SQLite – это самостоятельная система.

SQLite не использует какие-либо внешние библиотеки или интерфейсы (кроме стандартных си-функций ниже).

Минимальная сборка БД требует наличие только функций из стандартной си-библиотеки функций:

* memcmp()
* memcpy()
* memmove()
* memset()
* strcmp()
* strlen()
* strncmp()

Большинство сборок использует еще системные функции управления памятью:

* malloc()
* realloc()
* free()

Но они могут быть отключены в параметрах компиляции.

Вызовы операционной системы - open(), read(), write(), fsync() и т.д. содержается в ВФС.

БД – это безсерверая система – процесс, который хочет получить доступ к БД читает/записывает данные напрямую с диска. Нет промежуточного серверного процесса.

Плюсы:

Нет необходимости устанавливать, настраивать, конфигурировать, инициализировать, управлять и исправлять серверную часть.

Нет необходимости в административной поддержке для настройки ядра БД. Любая программа, имеющая доступ к физ.памяти диска, где рамположена БД – имеет доступ к БД.

Минусы:

1. Серверная часть обеспечивает лучшую защиту от некорректных клиентских запросов – такие как – ошибочные указатели и т.д.
2. Один серверный процесс точнее контролирует доступ к БД, обеспечивая более тонкую блокировки и лучший параллелизм.

**Классичесская безсерверная БД и Нео-безсерверная БД**

1. Классическая безсерверная – ядро БД работает в одном процессе. Нет передачи каких-либо сообщений или сетевой активности.
2. Нео-безсерверная – ядро БД работает в другом процессе (может быть даже на другом компьютере), но БД представляется как сервис «под ключ» от хоста-провадера, нет необходимости в самостоятельном управлении или администрировании, и ее так легко использовать, что программисты могут думать, что у БД нет серверной части. Примеры: [Microsoft Azure Cosmo DB](https://docs.microsoft.com/en-us/azure/cosmos-db/serverless-computing-database) и [Amazon S3](https://aws.amazon.com/s3/).
3. Не требует установки
4. Не требует настройки
5. Нет серверной части (которую необходимо было бы стартовать, останавливать, или конфигурировать)
6. Не нужно назначать права доступа пользователям
7. SQLite не использует конфигурационных файлов вообще
8. Не требуется каких-либо действий по восстановлению БД после системной ошибки или отклюяения питания.

БД – это транзакционная БД – БД в которой каждое изменение представляется как атомарное, согласованное, изолированное и стойкое - **ACID**). SQLite реализует сериализуемые транзакции.

**А** – атомарность. Ниже.

**С** – согласованность. Каждая выполненная транзакция (или откат) фиксирует только допустимые результаты. В процессе выполнения транзакции, БД может быть не согласована.

**I** – изолированность Во время выполнения транзакции, параллельные транзакции не должны оказывать на ее результат.

**При параллельном выполнении транзакций возможны следующие проблемы:**

1. Потерянное обновление – при одновременном изменении одного блока данных разными транзакциями теряются все изменения, кроме последнего. Пример: есть два запроса на увеличение значения поля конкретной строки на 20 и 25 соответственно. Обе транзакции одновременно читают изначальное значение — 5. Обе вычисляют. Обе пытаются записать результат — 25 и 30 соответственно. В БД будет 25 или 30 в зависимости от того какая транзакция последней запишет результат в БД.

2. Грязное чтение – чтение данных, добавленных или измененных другой(ими) транзакциями, которые не факт, что успешно закончатся. Пример: транзакция 1 изменяет значения поля ф1, транзакция 2 выбирает значение этого поля — ф1, после выполняется откат транзакции 1, теперь данные, полученные транзакцией 2, отличаются от данных в БД.

3. Фантомное чтение – одна транзакция в ходе своего выполнения несколько раз выбирает множество строк по одним и тем же критериям. Другая транзакция в интервалах между этими выборками **добавляет** строки или изменяет столбцы некоторых строк, используемых в критериях выборки первой транзакции, и успешно заканчивается. В результате получится, что одни и те же выборки в первой транзакции дают разные множества строк. Пример: В транзакции 1 высчитывается сумма по столбцу ф1, транзакция 2 **добавляет** новую строку, повторное выполнение транзакции 1 выдаст другой результат. Отличие от неповторяющегося чтения — результат повторного обращения к данным изменился не из-за изменения/удаления самих данных, а из-за появления новых (фантомных) данных.

4. Неповторяющееся чтение — при повторном чтении в рамках одной транзакции ранее прочитанные данные оказываются **измененными**. Пример: пусть транзакция 1 выбирает значение ячейки ф1, транзакция ф2 изменяет эту ячейку, при повторной попытке выбора значения ячейки ф1в транзакции 1 будет получен другой результат.

**Уровни изолированности транзакций** – условные значения, определяющие в какой мере в результате выполнения логически параллельных транзакций в СУБД допускается получения несогласованных данных (стандарт SQL-92):

1. Чтение незафиксированных данных (read uncommited). Если несколько параллельных транзакций пытаются изменять одну и ту же строку таблицы, то в окончании всех транзакций, строка будет иметь значение, определенное ВСЕМ набором успешно выполненных транзакций. Реализация: блокировка данных на время изменения.

2. Чтение фиксированных данных (read commited). Предотвращает грязное чтение. Имеет две реализации:

* Блокировка читаемых и изменяемых данных. Транзакция, пишущая изменения в блок данных блокирует читающие этот блок данных транзакции.
* Сохранение нескольких версий параллельно изменяемых строк. При каждом изменении строки СУБД создает новую версию этой строки в ОП, в то время как читающей транзакции возвращается последняя версия этой строки.

3. Повторяемость чтения (repeatable read). Никакая другая транзакция не может изменить данные, читаемые текущей транзакцией.

4**. Упорядочиваемость (serializable).** Транзакции полностью изолируются друг от друга.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Уровень изоляции** | **Фантомное чтение** | **Неповторяющееся чтение** | **«Грязное» чтение** | **Потерянное обновление**[[3]](https://ru.wikipedia.org/wiki/%D0%A3%D1%80%D0%BE%D0%B2%D0%B5%D0%BD%D1%8C_%D0%B8%D0%B7%D0%BE%D0%BB%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D0%B8_%D1%82%D1%80%D0%B0%D0%BD%D0%B7%D0%B0%D0%BA%D1%86%D0%B8%D0%B9#cite_note-3) |
| **SERIALIZABLE** | **+** | **+** | **+** | **+** |
| REPEATABLE READ | - | + | + | + |
| READ COMMITTED | - | - | + | + |
| READ UNCOMMITTED | - | - | - | + |

Более высокий уровень изолированности соответствует лучшей согласованности данных, но может снижать количество физически параллельно выполняемых транзакций.

**D** – Стойкость. Независимо от проблем на нижних уровнях все изменения в одной транзакции в SQLite либо выполняются полностью, либо не выполняются вообще, даже если во время записи изменений на диск прерывается: аварийным завершением программы, аварийным завершением ОС, выключением питания.

**Об атомарности в SQLite**

Атомарное совершение означает, что, либо все изменения в БД в пределах транзакции будут выполнены, либо ни одно из изменений не будет выполнено. Причем при выполнении нескольких записей в различные секции на диске, все они выполняются мгновенно и одновременно.

Реальное аппаратное обеспечение СЕРИАЛИЗУЕТ операции записи в ЗУ, но сама запись сектора – минимальной единицы записи – занимает некоторое конечное время. Таким образом, можно говорить, что запись нескольких секторов действительно происходит одновременно и мгновенно. Но логика атомарного совершения SQLite делает так, что ВСЕ ИЗМЕНЕНИЯ В ТРАНЗАКЦИИ представляются мгновенными и одновременными. То есть ни ошибка ОС, ни отключение питания эту атомарность разрушить не могут.

Далее рассказано о технике SQLite, создающей иллюзию атомарного совершения транзакции при включенном режиме журнала отката.

**Аппаратные допущения и определения при изложении техники, описывающего иллюзию атомарного совершения транзакции**. Под диском будет пониматься ЗУ или флэш-память. Минимальная единица записи и чтения – сектор, НО это касается только ЗУ. У флеш-памяти, как правило, минимальный размер чтения меньше минимального размера записи. В дальнейшем изложении мы будем считать **сектор как минимальные данные для записи в ЗУ.** Чтобы изменить данные, меньше сектора, необходимо изменить часть сектора и затем записать на диск весь сектор с измененными и неизмененными данными.

До версии SQLite 3.3.14 сектор был 512 байт. Сейчас идет тенденция на изменение размера сектора до 4096 байт. Именно поэтому, начиная с версии 3.3.14, есть метод уровня ОС, который опрашивает нижележащую ФС для нахождения истинного размера сектора, который возвращает жестко заданное число – 512, так как пока нет унифицированного метода нахождения истинного размера сектора на Юникс и Виндовс ОС. Метод можно изменить.

Если углубляться в техническую реализацию техники, описывающего иллюзию атомарного совершения транзакции, то необходимо сказать, что **SQLite традиционно считает, что операция записи не атомарная**. Он считает, что запись линейна, то есть, аппарат записи начинает запись с одного конца данных для записи и побайтно записывает, пока не достигнет другого конца. Если случится отключение питания, то одна часть сектора окажется измененной, а другая нет. **SQLite считает, что, если какая-либо часть сектора изменилась, то, либо первый байт, либо последний байт изменен.** SQLite не знает, правильно ли он определяет изменение в секторе.

Стандартное утверждение, что «SQLite традиционно считает, что операция записи не атомарная» было единственным до версии 3.5.0. Начиная с версии 3.5.0 появились ВФС – единственная прослойка между SQLite и ФС ОС. В интерфейсе ВФС есть функция xDeviceCharacteristics, которая определяет является ли операция записи атомарной. Если это так, то SQLite пытается взять выгоду от этого утверждения. Но если это не так, то выгода пропускается.

SQLite предполагает, что ОС буферизирует записи и что запрос на запись вернется до того, как данные будут фактически сохранены в ЗУ. Также, SQLite предполагает, что операции записи могут быть поменяны местами ОС. По этой причине SQLite использует flush и fsync операции. SQLite предполагает, что операции **flush** и **fsync** не вернут возвращаемого значения до тех пор, пока все запрошенные операции записи не отработают. Возможно, то для некоторых версий Линукс и Виндовс эти операции не работают.

SQLite предполагает, что размер файла сначала повышается и затем данные пишутся в файл. Метод интерфейса ВФС – xDeviceCharacteristics –может показать всегда ли ФС сначала повышает размер файла и затем пишет туда данные. Если это так, то SQLite накладывает дополнительную защиту (описанную ниже). Однако в стандарных ВФС нет такого предположения.

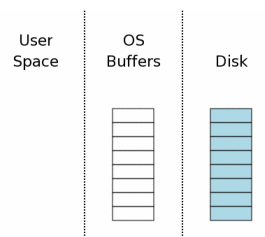
SQLite предполагает, что удаление файла атомарная, т.е., если при удалении файла, отключится питание, то после включения файла либо не будет видно в ФС вообще, либо остался не тронутым.

SQLite предполагает, что обнаружение и/или исправление битовой ошибки, вызванной космическими лучами, темпловым шумом, квантовой флуктуацией, багами ЗУ или другими механизмами, полностью лежит на аппаратном обеспечении и ОС. SQLite предполагает, что считанные данные абсолютно такие же, которые были раннее записаны.

По дефолту, SQL предполагает, что вызовы на запись данных операционной системой не тронут внешние данные даже в случае потери питания или ошибки ОС. SQLite называет это предположение характеристикой «powersafe overwrite». До версии 3.7.9 данной характеристики не было. Это предположение можно выключить параметром компилятора или прямо во время выполнения.

