Базы данных

Лекция 10 – Berkeley DB. Управление файлами БД

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

2024.12.09

Оглавление

Управление файлами базы данных

BDB может хранить на диске несколько типов файлов:

1) файлы данных, которые содержат фактические данные в базе данных;

2) файлы журналов, которые содержат информацию, необходимую для восстановления базы данных в случае сбоя системы или приложения.

3) файлы области[1], которые содержат информацию, необходимую для общей работы приложения.

4) временные файлы, которые создаются только при определенных особых обстоятельствах. Эти файлы никогда не нужно ни резервировать ни управлять каким либо иным образом.

Из вышеупомянутых требуют управления пользователем только файлы данных и файлы журналов, обеспечивая их резервное копирование.

Также следует обращать внимание на объем дискового пространства, которое занимают файлы журналов, и периодически удалять ненужные файлы.

Также можно по желанию настроить подсистему журналирования, чтобы она наилучшим образом соответствовала потребностям и требованиям приложения.

---

1) region files

Контрольные точки (Checkpoints)

Когда изменяются базы данных (фиксируется транзакция), изменения записываются в журналы БД, но они не обязательно отражаются в реальных файлах базы данных на диске.

Это означает, что с течением времени в файлах журналов содержится все больше данных, которые пока еще не попали в файлы с данными. В результате приходится хранить больше файлов журналов, чем может быть необходимо.

Кроме того, любое восстановление из файлов журналов будет занимать все больше времени, поскольку в файлах журналов содержится больше данных, которые должны быть восстановлены и попасть в файлы данных.

Можно уменьшить эти проблемы, периодически запуская «контрольные точки» для окружения.

Контрольная точка:

1) Очищает грязные страницы из кэша в памяти. Это означает, что изменения данных, обнаруженные в кэше в памяти, записываются в файлы базы данных на диске.

Контрольная точка также приводит к тому, что данные, загрязненные незафиксированной транзакцией, также записываются в файлы базы данных на диск. В этом последнем случае для удаления любых таких изменений, которые впоследствии были отменены приложением с помощью прерывания транзакции, используется обычное восстановление DB.

(Процедуры восстановления, стр. 15).

2) Выполняет запись контрольной точки.

3) Очищает журнал. Это приводит к записи всех данных в журнале, которые еще не были записаны на диск.

4) Записывает список открытых баз данных.

Есть несколько способов запустить контрольную точку. Один из способов — использовать утилиту командной строки **db_checkpoint(1)**.

Однако следует отметить, что эту утилиту командной строки нельзя использовать, если окружение было открыто с помощью **DB_PRIVATE**.

Также можно запустить поток, который периодически проверяет окружение, вызывая метод **DB_ENV->txn_checkpoint()**.

При необходимости можно предотвратить выполнение контрольной точки, если с момента последней контрольной точки не было записано больше указанного количества данных журнала.

Также можно предотвратить выполнение контрольной точки, если с момента последней контрольной точки не прошло больше указанного количества времени.

Эти условия особенно интересны, если есть несколько потоков или процессов, выполняющих запуск контрольных точек.

Запуск контрольных точек может быть довольно затратным. БД должна сбрасывать каждую грязную страницу в файлы резервной базы данных. С другой стороны, если контрольные точки не запускаются достаточно часто, время восстановления может быть неоправданно долгим, и может использовать больше дискового пространства, чем действительно нужно.

Кроме того, пока не будет запущена контрольная точка, невозможно удалить файлы журналов. Поэтому решение о том, как часто запускать контрольную точку, является одним из наиболее распространенных действий по настройке приложений БД.

Пример запуска контрольной точку из отдельного потока управления:

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

void* checkpoint_thread(void *); // поток с запуском контрольной точки

int main(void) {

    int         ret;
    u_int32_t   env_flags;
    DB_ENV     *envp = NULL;
    const char *db_home_dir = "/tmp/myEnvironment";
    pthread_t ptid;

    // Создаем окружение
    ret = db_env_create(&envp, 0);
    if (ret != 0) {
        fprintf(stderr,
                "Ошибка создания дескриптора окружения: %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

    // Открываем окружение
    ret = envp->open(envp, db_home_dir, env_flags, 0);
    if (ret != 0) {
        fprintf(stderr, "Ошибка открытия окружения: %s\n",
                db_strerror(ret));
        goto err;
    }
    // Старт потока с контрольной точкой
    ret = pthread_create(&ptid, NULL, checkpoint_thread, (void *)envp);
    if (ret != 0) {
        fprintf(stderr,
                "txnapp: failed spawning checkpoint thread: %s\n",
                db_strerror(ret));
        goto err;
    }
    // All other threads and application shutdown code omitted for brevity.
err:
    ...
}
```

