская транзакция прерывается, то дочерние транзакции также прерываются. Если родительская транзакция фиксируется, то любые изменения, выполненные дочерними транзакциями, также фиксируются.

---

7) Nested Transactions

5) Когда вложенная транзакция фиксируется, удерживаемые ей блокировки не снимаются. Вместо этого они теперь удерживаются родительской транзакцией до тех пор, пока эта родительская транзакция не зафиксируется.

6) Любые изменения базы данных, выполненные вложенной транзакцией, не видны за пределами более крупной охватывающей транзакции до тех пор, пока эта родительская транзакция не будет зафиксирована.

7) Глубина вложенности, которая может быть достигнута с помощью вложенных транзакций, ограничена только памятью.

Чтобы создать вложенную транзакцию, следует просто передайть дескриптор родительской транзакции при создании дескриптора вложенной транзакции.

Например:

```
DB_TXN *parent_txn, *child_txn;
// родительская транзакция
ret = envp->txn_begin(envp,
                      NULL,
                      &parent_txn,
                      0);
// дочерняя транзакция
ret = envp->txn_begin(envp,
                      parent_txn,
                      &child_txn,
                      0);
```

Транзакционные курсоры

Можно защитить транзакцию для операций с курсором, указав дескриптор транзакции во время создания курсора.

Кроме этого момента, дескриптор транзакции непосредственно методам курсора никогда не предоставляется.

ВАЖНО!!! — если курсор защищается транзакцией, необходимо убедиться, что курсор закрыт прежде, чем будет выполняться фиксация, либо транзакция будет отменяться.

Например:

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

int main(void) {

    DBT key, data;
    DBC *cursorp;

    DB_TXN *txn = NULL;

    int ret, c_ret;
    char *replacementString = "new string";
    ...
    // создание дескрипторов окружения и БД опущено
    ...
```

```
// получаем дескриптор транзакции
ret = envp->txn_begin(envp, NULL, &txn, 0);
if (ret != 0) {
    envp->err(envp, ret, "Transaction begin failed.");
    goto err;
}
// получим курсор в рамках транзакции
ret = dbp->cursor(dbp, txn, &cursorp, 0);
if (ret != 0) {
    envp->err(envp, ret, "Cursor open failed.");
    txn->abort(txn);
    goto err;
}
// используем курсор. Отметим, что txn методам курсора не предоставляем
// готовим структуры DBT
memset(&key,  0, sizeof(DBT));
memset(&data, 0, sizeof(DBT));
while (cursor->get(&key, &data, DB_NEXT) == 0) {
    data->data = (void *)replacementString;
    data->size = (strlen(replacementString) + 1) * sizeof(char);
    c_ret = cursor->put(cursor, &key, &data, DB_CURRENT);
    if (c_ret != 0) {
        // отменяем транзакцию и goto error
        envp->err(envp, ret, "Cursor put failed.");
        cursorp->close(cursorp);
        cursorp = NULL;
        txn->abort(txn);
```

```
            goto err;
        }
    }
    // фиксируем транзакцию.
    ret = cursorp->close(cursorp);
    if (ret != 0) {
        envp->err(envp, ret, "Cursor close failed.");
        txn->abort(txn);
        goto err;
    }
    ret = txn->commit(txn, 0);
    if (ret != 0) {
        envp->err(envp, ret, "Transaction commit failed.");
        goto err;
    }
err:
    // закрываем курсор
    // закрываем БД
    // закрываем окружение
    ...
    if (c_ret != 0) {
        ret = c_ret;
    }
    return (ret == 0 ? EXIT_SUCCESS : EXIT_FAILURE);
}
```