**Шаги, которые выполняет SQLite для поддержки атомарного совершения транзакции.**

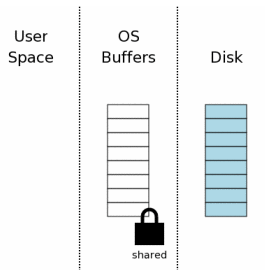
1. Изначальное состояние. Ниже представлен рисунок, где показано состояние компьютера при открытии соединения с БД. Disk – сектора в ЗУ. OS Buffers – ОП, User Space память процесса. Голубой цвет – данные заполнены корректными данными.



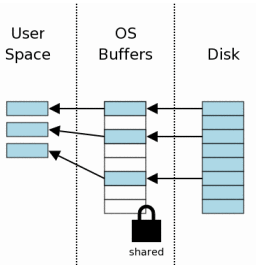
2. Приобретение блокировки на чтение. Теперь тем как SQLite может что-то записать, необходимо прочитать БД, чтобы посмотреть что там есть уже. Даже если вставляются новые данные, SQLite читает схему БД из таблицы sqlite\_master, чтобы узнать как парсить INSERT команду и обнаружить где в файле БД будет храниться новые данные.

Первый шаг для чтения данных из БД – получение разделяемой блокировки (shared lock). Разделяемая блокировка позволяет двум и более соединениям с БД читать из файла БД одновременно. Но не дает каким-либо другим соединениям с БД записывать информацию в файл БД, пока ее читают. Это необходимо, так как, если в один момент мы читаем строки, а другое соединение записывает или изменяет читаемые нами строки, мы получим недостоверную информацию. И таким образом, может показаться, что транзакция не атомарна.

Обратите внимание, что разделяемая блокировка храниться в ОП. Следовательно, блокировка исчезнет в случае потери питания. Также, обычно, блокировка исчезнет, если исчезнет процесс, который получил эту блокировку.

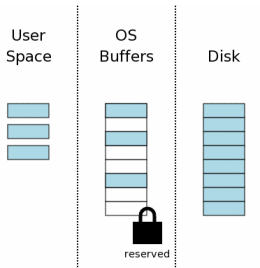


3. В этом сценарии мы предполагаем **холодный кэш**, то есть, данные сначала читаются из диска в ОП, затем из ОП в память процесса. Таким образом, последующие чтения могут найти данные в кэше, и тогда потребуется лишь передача данных из кэша в память процесса.



4. Приобретение зарезервированной блокировки. Перед какими-либо изменениями в БД, SQLite сначала приобретает зарезервированную блокировку. Зарезервированная блокировка, как и разделяемая блокировка, дает другим процессам читать из БД. Единственная зарезервированная блокировка может сосуществовать с несколькими разделяемыми блокировками. Таким образом, только один процесс может записывать информацию в БД.

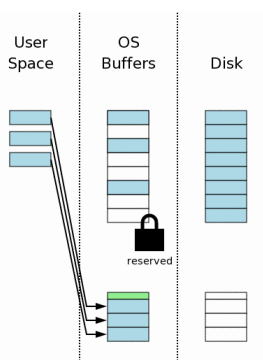
Идея зарезервированной блокировки состоит в том, что это сигнал, что процесс собирается скоро изменить БД. Другие процессы еще пока могут продолжать читать.



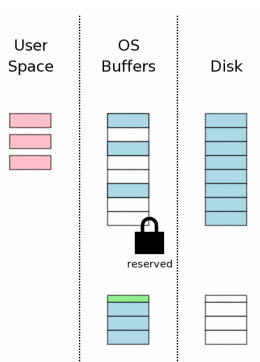
5. Создание журнала отката. Перед тем, как делать какие-либо изменения в БД, SQLite создает журнал отката и записывает туда изначальные (еще не модифицированные) данные из БД. Журнал содержит ВСЮ необходимую инф-ию для отката БД в изначальное (до модификации) состояние.

*Отступление. Когда создается новый файл, он фактически не записывается на диск. Он создается только в ОП. Файл не записывается сразу на диск, а лишь тогда, когда есть у диска время. Такой фокус создает ощущение, что ввод/вывод происходит намного быстрее, чем на самом деле.*

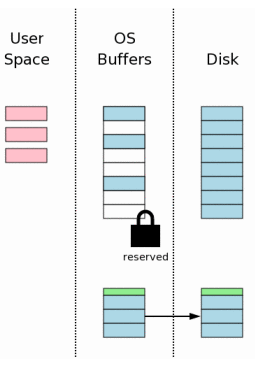
Согласно идее выше, сначала журнал отката создается в ОП.



6. Изменение страниц БД в памяти процесса. После сохранения изначальных данных в журнале отката, страницы можно менять. Каждое соединение с БД имеет свою память процесса, и, соответственно друг у друга память они не видят. Другие соединения (процессы) могут видеть СВОИ данные в ОП, которые еще не поменялись, и, таким образом, пока один процесс занят изменением данных в БД, другой может читать свои копии этих изначальных данных.



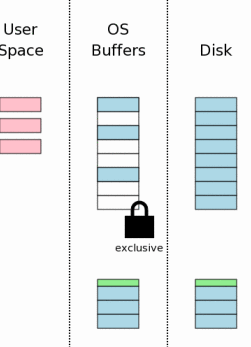
7. Запись (flush) журнала отката на диск. **Это критический шаг.** Также, это медленная операция (операция записи). Этот шаг немного сложнее, чем кажется. Первая запись записывает содержимое журнала (без заголовка). Затем, заголовок журнала в ОП меняется, теперь он показывает количество страниц в самом журнале. Почему так объяснено ниже.



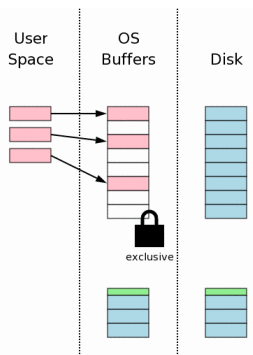
8. Приобретение эксклюзивной блокировки. Перед тем, как делать какие-либо изменения, мы должны получить эксклюзивную блокировку. На самом деле, получение эксклюзивной блокировки – это двухшаговый процесс. Вначале мы получаем ожидающую блокировку.

Ожидающая блокировка позволяет другим процессам, уже имеющим разделяемую блокировку, продолжать чтение БД. Но ожидающая блокировка предотвращает создание новых разделяемых блокировок. Идея ожидающей блокировки – предотвратить очень долгое ожидание писателя, вызванное огромным числом читателей. Может быть сотни процессов-читателей. Каждый процесс должен получить разделяемую блокировку перед чтением, прочитать что нужно и освободить блокировку. Если таких процессов-читателей очень много, то может произойти так, что новый процесс приобретает разделяемую блокировку еще до того, как предыдущий процесс освободил свою разделяемую блокировку. И, таким образом, не бывает момента, когда нет ни единой разделяемой блокировки в БД и, следовательно, не бывает возможности для писателя воспользоваться эксклюзивной блокировкой.

В конце концов, когда все оставшиеся разделяемые блокировки закончатся, тогда ожидающая блокировка вызовет эксклюзивную блокировку.



9. Запись изменений БД. Как только будет получена эксклюзивная блокировка, мы знаем, что никакой другой процесс больше не читает данные из БД и мы можем записать изменения в файл БД. Обычно, эти изменения вносятся сначала только в ОП.



10. Перенос изменений на диск. Этот шаг один из самых времязатратных (наравне с п.7).

11. Удаление журнала отката. После успешного внесения изменений БД на диск, журнал отката удаляется. Это тот момент, где транзакция завершается. Если произойдет отключение питания до этого шага, то процесс восстановления из журнала отката будет. Если это произойдет после – то это означает, что все изменения в БД внесены корректно.

Удаление файла, как было сказано ранее, не атомарная операция, но она представляется атомарной с т.з. пользовательского процесса. Процесс вправе постоянно спрашивать ОС «существует ли файл»? Процесс должен получить однозначный ответ да или нет. После отключения питания и его восстановления, SQLite спрашивает существует ли журнал отката? Если да, то транзакция была не довыполнена и происходит процесс отката, если нет, то все ок.

Так как удаление файла – времязатратное, то, в качестве оптимизации, SQLite может быть сконфигурирован следующими образами:

а. Сократить размер файла до 0 байт.

б. Перезаписать заголовок файла нулями. Как чаще всего бывает, журнал станет непригодным, даже если поменять в ней первый бит заголовка.

12. Освобождение блокировки. Это необходимо, чтобы другие процессы могли получить доступ к БД.

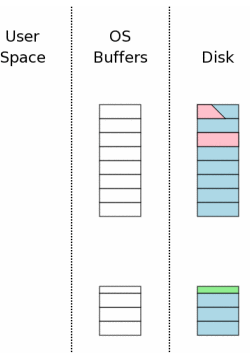
На рисунке видно, что информация стерлась из памяти процесса, когда блокировка снялась. Это было так для старых версий SQLite. Более новые версии держат память процесса в случае, если она понадобится при старте следующей транзакции.

Перед использованием данных, находящихся в памяти процесса, необходимо добавить разделяемую блокировку и затем убедиться, что никакой другой процесс не изменил БД. В первой странице БД есть счетчик, который инкрементируются каждый раз, когда файл БД модифицируется. Если изменился, то память процесса должна быть очищена.

ОТКАТ

1. Допустим, что при записи изменений в БД на диск (пункт 10 выше) произошла потеря питания. После починки, оказалось. Что одна запись изменилась полностью, вторая частично, а третья вообще не изменилась.

Журнал отката полный и не поврежден отключением питания. Это ключевой момент. Причина операции по п.7 выше – быть абсолютно уверенным, что журнал откат лежит в независимой от питания среде.



2. Горячие журналы отката. Первый раз, когда какой-либо процесс SQLite пытается получить доступ, он получается разделяемую блокировку. Но затем он замечает файл журнала отката. SQLite проверяет является ли этот журнал горячим. Горячий журнал формируется ТОЛЬКО, когда происходит фатальная ошибка или потеря питания.

Следующие утверждения истины для горячего журнала отката:

1. Журнал отката существует.

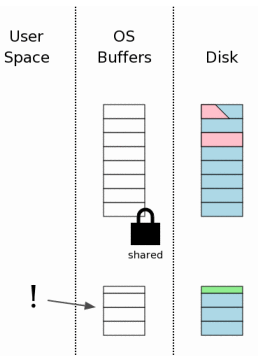
2. Журнал отката не пустой.

3. Нет зарезервированной блокировки в файле БД.

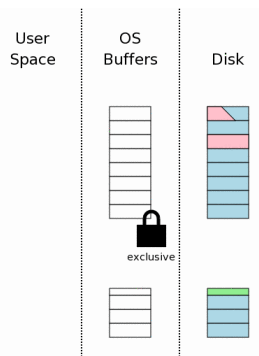
4. Заголовок журнала отката корректный и не перезаписан нулями.

5. Журнал отката не содержит имени файла мастера журналов (объяснено далее). Если содержит – мастер журналов существует.

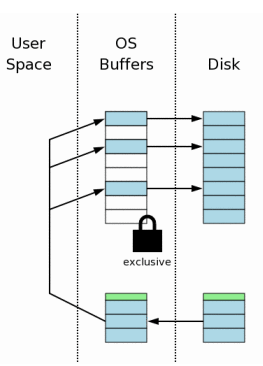
Значение горячего журнала отката состоит в том, что предыдущий процесс пытался совершить транзакцию, но был прерван до завершения транзакции.



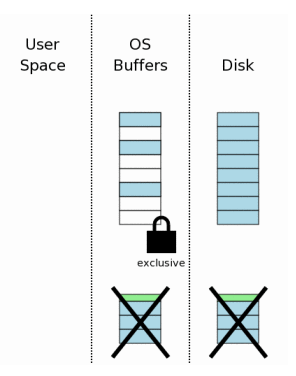
3. Получение эксклюзивной блокировки. Эта блокировка предотвращает двум и более другим процессам откатить по одному и тому же горячему журналу.