```
void* checkpoint_thread(void *arg) {

    DB_ENV *dbenv;
    int ret;
    dbenv = arg;
    // Checkpoint once a minute.
    for (;; sleep(60)) {
        if ((ret = dbenv->txn_checkpoint(dbenv, 0, 0, 0)) != 0) {
            dbenv->err(dbenv, ret, "checkpoint thread");
            exit(1);
        }
    }
    /* NOTREACHED */
}
```

Процедуры резервного копирования

Важной частью транзакционных гарантий является долговечность (Durability).

Это означает, что после успешного завершения транзакции приложение всегда будет видеть результаты этой транзакции.

Конечно, ни один программный алгоритм не может гарантировать долговечность в условиях физической потери данных. Жесткие диски могут выйти из строя, и если данные не скопированы в места, отличные от основных дисковых накопителей, то при выходе этих накопителей из строя данные определенно будут потеряны.

Поэтому, чтобы действительно получить гарантию долговечности, необходимо убедиться, что все данные, хранящиеся на диске, резервируются во вторичном или альтернативном хранилище, например, на вторичных дисковых накопителях или автономных лентах.

Существует три различных типа резервного копирования, которые можно выполнять с базами данных и файлами журналов:

- резервное копирование в автономном режиме (Offline backups);
- горячее создание резервных копий (Hot backups);
- инкрементное создание резервных копий (Incremental backups).

1) Резервное копирование в автономном режиме (Offline backups)

Этот тип резервного копирования, пожалуй, самый простой в выполнении, поскольку он включает в себя простое копирование базы данных и файлов журнала в область автономного хранения. Он также дает снимок базы данных в фиксированный, известный момент времени. Однако во время выполнения записи в базу данных этот тип резервного копирования выполнить невозможно.

2) Горячее создание резервных копий (Hot backups)

Этот тип резервного копирования дает снимок базы данных. Поскольку приложение может писать в базу данных во время создания снимка, точное состояние базы данных для этого снимка неизвестно.

3) Инкрементное создание резервных копий (Incremental backups)

Этот тип резервного копирования обновляет ранее выполненную резервную копию.

Примечание

Резервные копии баз данных не зависят от порядка байтов. То есть резервная копия, сделанная на машине с обратным порядком байтов, может быть использована для восстановления базы данных, находящейся на машине с обратным порядком байтов.

После выполнения резервного копирования можно выполнить и катастрофическое восстановление, чтобы восстановить базы данных из резервной копии. (Катастрофическое восстановление, стр.19).

Обратите внимание, что также можно поддерживать горячее аварийное переключение. (Использование горячих аварийных переключений, стр.31).

Об утилитах копирования Unix

Если копируется файлы базы данных, они должны копироваться ***атомарно, кратно размеру страницы базы данных.***

Это значит, что чтения, выполняемые программой копирования, не должны чередоваться с записями других потоков управления, и программа копирования должна считывать базы данных кратно размеру страницы базовой базы данных.

Как правило, операционные системы уже гарантируют это, а системные утилиты обычно считывают фрагментами размером степени двойки, которые больше максимально возможного размера страницы базы данных Berkeley DB.

На некоторых платформах (в частности, в некоторых выпусках Solaris) утилита копирования (cp) была реализована с использованием системного вызова mmap(), а не системного вызова read(). Поскольку mmap() не давал той же гарантии атомарности чтения, что и read(), утилита cp могла создавать поврежденные копии баз данных.

Кроме того, на некоторых платформах есть реализации утилиты tar, которая по умолчанию выполняет чтение блоков по 10 КБ. Даже если указан размер выходного блока, утилита все равно не будет читать базу данных блоками, кратными размеру блока БД. Соответсвенно, результатом может стать поврежденная резервная копия.

Чтобы избежать возможных проблемы, следует использовать утилиту **dd(1)** вместо **cp(1)** или **tar(1)**. При использовании **dd(1)** следует обязательно указать размер блока, равный или даже кратный размеру страницы БД. При использовании системных утилит для копирования файлов базы данных, имеет смысл использовать утилиту трассировки системных вызовов (ktrace или truss), чтобы убедиться, что не используется размер ввода-вывода, меньший размера страницы БД. Также можно использовать эти утилиты, чтобы убедиться, что системная утилита не использует системный вызов, отличный от `read( )`.

Резервные копии в автономном режиме (Offline Backups)

Чтобы создать резервную копию в автономном режиме следует:

1) зафиксировать или отменить все текущие транзакции.

2) приостановить все записи в базу данных.

3) принудительно создать контрольную точку. (Контрольные точки (Checkpoints)).

4) скопировать все файлы базы данных в место хранения резервной копии.

Можно скопировать все файлы базы данных или определить, какие файлы базы данных были записаны в течение жизненного цикла текущих журналов.

Для этого используется метод **DB_ENV->log_archive()** с опцией **DB_ARCH_DATA**, либо команда **db_archive** с опцией **-s**.

Резервное копирование только что измененных баз данных работает только в том случае, если есть все файлы журналов. Если файлы журналов были удалены по какой-либо причине, то использование **log_archive()** может привести к невосстановимой резервной копии.

5) Скопировать ***последний*** файл журнала в резервную копию.

Файлы журнала имеют имена log.xxxxxxxxxx, где xxxxxxxxxx — это порядковый номер.

Последний файл журнала — это файл с самым большим номером.

Горячее резервное копирование (Hot Backup)

Чтобы реализовать горячее резервное копирование, не нужно останавливать операции в базе данных. Транзакции могут быть текущими, и во время резервного копирования можно записывать данные в базу. Однако это означает, что никто не знает точно, в каком состоянии находится база данных во время резервного копирования.

Можно использовать утилиту командной строки **db_hotbackup** для создания горячего резервного копирования. Эта программа опционально запускает контрольную точку, а затем копирует все необходимые файлы в целевой каталог.

Также можно создать свой собственный объект горячего резервного копирования с помощью метода **DB_ENV->backup()**.

Кроме того, можно вручную создать горячее резервное копирование:

1. Следует установить флаг **DB_HOTBACKUP_IN_PROGRESS** в окружении, используе **DB_ENV->set_flags()**.

2. Следует скопировать все файлы базы данных в резервную копию.

Можно скопировать все файлы базы данных или определить, какие файлы базы данных были записаны в течение жизненного цикла текущих журналов. Для этого следует использовать либо **DB_ENV->log_archive()** с опцией **DB_ARCH_DATA**, либо команду **db_archive(1)** с опцией **-s**.

3. Следует скопировать все журналы в резервное хранилище.

4.Следует сбросить флаг **DB_HOTBACKUP_IN_PROGRESS**.

Важно скопировать файлы базы данных, и только потом затем журналы.

Таким образом, можно завершить или откатить любые операции с базой данных, которые были выполнены лишь частично при копировании баз данных.

Инкрементальное резервное копирование (Incremental Backups)

После того, как создана полная резервная копия (то есть обычная или горячая), можно создавать инкрементальные резервные копии. Для этого просто следует скопировать все текущие файлы журналов в место хранения резервной копии.

Инкрементальные резервные копии не требуют запуска контрольной точки или прекращения операций записи в базу данных.

При работе с инкрементальными резервными копиями следует помнить, что чем больше файлов журналов содержится в резервной копии, тем больше времени займет восстановление.

Следует запускать полное резервное копирование с некоторым интервалом, а затем выполнять инкрементальное резервное копирование с более коротким интервалом.

Частота, с которой нужно запускать полное резервное копирование, определяется скоростью изменения баз данных и чувствительностью приложения к длительному восстановлению (если оно требуется).

Также можно сократить время восстановления, запуская восстановление из резервной копии по мере создания каждой инкрементной резервной копии. Запуск восстановления по мере выполнения означает, что у DB будет меньше работы, если когда-либо понадобится восстановить окружение из резервной копии.

Процедуры восстановления (Recovery Procedures)

БД поддерживает два типа восстановления:

1) Обычное восстановление, которое запускается, когда окружение открывается при запуске приложения, проверяет только те записи журнала, которые необходимы для приведения баз данных в согласованное состояние с момента ***последней контрольной точки***.

Обычное восстановление начинается с журналов, которые использовались транзакциями, активными во время последней контрольной точки, и проверяются все журналы с этого момента до текущих журналов.

2) Катастрофическое восстановление выполняется так же, как и обычное восстановление, за исключением того, что оно проверяет все доступные файлы журналов. Катастрофическое восстановление используется для восстановления баз данных из ***ранее созданной резервной копии***.

Обычное восстановление следует запускать всякий раз, когда запускается приложение.

Катастрофическое восстановление запускается всякий раз, когда потеряны или повреждены файлы базы данных и необходимо восстановить их из резервной копии.

Нормальное восстановление (Normal Recovery)

Нормальное восстановление проверяет содержимое файлов журнала окружения и использует эту информацию, чтобы убедиться, что файлы базы данных согласованы с информацией, содержащейся в файлах журнала.

Нормальное восстановление также воссоздает файлы из области окружения.

Оно также очищает любые неразблокированные блокировки, которые приложение могло удерживать во время некорректного завершения работы приложения.

Нормальное восстановление запускается только для тех файлов журнала, которые были созданы с момента *последней контрольной точки*. По этой причине время восстановления зависит от того, сколько данных было записано с момента последней контрольной точки, и, следовательно, от того, сколько информации в файле журнала нужно изучить.

Если контрольные точки запускаются нечасто, то нормальное восстановление может занять относительно много времени.

Примечание

Следует запускать нормальное восстановление каждый раз, когда выполняется запуск приложения.

Чтобы запустить нормальное восстановление, следует:

- убедиться, что все дескрипторы окружения закрыты.
- нормальное восстановление должно быть однопоточным.
- указать флаг DB_RECOVER при открытии окружения.

Также можно запустить восстановление, приостановив или завершив работу приложения и используя утилиту командной строки **db_recover(1)**.

Пример

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

int main(void) {

    int ret;
    u_int32_t env_flags;
    DB_ENV *envp = NULL;
    const char *db_home_dir = "/tmp/myEnvironment";
    /* Open the environment */
    ret = db_env_create(&envp, 0);
    if (ret != 0) {
        fprintf(stderr, "Ошибка создания дескриптора окружения: %s\n",
                db_strerror(ret));
        return (EXIT_FAILURE);
    }
    /* Open the environment, specifying that recovery is to be run. */
    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
        DB_RECOVER;     // Run normal recovery
```

```
    /* Open the environment. */
    ret = envp->open(envp, db_home_dir, env_flags, 0);
    if (ret != 0) {
        fprintf(stderr, "Ошибка открытия окружения: %s\n",
                db_strerror(ret));
        goto err;
    }
// Все остальные операции отсюда идентичны.
// Следует обратить внимание, однако, что не создано никаких других потоков
// управления до завершения восстановления.
```