4. Откат изменений. После получения эксклюзивной блокировки, разрешено писать в БД. Помимо самих изначальных данных, содержащихся в журнале, в заголовке хранится изначальный размер БД. В конце этого шага БД должна иметь размер и содержать такие же данные, которые были до выполнения транзакции.



5. Удаление горячего журнала. См. п. 11 выше.



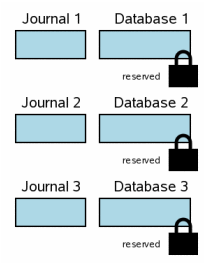
6. Продолжение работы, как будто незавершенной транзакции и не было. Последний шаг отката – освобождение эксклюзивной блокировки.

**МНОГОФАЙЛОВОЕ ВЫПОЛНЕНИЕ**

SQLite позволяет одному соединению общаться с двумя и более файлами баз данных одновременно, с помощью команды ATTACH. Когда изменяются несколько файлов БД, то изменяются ВСЕ файлы БД. Либо ВСЕ, либо НИКАКИЕ. Следующая технология рассказывает о том, как выполняется атомарное совершение с несколькими файлами.

1. Когда в одной транзакции замешены две и более БД, то каждая БД имеет свой журнал отката. На рисунке ниже показано, как данные в трех БД меняются в пределах одной транзакции. Каждая БД имеет свою зарезервированную блокировку. Для каждой БД, изначальные данные записаны в свой соответстввующий журнал отката, но сам журнал еще не записан на диск. Соответственно, никакие изменения еще не сделаны в самих БД на диске. Сделаны изменения только в памяти процесса аналог – п.6 выше, с одним фалом БД).

На рисунке не представлены отдеьные страницы БД. Голубой – изначаьные данные. Розовый – измененные данные. Не сделано различие между памятью в ОП и памятью на диске. Все детали также существуют и в многофайловом совершении транзакции.

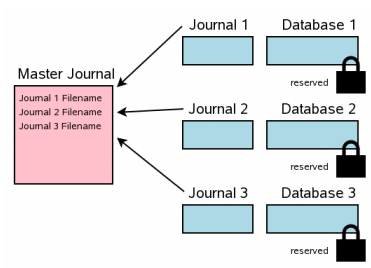


2. Мастер журнал. Следующий шаг в транзакции – создание мастер журнала. Имя мастер журнала – имя БД (та, которая изначально была открыта sqlite3\_open(), а не при помощи ATTAH) плюс –mj плюс 8 рандомных 16-ричных чисел. Такая формула наименования жернала может измениться в следующих релизах SQLite.

В отличие от журнала отката, мастер журнал не содержит изначальных данных баз данных, он содержит полные имена журналов отката для каждой БД, участвующей в транзакции.

После создания журнвла, он записывается на диск.

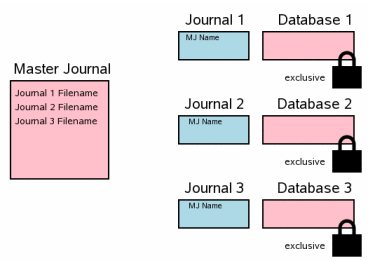
Мастер журнал можно отключить с помощью параметров компилятора – PRAGMA synchronous=OFF или PRAGMA journal\_mode=MEMORY.



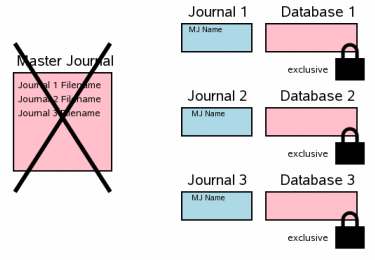
3. Обновление заголовков журналов отката. Следующий шаг – записать полное имя мастер журнала в заголовок каждой журнала отката. Память под это изначально резервировалась.

Содержание каждого журнала отката записывается на диск и только потом записывается имя мастер журнала в заголовок каждого журнала отката на диск (это аналог п.7 при совершении транзакции в одной БД).

4. Изменение файлов БД. После того, как все журнала записаны на диск, можно обновлять БД. Необходимо получить эксклюзивную блокировку для каждой БД. После, идет запись изменений БД в ОП и потом на диск (п. 8,9,10 выше).



5. Удаление мастер журнала. Это критически важный шаг. Если произойдет потеряпитани при удалении мастер журнала – то отката не произойдет, даже если журналы отката будут присутствовать. Все потому, что в заголовке журналов отката есть путь к мастер журналу. Если бы там не было пути к мастер журнала (это сценарий в один файлом БД), то откат бы произошел.



6. Удаление журналов отката. В этот шаг входит удаление журналов отката и освобождение эксклюзивных блокировок. Удаление журналов отката не критический шаг, так как транзакция уже совершена и не будет откатана назад. Алгоритм удаления следующий: первый жкрнал отката удаляется, блокировка для соответствующей БД убирается, затем идет удаление второго журнала отката и т.д. В будущем порядок могут изменить.

ДОПОЛНИТЕЛЬНЫЕ ДЕТАЛИ ПРОЦЕССА СОВЕРШЕНИЯ ТРАНЗАКЦИИ

1. Заполненные секторы журнала оката. Когда записывается нетронутые изначальные данные в журнал оката, записываются целые секции, даже если страница меньше размера сектора. Например, допустим, что в секторе хранятся страницы 1,2,3 и 4. Транзакция меняет данные второй страницы, тогда аппаратное обеспечение должно также перезаписать 1,3 и 4 страницы.

2. Как быть с мусором, записанным в журналы отката. Когда данные присоединяются в уже записанным данным в жкрнале отката, SQLite делает предположение, что сначала файл увеличивает свой размер (соответственно после увеличения размера соержит мусорные данные на конце) и затем заменяет мусорные данные корректными данными. Если произойдет отключение питания в момент, когда будет только увеличен размер файла, то после восстановление другой процесс SQLite заметит журнал отката и изменит данные в БД на мусорные.

SQLite содержит два способа избежать этого:

1. SQLite записывает количество записанных в журнал отката страниц в заголоке журнала отката. Изначально число 0. Таким образом, при попытке откатить БД в помощью неполного журнала отката, процесс делающий откат заметит в заголовке журнала, что страниц ноль. После того как страницы журнала отката перенесены на диск, изменяется кол-во страниц в заголовке журнала. Сектор в заголовком всегда держится в отдельном секторе от самих страниц, соержащих изначальные данные БД. Помните, что журнал отката записывается на диск двумя разами. Сначала страницы, потом заголовок.

Вышеописанные действия истины для параметра компилятора PRAGMA synchronous=FULL. Если установлено PRAGMA synchronous=NORMAL, то SQLite записывает журнал за один раз. Риск понятен, счетчик страниц может записаться раньше самих страниц, так как SQLite допускает, что запросы на запись на диск могут смешаться из-за нижележащей ФС.

2. SQLite использует 32-битовую контрольную сумму для каждой страницы в журнале. При откате процесс проверяет каждую контрольную сумму. Если окажется, что контрольная сумму некорректна, то откат прерывается. Конено, есть маленькая вероятность, что контрольная сумма окажется одинакова и для некорректных данных.

3. Преждевременная сохранения данных из памяти процесса на диск. Вышеописанные механизмы совершения атомарной транзакции предполагают, что все изменения данных в БД помещаются в память процесса. Если это не так, то должна быть возможность перенести эти данные в БД на диск до завершения транзакции.

В начале переноса данных из памяти процесса состояние как в п.6 при совершении атомарной транзакции с одним файлом БД: изначальные данные БД сохранены в журналы отката, измененные данные лежат в памяти процесса. Чтобы перенести эти измененные данные на диск SQLite выполняет шаги 7 и 9, т.е. журнал отката записывается на диск, выдается эксклюзивная блокировка. Далее, заголовок журнала отката присоединяется на диск и эксклюзивная блокировка сохраняется. Эти шаги повторяются всякий раз. Когда надо перенести данные из памяти процесса на диск.

ОПТИМИЗАЦИИ. Профайлинг показывает, что большинство ОС в большинстве случаев тратят львиную долю времени на ввод/вывод на диск. Следующие оптимизации пытаются минимизировать ввод/вывод на диск:

1. До версии 3.3.14 после убирания разделяемой блокировки вся память процесса должна была быть стерта. Так было, потому что без разделяемой блокировки другие процессы могли свободно изменять БД. И, таким образом, данные в памяти процесса могли быть уже не корректны. В версиях позднее 3.3.14 память процесс остается, когда блокировка БД освобождается. Далее, когда создается разделяемая блокировка в начале транзакции, SQLite проверяет не модифицировал ли кто-либо БД после снятия разделяемой блокировки процесса. Если да, то память процесса очищается.

Для того чтобы узнать изменилась ли БД или нет, SQLite использует счетчик в заголовке БД, который увеличивается всякий раз, когда изменяется БД. Перед освобождением разделяемой блокировки процесс сохраняет копию значения счетчика на данный момент. И тогда следующая разделяемая блокировка сравнивает сохраненное значения счетчика с текущим.

2. Режим эксклюзивного доступа. Начиная с версии 3.3.14 добавлен концепт режима эксклюзивного доступа. В этом режиме SQLite сохраняет эксклюзивную блокировку для каждой транзакции. Тако режим не дает другим процессам получить доступ к БД, но и во многих случаях только один процесс всегда и взаимодействует с БД. Такой режим может уменьшить ввод/вывод на диск несколькими способами:

1) Нет необходимости изменять каждый раз счетчик изменений БД.

2) Никакой другой процесс не может получить доступ к БД, следовательно не надо проверять счетчик изменений БД, следовательно не надо очищать каждую транзакцию память процесса.

3) Каждая транзакция завершается перезаписыванием нулей в заголовок журнала отката. Это избавляет от необходимости изменять запись каталога для файла журнала (шта?) и избавляет от необходимости освобождать сектора, на которых был записан журнал. Более того, следующая транзакция перезапишет содержимое старого журнала, что быстрее, чем добавление новых данных.

Третья оптимизация не связана с постоянным задержанием эксклюзивной блокировки.

3. Не публикуйте страницы freelist. Когда данные из БД удаляются/модифицируются, удаленные/старые данные помещаются в фрилист. Следующие вставки данных могут взять страницы из фрилиста и использовать их для хранения данных, а не увеличивать размер БД.

Некоторые страницы фрилиста содержат критические данные, а именно ссылку на другие страницы фрилиста. Но чаще всего страницы содержат ненужную информацию. А последние страницы называются листьями. Мы может менять данные в листьях, при этом не портя данные в БД.

Так как содержимое листье не важно, SQLite не включает эти данные в журнал отката. Похожим образом, данные новой страницы фрилиста никогда не будут записаны обратно в БД на шаге 9 механизма совершения транзакции в одной БД, а также никогда не прочитаются из БД на шаге 3 того же механизма.

Эти оптимизации сиьно сокращают ввод/вывод на диск.

4. Обновление одной страницы и атомарные записи сектора. С версии 3.5.0 появился интерфейс xDeviceCharacteristics, который, помимо всего прочего, может указать на возможность атомарной записи сектора.

SQLite сам по дефолту через этот интерфейс узнает запись сектора атомарна или линейна. SQLite верит, что большинство современных дисков реализуют атомарную запись сектора. Когда теряется питание, диск используется энергию конденсаторов и момент импульса диска, чтобы закончить текущую операцию. Как бы то ни было, этих слоев между запросом на запись на диск и самим дисковым устройством так много, что SQLite сохранил подход в ВФС для Юникс и Виндовс, где полагается не атомарная запись сектора.