Катастрофическое восстановление (Catastrophic Recovery)

Катастрофическое восстановление используется, когда восстанавливаются базы данных из ранее созданной резервной копии.

Для восстановления баз данных из предыдущей резервной копии следует скопировать резервную копию ***в новый каталог окружения***, а затем запустить катастрофическое восстановление. Невыполнение этого требования может привести к тому, что внутренние структуры базы данных не будут синхронизированы с файлами журналов.

Катастрофическое восстановление должно быть запущено в однопоточном режиме.

Чтобы запустить катастрофическое восстановление, следует:

- завершить все операции с базой данных.
- восстановить резервную копию в пустой каталог.
- указать флаг **DB_RECOVER_FATAL** при открытии окружения. Эта открытое окружение должно быть однопоточным.

Также можно запустить восстановление, приостановив или завершив работу приложения и использовать утилиту командной строки **db_recover** с опцией **-c**.

Катастрофическое восстановление проверяет каждый доступный файл журнала, а не только те файлы журнала, которые были созданы с момента последней контрольной точки, как в случае обычного восстановления. По этой причине катастрофическое восстановление, скорее всего, займет больше времени, чем обычное восстановление.

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

int main(void) {

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

    // Open the environment
    ret = db_env_create(&envp, 0);
    if (ret != 0) {
        fprintf(stderr, "Ошибка создания дескриптора окружения: %s\n",
                db_strerror(ret));
        return (EXIT_FAILURE);
    }
    // Открываем окружение, указав, что необходимо запустить восстановление
    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
        DB_RECOVER_FATAL; // Run catastrophic recovery
```

```
    // Открываем окружение
    ret = envp->open(envp, db_home_dir, env_flags, 0);
    if (ret != 0) {
        fprintf(stderr, "Ошибка открытия окружения: %s\n",
                db_strerror(ret));
        goto err;
    }
// Все остальные операции отсюда обычные.
```

Проектирование приложения для восстановления

При создании приложения БД следует подумать о том, как будет запускаться восстановление.

Если это однопоточное приложение с одним процессом, восстановление запустить довольно просто, когда приложение впервые открывает свое окружение.

В этом случае нужно только решить, будет ли запускаться восстановление каждый раз, когда приложение стартует (рекомендуется) или только иногда, запускаемое параметром, контролируемым пользователем приложения.

Однако для многопоточных и многопроцессных приложений необходимо тщательно продумать код запуска приложения, чтобы запускать восстановление только тогда, когда это имеет смысл.

Восстановление для многопоточных приложений

Если приложение использует только один дескриптор окружения, то обработка восстановления для многопоточного приложения не сложнее, чем для однопоточного приложения. В этом случае просто открывается окружение в главном потоке приложения, а затем этот дескриптор передается каждому из потоков, которые будут выполнять операции с базой данных. (Пример транзакции, стр. Ошибка: источник перекрёстной ссылки не найден).

В качестве альтернативы можно сделать так, чтобы каждый рабочий поток открывал свой собственный дескриптор окружения. Однако в этом случае проектирование восстановления немного сложнее.

Обычно, когда поток, выполняющий операции с базой данных, дает сбой или зависает, часто лучше просто перезапустить приложение и запустить восстановление при запуске приложения в обычном режиме.

Однако не все приложения могут позволить себе перезапуск из-за неправильного поведения одного потока.

Если необходимо продолжить работу в условиях неправильного поведения потока, то как минимум, если поток, выполняющий операции с базой данных, выходит из строя или зависает, должно быть запущено восстановление.

Следует помнить, что восстановление очищает окружение от всех неосвобожденных блокировок, включая любые, которые могут быть принадлежать прерванному потоку.

Если эти блокировки не сбрасываются, другие потоки, выполняющие операции с базой данных, могут столкнуться с блокировками, полученными, но так и не сброшенными невыполненным потоком. В результате приложение зависнет на неопределенный срок.

Чтобы запустить восстановление в этих обстоятельствах, следует:

1. приостановить или завершить работу всех других потоков, выполняющих операции с базой данных.

2. отказаться от всех открытых дескрипторов окружения – попытка корректно закрыть эти дескрипторы может привести к проблемам; закрытие может завершиться неудачей, если окружение нуждается в восстановлении. По этой причине лучше и проще всего просто от дескриптора отказаться .

3. открыть новые дескрипторы окружения, запустив восстановление по мере их открытия. (Обычное восстановление, стр. 16).

4. перезапустить все потоки базы данных.

Традиционный способ справиться с этой активностью — создать поток-наблюдатель, который следит за тем, что с потоками все в порядке, и выполняет вышеуказанные действия, если это не так.

Однако в случае, когда каждый рабочий поток открывает и поддерживает свой собственный дескриптор окружения, восстановление усложняется по двум причинам:

1) Для некоторых приложений и рабочих нагрузок может быть целесообразно предоставить потокам базы данных возможность аккуратно завершить любые текущие транзакции.

Если это так, код должен быть способен сигнализировать каждому потоку остановить действия базы данных и закрыть его окружение – если же просто запустить восстановление в окружении, потоки базы данных обнаружат это и выйдут из строя в процессе выполнения своих операций с базой данных.

2) Код должен гарантировать, что только один поток запустит восстановление, прежде чем разрешить всем остальным потокам открыть соответствующие дескрипторы окружения.

Восстановление должно быть однопоточным, потому что когда в окружении запускается восстановление, окружение удаляется, а затем создается заново. Это приведет к тому, что все другие процессы и потоки, когда попытаются выполнить операции в недавно восстановленном окружении, «потерпят неудачу».

Если все потоки запустят восстановление при запуске, то, скорее всего, некоторые потоки потерпят неудачу, потому что используемое ими окружение было уже восстановлено. Это заставит поток повторно выполнить свой собственный путь восстановления. В лучшем случае это неэффективно, а в худшем — может привести к тому, что приложение перейдет в бесконечный цикл восстановления.

Восстановление в многопроцессных приложениях

Часто приложения БД используют несколько процессов для взаимодействия с базами данных. Например, может быть долго работающий процесс, такой как какой-то сервер, а затем ряд административных инструментов, которые вы используете для проверки и администрирования базовых баз данных. Или, в некоторых веб-архитектурах, различные службы запускаются как независимые процессы, которые управляются сервером.

В любом случае, восстановление для многопроцессной окружения осложнено по двум причинам:

1) В случае необходимости запуска восстановления бывает необходимо уведомить процессы, взаимодействующие с окружением, о том, что вот-вот произойдет восстановление, и дать им возможность корректно завершить работу. Стоит ли это делать, полностью зависит от характера приложения.

Некоторые долго работающие приложения с несколькими процессами, выполняющими серьезную работу, могут захотеть это сделать.

Другие приложения с процессами, выполняющими операции с базой данных, которые, будут повреждены из-за ошибок в других процессах, скорее всего это делать не будут, поскольку для этой группы шансы на корректное завершение работы могут быть низкими.

2. В отличие от сценариев с одним процессом, запуск восстановления для каждого процесса, взаимодействующего с базами данных, при его старте будет расточительным. Это отчасти потому, что восстановление занимает некоторое время, но в основном имеет смысл избежать ситуации, когда сервер будет повторно открывать все свои дескрипторы окружения только потому, что кто-то запустил утилиту администрирования базы данных из командной строки, которая всегда запускает восстановление.

DB предлагает два метода, с помощью которых можно управлять восстановлением для многопроцессных приложений DB. Каждый из них имеет свои сильные и слабые стороны.