Если все-таки запись атомарна и размер страницы БД равен размеру сектора и, когда изменения затрагивают только одну страницу в БД, тогда SQLite пропускает все журналирование изначальных данных и процесс синхронизации и просто записывает напрямую измененные страницы прямо в БД на диск.

ФСы с семантикой безопасного добавления. Еще одна, уже нам изветсная оптимизация, оптимизация, связанная с размером файла и дальнейшем его наполнении. Начиная с версии 3.5.0 SQLite предоставляет свое безопасное добавление данных в файл. SQLite предполагает, что, когда надо добавить данные в файл, сначала увеличивается размер файла и затем это пространство заполняется корректными данными. Если произойдет потеря питания между увеличение размера файла и добавление корректных данных, то файл будет содержать некорректные данные. Интерфейс xDeviceCharacteristics ВФС может определить реализует ли ФС безопасное добавление данных в файл.

Когда это безопасное добавление данных включено в ФС, SQLite хранит -1 в счетчике страниц в заголовке журнала отката. Это говорит процессу, который собирается удалить журнал, что количество страниц в журнале должно быть вычислено из размера журнала. Таким образом, будет лишь один запрос на запись страниц журнала на диск (не понятно).

6. Настойчивые журналы отката. Как мы знаем, удаление файла это времязатратная операция. SQLite может быть сконфигурирован так, что, вместо удаления файла, он может, либо поставить размер файла в ноль, либо перезаписать заголовок нулями. Сокращение размера файла до нуля сохраняет необходимость обновлять папку, содержащую удаляемый файл. Перезапись заголовка позволяет не обновлять размер файла (и как следствие не обновлять папку), а также не иметь дело в ново-выделенным пространством для записи, так как перезапись быстрее добавления.

Конфигурация SQLite для перезаписи заголовка журнала отката нулями: PRAGMA journal\_mode=PERSIST

Минус этой конфигурации – это то, что журналы отката будут оставаться на диске и занимать пространство.

Конфигурация SQLite для полного удаления журнала отката:

journal\_mode=DELETE

BEGIN EXCLUSIVE

COMMIT

Конфигурация SQLite для сокращение размера файла до нуля:

journal\_mode=TRUNCATE

Плюс в том, что не затрагивается папка, содержащая файл. Дополнительный плюс в том, что не используется системный вызов например, fsync() для синхронизации изменений на диске. Конечно, это может быть и минус, но современные ФС представляют сокращение размера файла как атомарную и синхронную (хз что это значит) операцию.

ТЕСТИРОВАНИЕ ПОВЕДЕНИЯ АТОМАРНОГО СОВЕРШЕНИЯ. Разработчики SQLite уверены, что он надежен в условиях сбоев электропитания и сбоев системы, поскольку процедуры автоматического тестирования выполняют обширные проверки способности SQLite восстанавливаться после симулированной потери питания.

Для таких краш-тестов испольуется специальный измененный ВФС, кторый может симулировать различные повреждения ФС, окторые порисходят во время потери питания или подения ОС. ВФС для краш-тестов может сэмулировать неполностью записанный сектор, страницы заполненные мусором, запись за границами. Краш-тесты выполняться снова и снова, варьируя время сэмулированной потери питания и различные параметры наносимого ущерба. После каждого теста БД проверяется на корректность, т.е. была ли выполнена транзакция или произошел откат.

Некоторые найденные баги были очень смутные и вряд ли бы их удалось найти с помощью осмотра кода и методов анализа.

ВЕЩИ, КОТОРЫЕ МОГУТ ПОЙТИ НЕ ТАК.

1. Некорректная реализация блокировки. SQLite использует блокировки на уровне ОС, чтобы убедиться точ только один процесс и одно соединение может изменить БД. Механизм блокировки реализуется в ВФС и может меняться в зависимости от ОС. SQLite зависит от корректности реализации механизма блокировки. Если что-то пойдет не так, то два и более процесс смогут изменить БД одновременно.

SQLite получил отчет о плохой работе блокировок для Windows network FS и NFS.

2. Неполное запись на диск. SQLite использует системный вызов fsync для Юникс и FlushFileBuffers для win32 для того, чтобы записать данные на диск. К сожалению, SQLite получил много отчетов, что ни один из этих интерфейсов не работает на многих системах. Они слышали, что FlushFileBuffers может быть отключен с помощью реестра в некоторых версиях Виндовс. Даже та тех, системах, где эти системные вызовы работают, часто IDE диск врет и говорит, что данные уже записаны, хотя они еще только в ОП.

3. Частичные удаления файла. Как уже говорилось ранее, SQLite предполагает, что операция удаления файла – атомарная операция. Но транзакция удаления файла может оказаться не атомарной там, где не работает эта атомарность.

4. Мусорные данные в файлах. Файлы БД SQLite - это обычные файлы ФС. Любой, кто имеет доступ к файлу на уровне ФС может открыть файл и внести туда некорректные данные. Повредить данные может также баги в ОС или контроллере диска, а особенно – потеря питания.

5. Удаление или переименование горячего журнала. Если произошла ошибка ОС или потеря питания, после восстановления работы на уровне ОС, обязательно необходимо откатить БД, если существует горячий журнал отката. Обязательным условием отката является, чтобы перед первым подключением БД, имя БД и имя горячего журнала отката остались прежними. Во время отката, SQLite ищет журнал в папке с БД и с правильным именем.

Существует много жестких ссылок на файл БД, журнал отката создается по одной ссылке БД. И если наступит сбой, после восстановления БД будет открыта через другую жесткую ссылку, то горячий журнал не найдется.

Бывает и такое, что при потере питания повреждается ФС и имена недавно измененных файлов забываются. Тогда файлы помещаются в папку lost+found. Когда такое происходит – горячий журнал не найдется.

ЛОГИРОВАНИЕ С ОПЕРЕЖЕНИЕМ ЗАПИСИ.

Стандартный меод, который реализует SQLite – журнал отката. Начиная с версии 3.7.0 появилась новая опция – лоигрование с опережением записи (WAL).

У этой новой опции есть как плюсы. Так и минусы:

1. ЛОЗ в большинстве сценариев значительно быстрее.

2. ЛОЗ предоставляет больший параллелизм, так как читатели не блокируют писателей, а писатель не блокирует читателей. Чтение и запись может происходить параллельно.

3. Операции ввода/вывода на диск более последовательны в ЛОЗ.

4. ЛОЗ использует намного меньше операций fsync() и менее зависим от систем, где fsync некорректен.

Минусы:

1. Обычно ЛОЗ требует от ВФС поддержки примитивов резделяемой памяти. Исключение: ЛОЗ без разделяемой памяти. Встроенная в Юникс и Виндовс ВФСы поддержка этих примитивов есть, чего нельзя сказать о ВФСах сторонник разработчиков.

2. Все процессы, использующие конкретную БД должны находится на одном компьютере. ЛОЗ не работает через сеть.

3. Транзакции являются атомарными только в пределах одной БД, присоединенной к соеиднению при помощи команды ATTACH. Транзакция не является атомарной в пределах двух и более БД.

4. Нет возможности изменить размер страницы в БД, не важно является ли БД пустой или используется команда VACUUM или БД восстановлена через бэкап. Чтобы изменить размер страницы надо использовать режим журнала отката.

5. Начиная с версии 3.22.0 БД с режимом ЛОЗ только для чтения может быть открыта, если файлы –shm и –wal уже существуют, либо они могут быть созданы или БД закрыта для изменений.

6. ЛОЗ может быть на1-2% медленней журнала отката в приложениях, где часто выполняются чтение и редко запись.

7. Существуют дополнительные файл –wal и файл разделяемой памяти –shm соотнесенные с каждой БД, которые могут сделать SQLite менее привлекательной для использования ее в качестве формата приложения.

8. В ЛОЗ есть дополнительная операция сохранения, которая по дефолту выполняется автоматически.

9. Начиная с версии 3.11.0, режим ЛОЗ работает также эффективно с большими транзакциями, как и режим журнала отката.

КАК РАБОТАЕТ ЛОЗ.

ЛОЗ использует режим журнала отката наоборот: Начальные данные остаются в БД на диске, а измененные данные добавляются в ЛОЗ. Завершение транзакции наступает тогда, когда в ЛОЗ добавляется специальный индикатор записи. Завершение транзакции может произойти и без записи измененных данных на диск, что позволяет читателям продолжать использовать БД. Несколько транзакций может добавлять данные в конец ЛОЗа.

СОХРАНЕНИЕ. Конечно, в один момент кто-то захочет перенести все изменения из ЛОЗ на диск. Это и называется сохранением.

По дефолту, SQLite автоматически сохраняет ЛОЗ на диск, когда размер достигает 1000 страниц. Параметр компилятор SQLITE\_DEFAULT\_WAL\_AUTOCHECKPOINT может изменить это число. Также можно отключить автоматическое сохранение и включать его, когда БД простаивает.

ПАРАЛЛЕЛЬНОСТЬ В ЛОЗ. При выполнении операции чтения в БД в режиме ЛОЗ, он, в первую очередь, запоминает адрес последней успешно записанной страницы в ЛОЗ. Назовем эту точку «конечная метка». ЛОЗ может расти в размерах добавляя новые записи пока читатели присоединяются к БД, таким образом, каждый читатель имеет потенциально свое значение конечной метки. Но для каждого конкретного читателя значение конечной метки не меняется на протяжении конкретной транзакции, это гарантирует, что один читатель видит содержимое БД такое, какое оно есть в начале транзакции.

Когда читателю нужна страница из БД, он в первую очередь ищет эту страницу в ЛОЗ, если она там есть, он берет последнюю ее копию, которая находится до конечной метки читателя. Если нет страницы в ЛОЗ до конечной метки, то страница читается из диска. Читатели могут быть из разных процессов, поэтому, чтобы избежать последовательного чтения ЛОЗа каждым читателем (ЛОЗ может весить и несколько Мб), структура данных, называемая, индекс-ЛОЗ, обслуживаемый в резделяемой памяти, помогает читателям находить страницы в ЛОЗ быстро и с минимальным вводом/выводом на диск. Индекс-ЛОЗ позволяет значительно увеличить производительность чтения, но использование разделяемой памяти означает, что все читатели должны находится на одном компьютере.

Писатели просто добавляют новые данные в конец ЛОЗа. Так как писатели не делают ничего такого, что может пересекаться с читателями, и те и другие могут работать одновременно. Однако, если ЛОЗ один, то писатель может в один момент времени может быть только один.

Операция сохранения может использоваться параллельно с читателями, но сохранение должно остановиться там, где страница достигает конечной метки хотя бы одного читателя, так как, в противном случае, сохранение может перезаписать ту часть БД, которую читатель активно использует.

Казалось бы, долгий процесс чтения может надолго остановить сохранение, но рано или поздно транзакция закончится и сохранение, если оно необходимо, продолжится.

Когда в любой момент появляется операция чтения, писатель проверяет как много сделало сохранение: если весь ЛОЗ перенесен на БД и синхронизирован и если читатели не используют ЛОЗ, то писатель перематывает ЛОЗ на начало и начинает добавлять новые транзакции в начало ЛОЗ. Такой механизм предотвращает бесконечное возрастание ЛОЗ.

ПРОИЗВОДИТЕЛЬНОСТЬ. Транзакция записи в ЛОЗ очень быстрая, так как запись происходит один раз (в отличие от двух сохранений страниц потом заголовка журнала отката) и еще потому что запись идет последовательно. К тому же, синхронизация данных на диске не требуется, если приложение готово пожертвовать долговечностью после потери питания или горячей перезагрузки. (Писатели синхронизируют ЛОЗ при включенном в FULL параметре компилятора synchronous, либо синхронизация пропускается при значении параметра NORMAL.

С другой стороны, ухудшается производительность чтения ЛОЗ так как он увеличивается в размерах, и каждый читатель должен проверить содержимое ЛОЗ, а время проверки пропорционально его размеру. ЛОЗ-индекс позволяет находить данные в ЛОЗ намного быстрее, но производительность все равно падает при увеличении размера ЛОЗ. Поэтому, чтобы поддерживать хорошую производительность чтения необходимо периодически сохранять ЛОЗ.

Сохранение требует операции синхронизации, чтобы избежать вероятность поломки БД из-за потери питания или горячей перезагрузки. ЛОЗ должен быть синхронизирован в постоянное хранилище перед переносом данных из ЛОЗ на диск и еще БД должна быть синхронизирована перед сохранением ЛОЗ. Сохранение требует проводить много поисков. Сохранение старается записать на диск как можно больше страниц, но даже в этом случае будет много операций поиска по страницам в ЛОЗ. Эти факторы делают сохранение медленней записи.

По стандартная стратегияSQLite - позволить транзакциям записывать данные в ЛОЗ пока ЛОЗ не достигнет 1000 строк, затем выполнить сохранение для каждой совершенной транзакции пока ЛОЗ не станет меньше 1000 строк. То есть, по дефолту, сохранение вызывается автоматически потоком, который совершает транзакцию и из-за него ЛОЗ становится больше 1000 строк. Таким образом получается, что совершение записи происходит очень быстро, но та запись, которая увеличивает ЛОЗ больше 1000 строк – медленная. Можно настроить сохранение в другом процессе/потоке.

Отмечено, что при параметре компилятора synchronous=NORMAL сохранение – это единственна операция, которая создает барьер ввода/вывода или операцию синхронизации (fsync или FlushFileBuffers). Если приложение выполняет сохранение в отдельном процессе или потоке, то главный процесс или поток, выполняющий запросы к БД или изменяющий ее, никогда не заблокируют операцию синхронизации. Это помогает избежать «latch-up» в приложениях, работающих на нагруженном диске. Минус такой конфигурации – транзакции больше не надежны и могут быть откатаны из-за потери питания или горячей перезагрузки.

Стоит также отметить, что есть обратная взаимоисключение между средней производительностью чтения и средней производительностью записи. Чтобы максимально увеличить производительность чтения необходимо держать размер ЛОЗ как можно меньше и тем самым делать сохранения как можно чаще, возможно при каждом совершении транзакции. Чтобы максимально увеличить производительность записи, необходимо уменьшать затраты на каждое сохранение на как можно большем количестве записей, то есть, необходимо редко использовать сохранение и дать ЛОЗ расти.

АКТИВАЦИЯ И КОНФИГУРАЦИЯ РЕЖИМА ЛОЗ. Pragma journal\_mode=WAL.

АВТОМАТИЧЕСКОЕ СОХРАНЕНИЕ. Иногдаприложения, чтобы больше контролировать сохранения могут использовать [wal\_checkpoint pragma](https://www.sqlite.org/pragma.html#pragma_wal_checkpoint) или использовать метод [sqlite3\_wal\_checkpoint()](https://www.sqlite.org/c3ref/wal_checkpoint.html). Кол-во страниц дл атвоматического сохранение можут быть изменено или отключено используя [wal\_autocheckpoint pragma](https://www.sqlite.org/pragma.html#pragma_wal_autocheckpoint) или используя метод [sqlite3\_wal\_autocheckpoint()](https://www.sqlite.org/c3ref/wal_autocheckpoint.html). Приложение также может использовать [sqlite3\_wal\_hook()](https://www.sqlite.org/c3ref/wal_hook.html) чтобы зарегистрировать обратный выхов, который вызывается при завершении какой-либо транзакции в ЛОЗ. Обратный вызов затем вызывает [sqlite3\_wal\_checkpoint()](https://www.sqlite.org/c3ref/wal_checkpoint.html) или [sqlite3\_wal\_checkpoint\_v2()](https://www.sqlite.org/c3ref/wal_checkpoint_v2.html) смотря какой критерий корректный. (Мехнизм автоматического сохранения – это просто обертка вокруг [sqlite3\_wal\_hook()](https://www.sqlite.org/c3ref/wal_hook.html).)

Сохранение, активированные приложением. Приложение может вызвать сохранение, вызвав sqlite3\_wal\_checkpoint() или sqlite3\_wal\_checkpoint\_v2(). Есть три вида сохранения: PASSIVE, FULL, RESTART. Стандартный – PASSIVE, то есть, который сохраняет столько сколько сожет, не мешая другим соединениям.

Настойчивость режима ЛОЗ: В отличие от режима журнала отката, если закрыть БД и потом открыть, то режим ЛОЗ сохраняется. В режиме журнала отката, если заново открыть БД, то режим вернется в дефолтный DELETE.

Смысл такой настойчивости состоит в том, что можно один раз выставить параметр компилятора PRAGMA journal\_mode=WAL и далее не указывать этот параметр, тогда, без изменения кода, БД поменяет режим.

ФАЙЛ ЛОЗ. Имя лоз обычно состоит из имени БД плюс «-wal». Могут быть и другие правила наименования.

ЛОЗ существует до тех пор пока не закроется последнее соединение с последней БД, тогда он удаляется. Однако, если последний процесс с соединением с БД закрылся, но явно не закрыл соединение с БД или используется контрол файла SQLITE\_FCNTL\_PERSIST\_WAL, тогда ЛОЗ м.б. записан на диск после того как закроются все соединения. ЛОЗ это часть БД и должна храниться вместе с БД, если их разделить, то транзакции, которые ранее были выполнены могут быть потеряны или БД сломается. Единственно надежный способ убрать ЛОЗ, то открыть соединение и потом его закрыть.

БД ТОЛЬКО ДЛЯ ЧТЕНИЯ. Начиная с версии 3.22.0 БД в режиме чтения с ЛОЗ или БД в режиме ЛОЗ без разрешения на запись могут быть прочитаны, если соблюдаются следующие условия:

1. Файлы –shm и –wal существут и они читаемы.

2. Есть права на запись в папке, где находятся эти файлы.

3. БД открыта с параметром immutable – режим только для чтения.

ИЗБЕГАНИЕ ОЧЕНЬ БОЛЬШОГО РАЗМЕРА ЛОЗ. Как уже быол сказано ранее, обычно новые данные добавляются в ЛОЗу, пока ЛОЗ не достигнет 1000 строк, это триггер по которому SQLite автоматически выполняет сохранение. Сохранение не сокращает размер ЛОЗа автоматически. Напротив, он позволяет SQLite преписывать ЛОЗ с начала. Это так сделано, потому что, как правило, перезапись быстрее записи. Когда последнее соединение с БД закрывается, SQLite выполняет последнее сохранение и удаляет ЛОЗ и соответствующий ЛОЗу файл разделяемой памяти.

Таким образом, в большинстве случает, разработчику не надо беспокоиться и ЛОЗе. Но возможно сделать так, что файл ЛОЗ будет расти до предела. Ниже представлены способы устранить такую проблему:

1. Отключить механизм автоматического сохранения. Можно изменить количество страниц, при котором активируется сохранение. Либо вообще его отключить.

2. Голодание сохранения. Сохранение может дойти до конца и сбросить ЛОЗ, если нет никаких других соединений, использующих ЛОЗ. Если какое-либо подключение читает ЛОЗ, то сохранение не может выполнить удаление данных из ЛОЗ, так как такое действие может удалить нужные читателю данные.

Однако, если БД имеет много параллельных читателей, то нет ни одного момента, когда нет читателей в БД и тогда ЛОЗ будет расти бесконечно.

Такой сценарий можно обойти, если кто-то из подключений захочет выполнить ручное сохранение, присет SQLITE\_CHECKPOINT\_RESTART может не до конца дойти до конца ЛОЗ, а SQLITE\_CHECKPOINT\_TRUNCATE дойдет, но с возможными блокировками читателей.

Очень большие транзакции записи. Сохранение может закончится полностью, если никакие другие транзакции не выполняются, что означает, что ЛОЗ не может быть сброшен на середине транзакции записи в ЛОЗ. Именно поэтому, большие изменения в БД могут привести к большому размеру ЛОЗа.ЛОЗ можно сохранить лишь тогда, когда операция записи закончится, но во время ожидания, ЛОЗ может сильно вырсти в размерах.

РЕАЛИЗАЦИЯ РАЗДЕЛЯЕМОЙ ПАМЯТИ ДЛЯ ЛОЗ-ИНДЕКСА. ЛОЗ-индекс реализован как обычный файл, это сделано для надежности. Ранее ЛОЗ хранился в незащищенной разделяемой памяти. Проблема ранего подхода это то, что процессы с различным корневым каталогом будут видеть различные файлы, а следовательно использовать различные пространства разделяемой памяти, приводя к поломке БД. Другие методы создания безымянного блока разделяемой памяти не переносимы из-за различных особенностей юникса. И SQLite не может найти метод создания безымянного блока разделяемой памяти для Виндовс. Единственное что SQLite нашел, чтобы гарантировать, что все процессы, подключенные к одной БД, имеют одну и ту же разделяемую память – это создать разделяемую память, привязав файл в той же папке, где и БД. Такой подход имеет минус: он может делать необзяательные вводы/выводы к диску. Однако, разработчики не думают, что это большая проблема, так как ЛОЗ-индекс редко достигает 32Кб и никогда не синхронизируется. Более того, бэкап ЛОЗ-индекса удаляется, когда последнее соединение отключается, что часто не позволяет случится вводу/выводу вообще.

Спициализированные приложения, которым не подходит стандартная реализация разделяемой памяти могут разработать альтернативные методы, используя самописную ВФС. Например, известно, что определенные БД могут быть доступны только через поотоки в одном процессе, ЛОЗ-индекс может быть реализован, используя кучу, вместо разделяемой памяти.

ИСПОЛЬЗОВАНИЕ ЛОЗ БЕЗ РАЗДЕЛЯЕМОЙ ПАМЯТИ.

Начиная с версии 3.7.4, БД с ЛОЗ могут создаваться, читаться и писаться, если даже не доступна разделяемая память до тех пор, пока установлен режим блокировки EXCLUSIVE до первой попытки доступа к БД. Другими словами, процесс может взаимодействовать с БД ЛОЗ без использования разделяемой памяти, если этого процесс гарантирует, что он будет единственным процессом, у которого есть доступ к БД. Это свойство дает БД ЛОЗ быть созданными, прочитанными и записанными самописными ВФС без версии 2 методов разделяемой памяти xShmMap, xShmLock, xShmBarrier и xShmUnmap объекта sqlite3\_io\_methods.

Если режим блокировки EXCLUSIVE включен перед первым досутпом к БД, то SQLite никогда не будет пытаться вызвать методы разделяемой памяти и следовательно никакой ЛОЗ-индекс в разделяемой памяти не создастся. В этом случае, такой режим блокировки остается до тех пор, пока метод БД – ЛОЗ; попытки изменить режим через PRAGMA locking\_mode=NORMAL не сработают. Есдинстенный способ изменить режим блокировки – изменить режим БД - ЛОЗ - на другой.

Если режим блокировки NORMAL действует для первого доступа к БД, то ЛОЗ-индекс в разделяемой памяти создастся. Это означает, что ВФС должен поддерживать методы разделяемой памяти 2 версии. Если ВФС не содержит методы разделяемой памяти, то попытки открыть БД, которая уже в режиме ЛОЗ, или попытки сконвертировать БД в режим ЛОЗ, будут неудачны. Но пока только одно соединение использует ЛОЗ-индекс разделяемой памяти, режим блокировки можно свободно менять между NORMAL и EXCLUSIVE. Это тот единственный случай когда ЛОЗ-индекс пропускается, когда режим блокировки EXCLUSIVE перед первым доступом к БД с ЛОЗ, режим блокировки застревает в EXCLUSIVE.