Эффекты многопроцессного восстановления

Стоит отметить, что процессы восстановления могут привести к тому, что один процесс будет выполнять восстановление, в то время как другие процессы в данный момент активно выполняют операции с базой данных.

Когда это происходит, текущая операция базы данных аварийно завершается сбоем, указывая на состояние **DB_RUNRECOVERY**. Это означает, что приложение должно немедленно прекратить любые операции базы данных, которые оно может выполнять, отменить все открытые им дескрипторы окружения и получить и открыть новые дескрипторы.

Чистый эффект заключается в том, что любые записи, выполненные «неразрешенными» транзакциями, будут потеряны. Для постоянных приложений (например, серверов) предоставляемые ими сервисы также будут недоступны в течение времени, необходимого для завершения восстановления и для всех участвующих процессов, чтобы повторно открыть свои дескрипторы окружения.

Регистрация процесса (Process Registration)

Один из способов управления многопроцессным восстановлением — это «регистрация» каждым процессом своего окружения.

При этом процесс получает возможность видеть, используют ли окружение другие приложения, и если да, то были ли они ненормально завершены. Если обнаружено ненормальное завершение, процесс запускает восстановление; в противном случае не запускает.

Использование регистрации процесса также гарантирует, что восстановление будет сериализовано между приложениями. То есть, только один процесс за раз имеет возможность

запустить восстановление. Обычно это означает, что первый запущенный процесс запустит восстановление, а все остальные процессы молча не будут запускать восстановление, поскольку оно не нужно.

Чтобы приложение зарегистрировало свое окружение, следует при открытии окружения указать флаг **DB_REGISTER**. Также можно указать **DB_RECOVER**.

Указывать **DB_RECOVER_FATAL** при использовании флага **DB_REGISTER** является ошибкой.

Если во время открытия DB определяет, что необходимо запустить восстановление, она автоматически запустит правильный тип восстановления, если в открытом окружении будет указано нормальное восстановление.

Если окружение будет зарегистрировано, а нормальное восстановление не будет указано, то если процесс регистрации определит необходимость восстановления, оно запущено не будет. В этом случае открытие окружения просто завершается неудачей, возвращая **DB_RUNRECOVERY**.

Примечание

Если нормальное восстановление при открытии в первый раз зарегистрированного окружения в приложении указано не будет, то это приложение завершит открытие окружения сбоем, вернув DB_RUNRECOVERY. Это связано с тем, что первый процесс для регистрации должен создать внутренний файл регистрации, и при создании этого файла принудительно выполняется восстановление.

Чтобы избежать ненормального завершения открытого окружения, следует указать восстановление в открытом окружении хотя бы для первого процесса, запускаемого в приложении.

Кроме того, если при регистрации своей окружения указывается **DB_ENV_FAILCHK**, то при открытии окружения выполняется проверка на отказ. Если во время процесса проверки на отказ обнаруживается ненормальное завершение для любого из процессов, задействованных в приложении, DB снимает все блокировки чтения, удерживаемые мертвым процессом, и выполняет прерывания транзакций по мере необходимости. Это делается в попытке очистить окружение. В этой ситуации, если общая очистка окружения невозможна и если нормальное восстановление не указано при открытии окружения, то открытие будет прервано с возвращением **DB_RUNRECOVERY**.

Однако, если такая ситуация возникает и было указано нормальное восстановление, то чтобы вернуть окружение в работоспособное состояние, запускается соответствующий тип восстановления (нормальное или фатальное).

Существуют некоторые ограничения/требования в отношении координации различных процессов при восстановлении с помощью регистрации:

1) Может быть только один дескриптор окружения на одно окружение в одном процессе. В случае многопоточных процессов дескриптор окружения должен быть общим для потоков.

2) Все процессы, совместно использующие окружение, должны использовать регистрацию. Если регистрация не используется единообразно во всех участвующих процессах, то можно увидеть противоречивые результаты с точки зрения способности приложения распознавать необходимость запуска восстановления.

Проверка отказов (Failure Checking)

Для очень больших и надежных многопроцессных приложений наиболее распространенным способом обеспечения работы всех процессов в соответствии с их предназначением является использование сторожевого процесса. Чтобы помочь процессу-сторожу, DB предлагает механизм проверки отказов.

Когда поток управления выходит из строя с открытыми дескрипторами окружения, результатом является то, что ресурсы могут остаться заблокированными или поврежденными. Другие потоки управления в зависимости от шаблонов доступа к данным могут как быстро обнаружить, что ресурсы недоступны, так и не обнаружить их вообще.

Механизм проверки отказов БД позволяет сторожевому процессу обнаружить, является ли окружение непригодным для использования в результате сбоя потока управления.

Механизм проверки отказов должен вызываться периодически (например, раз в минуту) из процесса сторожевого таймера. Если окружение считается непригодным для использования, то процесс сторожевого таймера уведомляется о том, что следует запустить восстановление.

Затем сторожевой процесс должен фактически запустить восстановление. Также сторожевой процесс должен решить, что делать с текущими запущенными процессами перед запуском восстановления. Например, он может попытаться корректно завершить или уничтожить все соответствующие процессы перед запуском восстановления.

Проверка отказов не обязательно должна запускаться из отдельного процесса, хотя концептуально именно так предполагается использование механизма. Этот же механизм может использоваться в многопоточном приложении, которому требуется поток-сторожевой таймер.

Для использования проверки на отказы необходимо:

1. Предоставить обратный вызов **is_alive()** с помощью метода **DB_ENV->set_isalive()**. DB использует этот метод для определения того, активен ли указанный процесс и поток при выполнении проверки на сбой.

2. По возможности предоставить обратный вызов **thread_id()**, который уникально идентифицирует процесс и поток управления. Этот обратный вызов необходим только в том случае, если стандартные функции идентификации процесса и потока для платформы недостаточны для использования при проверке на сбой. Это требуется редко и обычно потому, что идентификаторы потока и/или процесса, используемые системой, не могут вписаться в беззнаковое целое число.