ИНОГДА ЗАПРОСЫ ВОЗВРАЩАЮТ SQLITE\_BUSY В РЕЖИМЕ ЛОЗ. Второй плюс режима ЛОЗ – писатели не блокируют читателей и читатели не блокируют писателей. Это чаще всего правда. Но есть некоторые неочевидные случаи, где запросы на БД в режиме ЛОЗ могут возвратить SQLITE\_BUSY, так что приложения должны быть к этому готовы.

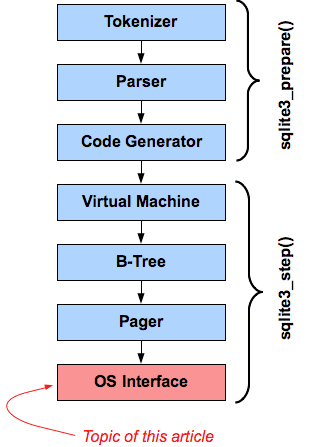
Это следующие случаи:

1. Если еще одно соединение к БД откроется в режиме болкировки – EXCLUSIVE, то все запросы к БД возвратят SQLITE\_BUSY. БД Chrome и Firefox открывают свои БД в режиме блоктровки EXCLUSIVE, так что попытки прочитать БД Chrome и Firefox пока приложения работают могут привести к проблеме.

2. Когда последнее соединение к БД закрывается, в процессе получая эксклюзивную блокировку на короткое время для очищения ЛОЗ и разделяемой памяти. Если второе соединение попытается открыть БД и дать ей запрос, пока первый еще находится в процессе очищения, то второе соединение может вернуть SQLITE\_BUSY.

3. Если последнее соединение к БД упадет с ошибкой, второе соединение начнет процесс восстановления. Эксклюзивная блокировка держится во время восстановления. Так что третье соединение попытается соединиться и отправить запрос, пока второе соединение выполняет процесс восстановления, третье соединение может выдать SQLITE\_BUSY.

ВФС. Внутренняя структура SQLite может быть предтавлена, как н рисунке ниже:



Компоненты Tokenier, parser и code generator используются для обработки запросов SQL и конвертирование их в исполняемые программы на языке ВМ или в байт-код. Грубо говоря эти три слоя реализуют функцию sqlite3\_prepare\_v2(). А байт-код, сгенерированный на третьем слое – подготовленный запрос. Модуль «виртуальная машина» ответственнен за выполнение байт-кода SQL-запроса. Модуль «B-Tree» организует файл БД в различные объекты «ключ-значение», хранящиеся с последовательными ключами и логарифмической производительностью. Модуль «Pager» ответственный за загрузку страниц файла БД в ОП для реализации и контроля транзакций, и для создания и обслуживания журналов отката. Модуль «Интерфейс ОС» ИЛИ ПО-ДРУГОМУ ВФС это тонкий слой, который сопровождает общий набор функций для адаптации выполнения SQLite на различных ОС. Грубо говоря, последние 4 слоев реализуют функцию sqlite\_step().

Далее речь пойдет о последнем слое.

Когда какому-либо слою нужно пообщаться с ОС, они вызывают методы ВФС.ВФС затем вызывает специфичные для конкретной ОС функции. Следовательно, для портирования SQLite необходимо написать новый ВФС.

Стандартное дерево исходников SQLite содержит встроенные ВФСы для Юникс и Виндовс. Альтернативные ВФСы можно добавить статично или динамично используя sqlite3\_vfs\_register().

Несколько ВФСов могут быть зарегистрированы одновременно. Каждый ВФС имеет уникальное имя. Отедльные соединение с БД в одном процессе могут использовать различные ВФСы.

Стандартные ВФСы Виндовс. Стандартный ВФС для Виндовс – win32. Другие ВФСы:

1. win32-longpath – это как стандартный, только имена путей могут быть до 65534 байт, в о время как в стандартном до 1040 байт.

2. win32-none – все операции блокировки отключены.

3. win32-longpath-none – комбинация двух первый ВФСов.

Как определить какую ВФС использовать. Всегда есть стандартная ВФС: unix для unix-подобных ОС и win32 – для Виндовс. При создании нового соединения с БД применяется стандартная ВФС.

Стандарную ВФС можно изменить с помощью sqlite3\_vfs\_register() или 4-ым параметром sqlite3\_open.

Если активированы имена URI, то можно указать ВФС так: ATTACH ‘file:demo.db?vfs=unix-none’ AS demo;

Прокладка ВФС. С т.з. более высоких слоев стека SQLite, каждое соединение с БД использует только один ВФС. Но на самом деле, конкретный ВФС может быть только тонкой оберткой вокруг другого ВФС, который делает реальную работу. Этот тонкий слой и называется прокладка.

Простой пример прокладки – ВФС «vfstrace». Эта ВФС пишет сообщения, связанные с каждым вызовом методов ВФС в лог-файл, затем передает управление другой ВФС для продолжения обработки метода.

Ниже представлены другие ВФСы из дерева исходников:

1. appendvfs.c – позволяет добавляет БД в конец какого-либо файла.

2. test\_demovfs.c – помощник в изучении ВФС для разработки собственной ВФС, это ВФС для Юникс.

3. test\_quota.c – эта прокладка ВФС устанавливает совокупный размер набора файлов БД.

4. test\_multiplex.c – данная прокладка ВФС позволяет превышать размер файла БД нижележащей ФС.

5. test\_onefile.c – эта ВФС показывает как можно использовать ВФС во встроенных устройствах без ФС.

6. test\_journal.c – данный файл реализует прокладку ВФС, которая тестирует и проверяет, что БД и журнал отката пишется в правильном порядке и что синхронизация (запись на диск из ОП) проводится в нужное время, для того чтобы гарантировать, что БД можно будет восстановить после потери питания.

7. test\_vfs.c – данная прокладка ВФС может быть использована для симуляции ошибок ФС.

Реализации ВФС. Новые ВФС реализуются путем создания подклассов трех объектов:

1. sqlite3\_vfs()

2. sqlite3\_io\_methods()

3. sqlite3\_file()

Объект sqlite3\_vfs() определяет имя ВФС и корневые методы, которые реализуют интерфейс с ОС (проверка существования файла, удаление файлов и т.д.). Также объект содержит методы получения случайности с ОС, для удержания процесса и для нахождения текущей даты\времени.

Объект sqlite3\_file представляет открытый файл. Метод xOpen объекта sqlite3\_vfs конструирует объект sqslite3\_file, когда файл открыт. Sqlite3\_file держит состояние файла, пока файл открыт.

Объект sqlite3\_io\_methods содержит методы для взаимодействия в открытым файлом. Каждый sqlite3\_file содержит указатель на объект sqlite3\_io\_methods, который соответствует представленному файлу. Sqlite3\_io\_methods содержит методы для чтения, записи, усечения размера, записи изменения на диск, нахождения размера файла, закрытия файла и уничтожения объекта sqlite3\_file.

Написание кода новой ВФС вовлекает создание подкласса для объекта sqlite3\_vfs и затем регистрацию ВФС с помощью sqlite3\_register(). Реализация ВФС также сопровождает подклассы sqlite3\_file и sqlite3\_io\_methods, но эти объекты не регистрируются напрямую SQLite. Вместо этого, объект sqlite3\_file возвращается методом xOpen() объекта sqlite3\_vfs, а объект sqlite3\_file указывает на инстанцию объекта sqlite3\_io\_mehods.

UNIFORM RESOURCE IDENTIFIERS

Начиная с версии 3.7.7 аргумент БД интерфейсов sqlite3\_open(), sqlite3\_open16(), sqlite3\_open\_v2() и команда ATTACH могу быть определены либо как обычное имя файла, либо как URI. Преимущество использования URI заключается в том, что параметры запроса URI могут быть использованы для детального контроля создаваемого соединения с БД. Например, альтернативный ВФС может быть определен через параметр запроса “vfs=”.

Обратная совместимость. Для обеспечения обратной совместимости URI по умолчанию отключен. URI имена файлов могут быть включены с помощью параметра компилятора SQLITE\_USE\_URI=1. Также можно включить URI в коде sqlite3\_config(SQLITE\_CONFIG\_URI,1). Можно включть конфиг для отдельного соединения вставив в параметр F константу SQLITE\_OPEN\_URI метода sqlite3\_open(N,P,F,V);

SQLite познает имя файла независимо включен ли URI или нет (file:1.db или 1.db).

Формат URI. Согласно RFC 3986, URI состоит из схемы, хоста, пути, строки запроса и фрагмента.

SQLite использует синтаксис URI “file=” для идентификации файла БД. Кратко о правилах парсинга URI:

1. Схема URI должна быть “file:”, любые другие схемы рассматриваются SQLite как обычное имя файла.

2. Хост может быть пропущен, может быть пустым или “localhost”. Любое другое – ошибка. Если SQLite скомпилирован с параметром SQLITE\_ALLOW\_URI\_AUTHROTY, тогда любые другие имена хостов будут передаваться ОС как имя файла UNC.

3. Путь указывается, если представлен хост.

4. Если строка запроса представлена, то все параметры запроса передаются ВФС.

5. Не обязателен.

Ноль и более escape-последовательностей формы %НН (где Н представляет 16-ричное число) могут встретиться в пути, запросе или фрагменте.

Путь URI. Путь URI определяет папку диска, где находится БД. Если путь пропущен, то БД пишется во временную папку, и она удаляется после закрытия соединения. Если указан хост, то путь должен быть абсолютным. Если не указан хост, то абсолютным, если первый символ / или относительный в любом другом случае.

Правила перевода любого пути в URI:

1. Заменить ? на %3f

2. Заменить # на %23

3. Только в виндовс, заменить \ на /

4. Заменить все последовательности двух и более / на один /

5. Только в виндовс, если путь зачинается с символа диска, то добавить в начало /

6. Добавить в начало схему file:

Строка запроса. Строка запроса делится на пары ключ/значение – параметры запроса. Параметры отделяются одним знаком &. Ключ отделяется от значение знаком =.

Текст параметров запроса добавляется к аргументу имени файла метод xOpen. Все escape-последовательности разрешаются перед добавлением. Один нулевой байт отделяет аргумент «имя файла» метода xOpen от ключа первого параметра запроса, ключ от значения, каждый последующий ключ от предыдущего значения. Список параметров запроса для добавления в xOpen заканчивается один ключом нулевой длины. Значение параметра запроса может быть нулевым.

Распознавание параметров запроса. Все параметры отправляются в метод xOpen ВФСа.

Параметры запроса для версии 3.15.0

Cache, immutable, mode, modeof, nolock, psow, vfs.

ФОРМАТ ФАЙЛА БД.

Далее идет описание и определение файла БД начиная с версии 3.0.0.

Файл БД. Законченное состояние БД обычно включает один файл на диске, называемый главный файл БД. Во время выполнения транзакции, SQLite хранит инф-ию в журнале отката или в ЛОЗ файле.

Горячие журналы.

Страницы. Главный файл БД состоит из одной и более страниц. Размер страницы 2 в какой-либо степени. Все страницы одно БД одного размера. Размер страницы для файла БД определяется как 2-байтовое целое со смещением в 16 байт с начала файла БД.

Нумерация страниц начинается с 1. Максимальное кол-во страниц – 2^31-2. Минимальный размер БД SQLite - это страница в 512 байт. Максимальный размер БД – 140 Тб.

В любой момент времени, каждая страница БД используется по одному из следующих назначении:

1. Страница с фиксированным размером.

2. Страница свободного списка

2.1. Стволовая страница свободного списка

2.2 Листовая страница свободного списка

3. Страница бинарное дерево

3.1. Внутренняя страница таблицы бинарного дерева

3.2. Листовая страница таблицы бинарного дерева

3.3. Внутренняя страница индекса бинарного дерева

3.4. Листовая страница индекса бинарного дерева

4. Страница переполнения полезной нагрузки

5. Страница карты указателя

Все чтения и записи в БД начинаются в границе страницы и все записи – целое число страниц. Чтения тоже чаще всего целое число страниц за исключением, когда БД первый раз открывается читается первые 100 байт файла БД (заголовок файла БД).

Перед изменением любой страницы, хранящей информацию, изначальная нетронутая информация записывается в журнал отката. Если транзакция прерывается и необходим откат, журнала отката используется для обновления БД в изначальное состояние. Листовые страницы свободного списка не содержат полезную информацию, которую необходимо восстанавливать после поломки БД, поэтому эта бесполезная информация не содержится в журнале отката, а также чтобы уменьшить ввод/вывод диска.

ЗАГОЛОВОК БД.

Первые 100 байт файла БД – это заголовок БД. Он состоят и элементов указанных ниже.Каждый элемент хранится в big-endian.

|  |  |  |  |
| --- | --- | --- | --- |
| Сдвиг | Размер, байт | Описание | Дополнительно |
| 0 | 16 | Строка «SQLite format 3\000» | Каждая корректная БД SQLite начинается с 16 байт – последовательности 16-ричных чисел, означающих в кодировке UTF-8 «SQLite format 3» |
| 16 | 2 | Размер страницы в байтах. От 512 до 32768, 1 означает 65536 | До версии 3.7.0.1 – это было число (2 в степени) от 512 до 32768. Начиная с версии 3.7.1 появилось число 65536, но оно в 2 байта не помещается, поэтому обозначается 1 BE. Или можно представить 2-байтовое число как LE и сказать, что оно представляет размер страницы, деленная на 256 |
| 18 | 1 | Версия записи формата файла. 1 – самописный, 2 – WAL | Должны давать возможность для улучшения формата файла в будущем в будущих версиях SQLite |
| 19 | 1 | Версия чтения формата файла. 1 – самописный, 2 – WAL |
| 20 | 1 | Количество зарезервированных байт в конце каждой страницы | Это необходимо, например, для хранения контрольной суммы каждой страницы. Необходимо помнить, что минимальный размер страницы – 480 байт |
| 21 | 1 | Максимальная доля встроенной полезной нагрузки. Д.б. 64 | Это значения предполагались как настраиваемые параметры для изменения формата хранения алгоритма бинарного дерева. Пока такой функционал не поддерживается |
| 22 | 1 | Минимальная доля встроенной полезной нагрузки. Л.б. 32 |
| 23 | 1 | Доля полезной нагрузки листа. Д.б. 32 |
| 24 | 4 | Счетчик изменения файла | Инкрементируется, когда БД освобождается от блокировки после внесения изменений в БД. При использовании ЛОЗ, изменения отслеживаются индексом ЛОЗ. |
| 28 | 4 | Размер файла БД в страницах | Если значение некорректно, то размер считается просмотром непосредственно файла БД. Это нововведение. Значение считается корректным, если: оно не равно нулю и «Счетчик изменения файла» соотносится с «номером корректной версии» SQLite. Если самописная версия SQLite забудет про обновление этой ячейки заголовка, то и размер БД по данному значению будет некорректным, но еще и самописный код забудет обновлять значение «номер корректной версии» SQLite. Тогда данная ячейка не будет использоваться для получения размера БД |
| 32 | 4 | Номер страницы первой стволовой страницы свободного списка | Неиспользуемые страницы файла БД (страницы, данные которых были удалены пользовательскими запросами) хранятся в свободном списке. Данная ячейка заголовка хранит номер первой страницы свободного списка. А следующая ячейка – общее кол-во страниц свободного списка |
| 36 | 4 | Общее количество страниц свободного списка |
| 40 | 4 | Куки схема | Инкрементируется каждый раз, когда изменяется схема БД. Подготовленный запрос (созданный sqlite3\_prepare()) компилируется согласно какой-то версии схемы БД. Когда схема БД изменяется, подготовленный запрос должен быть переподготовлен. Когда подготовленный запрос выполняется, в сначала проверяет куки схемы чтобы убедиться, что значение осталось такое же, какой было и при подготовке схемы. |
| 44 | 4 | Номер формата схемы. 1,2,3,4 | Номер формата SQL  1 – понимают все версии SQLite 3.0.0  2 – Добавляет функциональность ALTER TABLE…ADD COLUMN  3 – добавляет возможность иметь добавленным столбцам при помощи ALTER TABLE…ADD COLUMN non-NULL значения по умолчанию  4 – добавляет ключевое слово DESC и булевы типы записи (сериальные типы 8 и 9). Используется по умолчанию |
| 48 | 4 | Стандартный размер страницы кэша | Рекомендуемый размер кэша файла БД в страницах |
| 52 | 4 | Номер самой большой корневой станицы бинарного дерева режиме авто-вакуум или инкрементальный-вакуум, в других режимах – 0 |  |
| 56 | 4 | Кодировка текста БД. 1 – utf8, 2 – utf16le, 3 – utf16be |  |
| 60 | 4 | Версия пользователя, задаваемая прагмой |  |
| 68 | 4 | ID приложения | Устанавливается прагмой для идентификации БД как относящейся или принадлежащей некоторому приложению. Он предназначен для файлов БД как файл-формат приложения |
| 92 | 4 | Записанное значение счетчика изменений, транзакции, записавшей значение SQLITE\_VERSION\_NUMBER. Оно показывает для какой транзакции актуален номер версии |  |
| 96 | 4 | Хранит значение SQLITE\_VERSION\_NUMBER для недавно изменившей файл БД библиотеки SQLite |  |
| Остальные байты – на будущее и д.б. пока нулю | | | |

СТРАНИЦА БЛОКИРОВКИ. Это одна страница в файле БД для реализации специфичной для ОС ВФС для реализации примитивов блокировки файла БД. Сама SQLite не использует страницу блокировки.

СВОБОДНЫЙ СПИСОК. Файл БД может включать одну или несколько страниц, которые не находятся в использовании, например, которые удалены. Все они хранятся в свободном списке и могут переиспользоваться, когда требуются дополнительные страницы.

Свободный список организован как список внутренних страниц, где каждая внутренняя страница содержит номера нуля и более страниц-листьев.

Внутренняя страница свободного листа состоит из массива целых чисел. Размер массива может достигать максимум все используемое пространство страницы. Первое целое число страницы – номер следующей внутренней страницы, либо ноль, если это последняя внутренняя страница. Второе целое число страницы – это номер листьевой страницы. Вызовите второе целое число внутренней страницы Л. Если Л больше нуля, то целые числа массива с индексами между [2;Л+1] содержат номера листьевых страниц.

Листьевые страницы не содержат информации.

СТРАНИЦЫ В-ДЕРЕВА.

Все страницы би дерева одного из следующих форматов:

1. Страница би дерева таблицы. Каждый лист содержит ключ (соответствующий rowid) и до 1Гб данных. Внутренняя страница содержит ключ (также соответствующий rowid) и указатели на потомков.

2. Страница би дерева индекса. Каждая страница содержит ключ до 1 Гб.

Ячейка = ключ + самый левый указатель на потомков, это для внутренней таблицы би дерева.

Ячейка = ключ (rowid) + данные, это для листьев би дерева таблицы.

Ячейка = ключ, это для листьев би дерева индекса.

Полезная нагрузка ячейки – нефиксированная часть ячейки. Ниже представлены части страниц би дерева — полезная нагрузка.

1. Би дерево таблицы

a. Внутренняя страница – нет.

b. Лист – данные.

2. Би дерево индекса

a. Внутренняя страница – ключ.

b. Лист – ключ.

Если полезная нагрузка превысила некий порог, то остальная часть помещается в ссылочный список страниц переполнения данных.

В файле БД у каждой rowid-таблицы, в т.ч у системной sqlite\_schema, м.б. только одно би дерево таблицы. У sqlite\_schema корневая страница – номер 1. Sqlite\_schema содержит номер корневой страницы для каждой таблицы и индекса в БД. Полезная нагрузка – ключ.

Для каждого индекса в схеме м.б. только одно би дерево индекса.

В виртуальных таблицах нет би деревьев.

Without rowid-таблица имеет только би дерево индекса.

Страница би дерева делится на следующие части:

1. Заголовок БД (только на первой странице).

2. 8 у листьев и 12 байтный заголовок у внутренних страниц би дерева:

a) Тип страницы,

b) начало первого свободного блока. Свободный блок – учет пустых мест на странице. Структура: 2 байта – указатель на следующий свободный блок, 2 байта – размер свободного блока. Если кусок свободного места < 4 байт, то он образует фрагмент.

c) количество ячеек на странице,

d) начало содержимого ячейки,

e) количество фрагментированных свободных байт,

f) 4 байт – самый правый указатель. Указателей у листа нет.

1. Массив указателей ячеек – массив смещений к содержимому соответствующих ячеек.

2. Не выделенная память. Место между последним указателем на ячейку из массива указателей ячеек и первой ячейкой.

3. Содержимое ячеек.

4. Зарезервированное пространство в конце каждой страницы, задается параметром в заголовке БД.

Пустое пространство страницы би дерева = не выделенное пространство + все свободные блоки + все фрагменты.

Виды ячеек:

Ячейка листа би дерева таблицы: байт полезной нагрузки, rowid, первая часть данных (или все данные, если переполнения нет), ссылка на первую страницу переполнения данных (если переполнение есть).

Ячейка внутренней страницы би дерева таблицы: указатель на левого потомка, rowid.

Ячейка листа би дерева индекса: байт полезной нагрузки, первая часть ключа (или весь ключ, если переполнения нет), ссылка на первую страницу переполнения данных (если переполнение есть).

Ячейка внутренней страницы би дерева индекса: указатель на левого потомка, байт полезной нагрузки, первая часть ключа (или весь ключ, если переполнения нет), ссылка на первую страницу переполнения данных (если переполнение есть).

Количество полезной нагрузки, которое идет на страницы переполнения данных зависит от типа страницы. У каждого типа страницы (которые имеют полезную нагрузку) есть максимальный и минимальный пороги размера полезной нагрузки, при котором остальная часть полезной нагрузки идет в страницы переполнения данных. Это необходимо для сохранения, как минимум, 4 ключей на одной странице би дерева индекса.

Карта указателей. Это доп. страницы в БД для увеличения эффективности режимов auto\_vacuum и incremental\_vacuum, когда страницы свободного списка перемещаются в конец файла БД и файл БД усекается для удаления страниц свободного списка при каждом выполнении транзакции. Страницы ptrmap содержат связки, идущие от потомка к родителю. Карта представляет собой массив 5-байтных элементов.

**СЛОЙ СХЕМЫ**.

Эта глава показывает, как низкоуровневое би дерево используется для реализации высокоуровневых возможностей SQL.

**Формат записи**. Ранее говорилось, что ключи в странице би дерева расположены в логическом порядке. Данная секция пояснит что такое логический порядок.

Полезная нагрузка всегда в формате записи. Запись – последовательность значений, соответствующих столбцам таблицы или индекса. Формат записи определяет количество столбцов, тип и содержимое каждого столбца.

Запись состоит из заголовка (общее кол-во байт в заголовке, сериальные типы столбцов) и тела (значения каждого столбца).

**Порядок сортировки записей**. Порядок ключей в би дереве индекса определяется порядком сортировки записей, которых представляют ключи. Сравнение записей идет по столбцам. Столбцы записи рассматриваются слева на право. Первая неодинаковая пара определяет порядок этих двух записей. Порядок сортировки: NULL, числа, текст, BLOB.

Функции сравнения: BINARY (дефолтный), NOCASE, RTRIM, sqlite3\_create\_collation().