Этот обратный вызов предоставляется с помощью метода **DB_ENV->set_thread_id()**.

3. Периодически вызывать метод **DB_ENV->failchk()**. Можно делать это либо периодически (например, раз в минуту), либо всякий раз, когда поток управления завершает работу приложения.

Если этот метод определяет, что поток управления завершил работу, удерживая блокировки чтения, эти блокировки автоматически снимаются.

Если поток управления завершил работу с неразрешенной транзакцией, эта транзакция прерывается.

Если существуют какие-либо другие проблемы помимо этих, такие, что окружение необходимо восстановить, метод вернет **DB_RUNRECOVERY**.

Использование горячих аварийных переключений (Using Hot Failovers)

Для целей отказоустойчивости можно поддерживать резервную копию.

Горячие аварийные переключения отличаются от процедур резервного копирования и восстановления тем, что данные, используемые для традиционных резервных копий, обычно копируются в автономное хранилище. Время восстановления для традиционной резервной копии определяется:

1) Насколько быстро можно извлечь этот носитель. Обычно носитель для критических резервных копий перемещается в безопасное место в удаленном месте, поэтому этот шаг может занять относительно много времени.

2) Насколько быстро можно считать резервную копию с носителя на локальный диск. Если очень большие резервные копии или носитель очень медленный, это может быть длительным процессом.

3) Сколько времени потребуется, чтобы запустить катастрофическое восстановление с использованием только что восстановленной резервной копии. Этот процесс может быть длительным, поскольку в процессе восстановления необходимо проверить каждый файл журнала.

При использовании горячего аварийного переключения резервная копия сохраняется в месте, к которому можно достаточно быстро получить доступ. Обычно это второй локальный диск машины. В этой ситуации время восстановления очень быстрое, поскольку требуется только заново открыть окружение и базу данных.

Горячие переключения не защищают от действительно катастрофических бедствий типа пожара, поскольку резервная копия находится на локальной машине. Однако можно защититься от более обыденных проблем, типа поломки жесткого диска, сохраняя резервную копию на втором диске, который управляется альтернативным контроллером диска.

Для поддержания горячего переключения следует:

1. Скопировать все активные файлы базы данных в резервный каталог. Можно использовать утилиту командной строки **db_archive** с опцией **-s**, чтобы определить все активные файлы базы данных.

2. Определить все неактивные файлы журналов в рабочем окружении и переместить их в резервный каталог. Можно использовать команду **db_archive** без опций командной строки, чтобы получить список этих файлов журналов.

3. Определить активные файлы журналов в рабочем окружении и скопировать их в резервный каталог. Можно использовать команду **db_archive** с опцией **-l**, чтобы получить список этих файлов журналов.

4. Запустить катастрофическое восстановление для резервного каталога. Можно использовать команду **db_recover** с опцией **-c**, чтобы это сделать.

5. При желании скопировать резервную копию в архивное местоположение.

После выполнения этой процедуры можно поддерживать активную горячую резервную копию, повторяя шаги 2–5 так часто, как того требует приложение.

Примечание

Если выполняется шаг 1, за ним должны следовать шаги 2–5, чтобы обеспечить согласованность горячей резервной копии.

Примечание

Вместо использования предыдущей процедуры можно использовать утилиту командной строки **db_hotbackup**, чтобы сделать то же самое. Эта утилита (необязательно) запустит контрольную точку, а затем скопирует все необходимые файлы в целевой каталог.

Чтобы фактически выполнить отказоустойчивость, просто следует:

1. Завершить все процессы, которые работают в исходном окружении.

2. Если есть архивная копия резервного окружения, можно дополнительно попробовать скопировать оставшиеся файлы журналов из исходного окружения и запустить катастрофическое восстановление в этой резервном окружении. Делать это следует только в том случае, если есть архивная копия резервного окружения.

Этот шаг может позволить восстановить данные, созданные или измененные в исходном окружении, но которые не успели отразиться в окружении горячего резервного копирования.

3. Повторно открыть окружение и базы данных в обычном режиме, но использовать вместо рабочего окружения резервное.

Удаление файлов журнала (Removing Log Files)

По умолчанию DB файлы журнала не удаляет. По этой причине файлы журнала DB со временем разрастутся и займут неоправданно большой объем дискового пространства.

Чтобы защититься от этого, следует периодически предпринимать административные действия по удалению файлов журнала, которые больше не используются приложением.

Можно удалить файл журнала, если выполняются все следующие условия:

- файл журнала не участвует в активной транзакции.
- после создания файла журнала была выполнена контрольная точка.
- файл журнала не является единственным файлом журнала в окружении.
- файл журнала, который предполагается удалить, уже был включен в автономную или горячую резервную копию.

Несоблюдение этого последнего условия может привести к тому, что резервные копии станут непригодными для использования.

DB предоставляет несколько механизмов для удаления файлов журналов, которые соответствуют всем критериям, кроме последнего – DB не может узнать, какие файлы журналов уже включены в резервную копию.

Следующие механизмы упрощают удаление ненужных файлов журналов, но могут привести к тому, что резервная копия станет непригодной для использования, если файлы журналов предварительно не будут сохранены в архивном месте.

Все следующие механизмы автоматически удаляют ненужные файлы журналов:

1) Запуск утилиты командной строки **db_archive** с опцией **-d**.
2) Вызов из приложения **DB_ENV->log_archive()** с флагом **DB_ARCH_REMOVE**.
3) Вызов метода **DB_ENV->log_set_config()** с флагом **DB_LOG_AUTO_REMOVE**.

Флаг **DB_LOG_AUTO_REMOVE** можно установить в любой момент жизненного цикла приложения. Установка этого параметра влияет на все дескрипторы окружения, открытые для окружения, а не только на дескриптор, используемый при установке флага.

В отличие от других механизмов удаления журналов, метод 3) фактически приводит к постоянному удалению файлов журналов по мере того, как они становятся ненужными. Это очень удобно если необходимо использовать минимум дискового пространства.

Данный механизм оставит файлы журналов, необходимые для запуска нормального восстановления. Однако весьма вероятно, что этот механизм не позволит запустить катастрофическое восстановление.

НЕ следует использовать этот механизм, если необходимо иметь возможность выполнять катастрофическое восстановление или иметь возможность поддерживать горячее резервное копирование.

Чтобы безопасно удалить файлы журнала и по-прежнему иметь возможность выполнять катастрофическое восстановление, следует использовать утилиту **db_archive**:

1. Следует запустить обычное или горячее резервное копирование (Процедуры резервного копирования, стр. 9) и, прежде чем продолжить, убедиться, что все эти данные надежно сохранены на резервном носителе.
2. Следует выполните контрольную точку для получения дополнительной информации, если еще этого не было сделано (Контрольные точки, стр. 4).
3. Если поддерживается горячее резервное копирование, следует выполнить процедуру горячего резервного копирования (Использование горячих отказов, стр. 31).
4. Следует запустить утилиту **db_archive** с параметром **-d** для рабочего окружения.
5. Запустить утилиту **db_archive** с параметром **-d** для отказоустойчивого окружения, если оно поддерживается.

Настройка подсистемы ведения журнала

Можно настроить следующие аспекты подсистемы ведения журнала:

1) Размер файлов журнала.
2) Размер области подсистемы ведения журнала.
3) Хранить журналы полностью в памяти.
4) Размер буфера журнала в памяти.
5) Расположение файлов журнала на диске.

Установка размера файла журнала (Setting the Log File Size)

Всякий раз, когда в файл журнала записывается предопределенный объем данных (по умолчанию 10 МБ), DB прекращает использование текущего файла журнала и начинает запись в новый файл.

Можно изменить максимальный объем данных, содержащихся в каждом файле журнала, с помощью метода **`DB_ENV->set_lg_max()`**. Этот метод можно использовать в любое время в течение жизненного цикла приложения.

Установка размера файла журнала больше значения по умолчанию — это в основном вопрос удобства и отражение предпочтений приложения в отношении носителя и частоты резервного копирования.

Однако, если будет установлен слишком малый размер файла журнала по сравнению с моделями трафика приложения, можно нажить себе неприятности.

С точки зрения производительности, установка размера файла журнала в низкое значение может привести к тому, что активные транзакции будут приостанавливать свои действия по записи чаще, чем это произошло бы с файлами журнала большего размера.

Всякий раз, когда транзакция завершается, буфер журнала сбрасывается на диск. Обычно другие транзакции могут продолжать писать в буфер журнала, пока выполняется этот сброс. Однако, когда один файл журнала закрывается и создается другой, все транзакции прекращают запись в буфер журнала до тех пор, пока переключение не будет завершено.

Помимо проблем с производительностью, использование меньших файлов журнала может привести к использованию большего количества физических файлов на диске. В результате приложение может исчерпать порядковые номера журнала, в зависимости от того, насколько загружено приложение.

Каждый файл журнала идентифицируется 10-значным номером. Более того, максимальное количество файлов журнала, которые приложение может создать за время своего существования, составляет 2 000 000 000.

Например, если приложение выполняет 6000 транзакций в секунду в течение 24 часов в сутки, и вы регистрируете 500 байт данных за транзакцию в файлах журнала размером 10 МБ, то файлы журнала закончатся примерно через 221 год:

Однако, если приложение записывает 2000 байт данных за транзакцию и использует файлы журнала размером 1 МБ, то файлы журнала закончатся примерно через 5 лет.

Все эти временные рамки, конечно, довольно длительные, но если закончатся файлы журналов после, скажем, 5 лет непрерывной работы, то придется сбросить порядковые номера журналов. Для этого необходимо:

1. Создать резервную копию баз данных, как будто готовясь к катастрофическому сбою.
2. Сбросить порядковый номер файла журнала с помощью параметра **`-r`** утилиты **`db_load`**.
3. Удалить все файлы журнала из окружения.

Это единственная ситуация, в которой все файлы журнала удаляются из окружения; во всех остальных случаях сохраняется по крайней мере один файл журнала.

4. Перезапустить приложение.

Настройка размера области ведения журнала (Configuring the Logging Region Size)

Размер области подсистемы ведения журнала по умолчанию составляет 60 КБ. Область ведения журнала используется для хранения имен файлов, поэтому может потребоваться увеличить ее размер, если будет открыто и зарегистрировано в менеджере журналов БД большое количество файлов (в случае очень большого количества баз данных).

Можно задать размер области ведения журнала с помощью метода **`DB_ENV->set_lg_regionmax()`**.

Этот метод можно вызвать только до открытия первого дескриптора окружения для приложения.

Настройка ведения журнала в памяти (ACI)

Можно настроить подсистему ведения журналов таким образом, чтобы журналы хранились полностью в памяти.

Однако, следствием этого является отказ от гарантии транзакционной устойчивости – без файлов журналов нет возможности запустить восстановление, поэтому любые системные или программные сбои, которые могут возникнуть, могут повредить базы данных.

Тем не менее, отказавшись от гарантий долговечности, можно значительно улучшить пропускную способность приложения, избежав дискового ввода-вывода, необходимого для записи информации журнала на диск. В этом случае все равно транзакционные атомарности, согласованности и гарантии изоляции сохраняются.

Чтобы настроить подсистему журналирования для сохранения журналов полностью в памяти, следует:

- Убедиться, что буфер журнала способен хранить всю информацию журнала, которая может накапливаться во время самой длительной транзакции.

- Не следует при открытии окружения запускать обычное восстановление. В этой конфигурации нет доступных файлов журнала, по которым можно запустить восстановление. В результате, если при открытии окружения будет указано восстановление, оно будет проигнорировано.

- Указать **DB_LOG_IN_MEMORY** для метода **DB_ENV->log_set_config()**. Это следует указать до того, как приложение откроет свой первый дескриптор окружения.

Ниже 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_ENV *envp = NULL;
    const char *db_home_dir = "/tmp/myEnvironment";

    // Создание окружения
    ret = db_env_create(&envp, 0);
    if (ret != 0) {
        fprintf(stderr, "Ошибка создания дескриптора окружения: %s\n",
                db_strerror(ret));
        return (EXIT_FAILURE);
    }

    // Указываем, что журналирование должно производиться только в памяти.
    // Это означает, что мы отказываемся от гарантии надежности транзакций.
    envp->log_set_config(envp, DB_LOG_IN_MEMORY, 1);

    // Настраиваем размер буфера памяти журнала. Он должен быть достаточно
    // большим, чтобы вместить всю информацию для самой длительной транзакции.
    // Если журналирование выполняется в памяти, размер буфера журнала по
    // умолчанию составляет 1 МБ.
```

```
// В этом примере мы произвольно устанавливаем буфер журнала равным 5 МБ.
ret = envp->set_lg_bsize(envp, 5 * 1024 * 1024);
if (ret != 0) {
    fprintf(stderr, "Error setting log buffer size: %s\n",
            db_strerror(ret));
    goto err;
}
// Устанавливаем обычные флаги для транзакционной подсистемы.
// Мы НЕ указываем DB_RECOVER. Также следует помнить, что подсистема
// журналирования автоматически включается при инициализации транзакционной
// подсистемы, поэтому мы явно не включаем здесь журналирование.
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

// Открываем окружение как обычно
ret = envp->open(envp, db_home_dir, env_flags, 0);
if (ret != 0) {
    fprintf(stderr, "Ошибка открытия окружения: %s\n",
            db_strerror(ret));
    goto err;
}
```

```
    // Создаем транзакции и выполняем операции с базой данных.
    // Эта часть опущена для краткости.
    ...
err:
    // Закрываем базы данных (опущено)
   ...
    // 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 размера буфера журнала в памяти (Setting In-Memory Log Buffer Size)

Если приложение настроено на ведение журнала на диске (поведение по умолчанию для транзакционных приложений), информация журнала хранится в памяти до тех пор, пока не заполнится выделенное для этого пространство или транзакция не заставит записать информацию журнала на диск.

Можно увеличить объем памяти, доступной для буфера журнала файла. Это повышает пропускную способность для длительных транзакций или для транзакций, которые производят большой объем данных.

Если подсистема ведения журнала настроена на сохранение журнала целиком в памяти, очень важно настроить размер буфера журнала, поскольку буфер журнала должен быть способен хранить всю информацию журнала, которая может накапливаться в течение самой длительной транзакции.

Следует убедиться, что размер буфера журнала в памяти достаточно велик, чтобы ни одна транзакция не охватывала весь буфер. Также следует избегать состояния, когда буфер в памяти заполнен и не может быть освобождено место, поскольку транзакция, которая начала первый «файл» журнала, все еще активна.

Если подсистема ведения журнала настроена на ведение журнала на диске, размер буфера журнала по умолчанию составляет 32 КБ. Если настроено ведение журнала в памяти, размер буфера журнала по умолчанию составляет 1 МБ.

Можно увеличить размер буфера журнала с помощью метода **`DB_ENV->set_lg_bsize()`**. Этот метод можно вызвать только до открытия первого дескриптора окружения для приложения.

Резюме и примеры

Анатомия транзакционного приложения

Транзакционные приложения характеризуются выполнением следующих действий:

1. Создание дескриптора окружения.

2. Открытие окружения, указав, что будут использоваться следующие подсистемы:

- Транзакционная подсистема (это также инициализирует подсистему ведения журнала).
- Пул памяти (кэш в памяти).
- Подсистема ведения журнала.
- Подсистема блокировки (если приложение является многопроцессным или многопоточным).

Также настоятельно рекомендуется запустить обычное восстановление при первом открытии окружения.

Обычное восстановление проверяет только те журналы, которые необходимы для обеспечения согласованности файлов базы данных относительно информации, содержащейся в файлах журналов.

3. При желании можнго запустить любые служебные потоки, которые могут понадобиться. Служебные потоки можно использовать для периодического запуска контрольных точек или для периодического запуска детектора взаимоблокировок, если вы не хотите использовать встроенный детектор взаимоблокировок БД.

4. Открываются любые необходимые дескрипторы базы данных.

5. Запускаются рабочие потоки. Сколько из них нужно и как они разделяют свою рабочую нагрузку БД, полностью зависит от требований приложения. Однако любые рабочие потоки, которые выполняют операции записи, будут делать следующее:

a. Начинать транзакцию.
b. Выполнять одну или несколько операций чтения и записи.
c. Фиксировать транзакцию, если все прошло хорошо.
d. Прерывать и повторять операцию, если обнаружена взаимоблокировка.
e. Прерывать транзакцию для большинства других ошибок.
6. При завершении работы приложения:
a. Следует убедиться, что нет открытых курсоров.
b. Следует убедиться, что нет активных транзакций. Перед завершением работы необходимо либо отменить, либо зафиксировать все транзакции.
c. Следует закрыть базы данных.
d. Следует закрыть окружение.

Примечание

Надежные приложения БД должны контролировать свои рабочие потоки, чтобы убедиться, что они неожиданно не завершились. Если поток завершается ненормально, следует остановить все рабочие потоки, а затем запустить нормальное восстановление (для этого придется заново открыть окружение). Это единственный способ очистить любые ресурсы (например, блокировку или мьютекс), которые мог удерживать ненормально завершающийся рабочий поток в момент своей остановки.

Невыполнение этого восстановления может привести к тому, что все еще работающие рабочие потоки в конечном итоге будут заблокированы навсегда в ожидании блокировки, которая никогда не будет снята.

В дополнение к этим действиям, которые полностью обрабатываются кодом в приложении, есть некоторые административные действия, которые следует выполнить:

- Следует периодически создавать контрольные точки для приложения. Контрольные точки сократят время выполнения восстановления в случае необходимости. (Контрольные точки, стр. 4).

- Следует периодически создавать резервные копии базы данных и файлов журналов. Это необходимо для того, чтобы в полной мере получить гарантию долговечности, предоставляемую поддержкой транзакций ACID в базе данных. (Процедуры резервного копирования, стр. 9).

- Можно поддерживать горячее аварийное переключение, если важна круглосуточная обработка с быстрым перезапуском в случае сбоя диска. (Использование горячих аварийных переключений, стр. 31).