**Представление SQL таблиц**. Каждая обычная таблица SQL в схеме БД представляется би деревом таблицы. Каждое вхождение в би дереве таблицы соотносится со строкой таблицы SQL. Rowid – ключ каждого вхождения би дерева таблицы.

Данные каждой строки в таблице SQL хранятся в файле БД с помощью объединения значений различных столбцов в массив байт (формат записи), затем хранение массива байт в виде полезной нагрузки во вхождении би дерева таблицы. Порядок значений в записи такой же как и в SQL описании таблицы. Когда появляется алиас rowid, этот столбец в записях появляется в виде null.

Ели тип столбца – REAL, и столбец содержит значение, которое можно конвертировать в integer без потери информации, то столбец может быть записан как integer. SQLite будет конвертировать значение в плавающую точку при вытаскивании строки.

**Представление without rowid таблицы**. Такая таблица использует би дерево индекса. Ключ каждой страницы би дерева – запись, состоящая из столбцов с PK, следующая за всеми оставшиесями столбцами. Столбцы с РК в порядке, который был объявлен с PRIMARY KEY, а остальные столбцы в порядке, объявленном в CREATE TABLE.

Разница с таблицей rowid:

1. Вначале идут столбцы с РК

2. Данные используются как ключ в би дереве индекса. А не как данные в би дереве таблицы.

**Подавление избыточных столбцов с РК without rowid таблиц**.

CREATE TABLE t1(a,b,c,d,PRIMARY KEY(a,c)) WITHOUT ROWID);

CREATE TABLE t1(a,b,c,d,PRIMARY KEY(a,c,a,c)) WITHOUT ROWID);

CREATE TABLE t1(a,b,c,d,PRIMARY KEY(a,A,a,C)) WITHOUT ROWID);

CREATE TABLE t1(a,b,c,d,PRIMARY KEY(a,a,a,a,c)) WITHOUT ROWID);

Везде будет созданы РК столбцы а с

**Представление индексов SQL**. Каждый SQL индекс – это би дерево индекса, не важно как оно создано (через CREATE INDEX, UNIQUE или PRIMARY KEY у without rowid таблиц). Ключ у би дерева индекса – запись, состоящая из ключа таблицы и индексированных столбцов таблицы. Для обычных таблиц ключ – rowed, для without rowid – PRIMARY KEY. В нормальных индексах, есть однозначное соответствие между строками в таблице и страницами в би дереве индекса. В частичных индексах, би дерево содержит страницы би дерева индекса, соответствующая строкам таблицы, для которых проходит условие CREATE INDEX **WHERE**.

**Подавление избыточных столбцов во вторичных without rowid индексах**.

CREATE TABLE ex25(a,b,c,d,e,PRIMARY KEY(d,c,a)) WITHOUT rowid;

CREATE INDEX ex25ce ON ex25(c,e); - каждая строка будет ко столбцами c e d a, т.к. первые два столбца – индексируемые, а остальные – столбцы с РК.

CREATE INDEX ex25acde ON ex25(a,c,d,e); - индекс будет состоять толко из a c d e.

CREATE INDEX ex25ae ON ex25(a COLLATE nocase,e);

Подавление избыточности проходит только для without rowid таблиц. Для обычных таблиц с rowed вхождение би дерева индекса всегда заканчивается rowed (уникальным значением таблицы) и не важно есть ли в индексе полный РК.

Хранение схемы БД SQL

…

Without rowid ВТ. С версии 3.14, в запросе на создание ВТ CREATE TABLE добавилось слово WITHOUT ROWID. Это необхоимо, если необходим составной РК.

Метод xConnect, в отличие от xCreate – создает соединение к существующей ВТ, а не создает ВТ.

Метод xBestIndex –позволяет найти наилучший способ доступа к ВТ. Структура: указатель на ВТ, структура sqlite3\_index\_info.

**СОЕДИНЕНИЯ С БД В МНОГОПОТОЧНОМ ПРИЛОЖЕНИИ**

1. Первый поток – вызов sqlite3\_open()

2. Последующие – использование команды ATTACH

3. В конце – sqlite3\_close()

**РЕКОМЕНДУЕМЫЕ ПАРАМЕТРЫ КОМПИЛЯЦИИ**

SQLITE\_DQS=0

SQLITE\_THREADSAFE = 1 или 2

SQLITE\_DEFAULT\_MEMSTATUS=0

~~SQLITE\_DEFAULT\_WAL\_SYNCHRONOUS~~

SQLITE\_LIKE\_DOESNT\_MATCH\_BLOBS

SQLITE\_MAX\_EXPR\_DEPTH=0

SQLITE\_OMIT\_DECLTYPE

SQLITE\_OMIT\_DEPRECATED

SQLITE\_OMIT\_PROGRESS\_CALLBACK

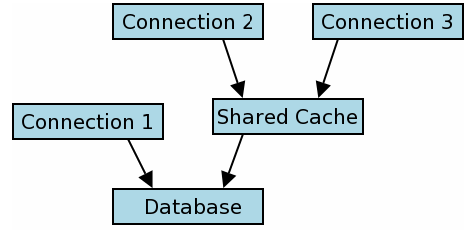
SQLITE\_OMIT\_SHARED\_CACHE

SQLITE\_USE\_ALLOCA

SQLITE\_OMIT\_AUTOINIT

**РЕЖИМ РАЗДЕЛЯЕМОГО КЭША SQLITE**

С т.з. другого процесса или потока, два и более соединения, использующих разделяемый кэш, являются одним соединением:



Нормальный протокол блокировки используется для сериализации доступа между соединением 1 и разделяемым кэшом. Внутренний протокол используется для сериализации доступа разделяемого кэша между соединениями 2 и 3.

У модели блокировки разделяемого кэша есть 3 уровня:

1. Блокировка уровня транзакции – как минимум одно соединение в одном разделяемом кэше может открыть транзакцию записи в один момент времени.

2. Блокировка уровня таблицы.

3. Блокировка уровня схемы.

А.

Б.

В. Соединение может не скомпилировать запрос, если какое-либо другое соединение держит блокировку таблицы sqlite\_schema какой-либо БД из ATTACH (включая БД из sqlite3\_open и «main».

**ИСПОЛЬЗОВАТЬ ПОСЛЕДНЮЮ ВЕРСИЮ SQLITE**

**ДИНАМИЧЕСКОЕ ВЫДЕЛЕНИЕ ПАМЯТИ В SQLITE**

Какие возможности это обеспечивает:

1. Стабильность вместо ошибок выделения памяти.

2. Нет утечек памяти.

3. Ограничение использования памяти

4. Возможность не вызывать malloc и free

5. Возможность использовать свои выделители памяти

6. Никогда не свалится в ошибку при выделении памяти и фрагментации кучи

7. Контроль со стороны приложения за количеством используемой памяти.

8. Сторонние отладчики памяти можно использовать

9. Минимальное количество вызовов выделителей памяти

10. Доступ к нижележащим функциям выделения памяти, используемые SQLite’ом: sqlite3\_malloc()…

**Тестирование**. У SQLite есть альтернативные выделители памяти: стандартный, отладочный, нативный Win32, ноль выделения памяти, экспериментальный, пользовательский, оверлей, заглушка.

**Страницы кеша.** Все соединения, работающие в одном процессе, разделяют одну страницу кеша. В такую страницу записываются данные из БД постранично. Когда SQLite необходимо получить данные из БД по запросу, он в первую очередь ищет эти данные в кеше. Если их там нет, он читает из БД и записывает данные в кеш.

Также страница кеша используется для буферизации операций записи в БД: перед изменением данных в самой БД, изменяются данные в кеше и затем уже данные из кеша копируются в саму БД.

Страница кеша содержит вхождения страницы кеша любого из трех типов. Вхождение имеет свою структуру.

**выделитель памяти lookaside**

БД делает много небольших выделений памяти при компиляции и выполнении запроса, они хранят имена таблиц, столбцов, узлов би дерева, промежуточные результаты выполнения запроса и т.д.

Начиная с 3.6.1 представлен выделитель lookaside для уменьшения нагрузки выделения памяти. С таким выделителем, каждое соединение предварительно выделяет один большой фрагмент памяти (обычно 60-120 Кб) и разделяет этот кусок на небольшие фиксированные слоты по 100-1000 байт. Это и становится пул памяти lookaside. И после этого, выделения, связанные с соответствующим соединением и которые не очень большие, удовлетворяются использованием слотов пула lookaside. Большие выделения памяти используют главный выделитель памяти.

Такой выделитель увеличивает скорость выделения и освобождения памяти за счет фиксированного размера слотов.

**Двухразмерный lookaside**

Начиная с 3.31.0 lookaside поддерживает для пула памяти: с маленькими слотами по 128 байт, и другой пул с большими, задающимися пользователем.

**Memory-Mapped I/O**

Стандартный механизм для доступа и обновления файлов БД — xRead(), xWrite(). Эти методы обычно реализованы как системные вызовы read(), write(), которые заставляют ОС копировать содержимое диска между кешем буфера ядра и пользовательским пространством.

Начиная с версии 3.7.17 появилась возможность доступа к содержимому диска напрямую, используя memory-mapped I/O (xFetch(), xUnfetch()).

Как работает MMap I/O. Для чтения, при выключенном ММАР, используется xRead(), сперва SQLite выделяет фрагмент памяти в куче размером в страницы, затем вызывает xRead(), результатом которого будет скопированные данные из БД в нововыделенную кучу памяти.

Если ММАР включен, то, перед вызовом, если понадобиться, xRead(), вызывается xFetch(), который узнает у ОС есть ли запрашиваемые данные уже в памяти процесса, если есть, то ОС возвращает указатель и xFetch() завершает работу, если нет, то возвращается null и далее вызывается xRead().

ММАР — только для чтения.

Для включения режима разделяемого кеша для каждого процесса, используется sqlite3\_enable\_shared\_cache(true/false). Каждый вызов этой функции влияет на последующие соединения, созданные через sqlite3\_open(). Каждое соединение, созданное через sqlite3\_open(), может решить принимать участие или нет в режиме разделения кеша через значение SQLITE\_OPEN\_SHAREDCACHE или SQLITE\_OPEN\_PRIVATECACHE третьего параметра.

Используя URI имена можно контролировать режим разделения кеша не процессов, созданных не только sqlite3\_open() а также через команду ATTACH: ATTACH ‘file:aux.db?cache=shared’ AS aux.

**КОНФИГУРАЦИЯ ПЛАТФОРМЫ**

\_HAVE\_SQLITE\_CONFIG\_H – можно включить config.h

HAVE\_FDATASYNC – для unix

HAVE\_GMTIME\_R – если SQLITE\_OMIT\_DATETIME\_FUNCS, то CURRENT\_TIME, CURRENT\_DATE и CURRENT\_TIMESTAMP используют gmtime\_r(), иначе — встроенные функции даты/времени.

HAVE\_ISNAN, если true, то использует функцию isnan() системной библиотеки. Если false – используется реализация SQLite.

HAVE\_LOCALTIME\_R, если true, то используется localtime\_r(), иначе localtime().

HAVE\_LOCALTIME\_S, если true, то используется localtime\_s(), иначе localtime().

HAVE\_MALLOC\_USABLE\_SIZE, если true, то SQLite пытается использовать malloc\_usable\_size() для нахождения выделенной памяти, полученной функцией malloc() стандартной библиотеки. Если false, то SQLite использует обертку вокруг malloc и realloc, которая увеличивает выделенную память на 8 байт и пишет размер выделенной памяти в начальные 8 байт, также SQLite реализует свою malloc\_usable\_size(), которая использует эти первые 8 байт для «нахождения» размера выделенной памяти.

HAVE\_STRCHRNUL, если true, то используется то используется стандартная функция strchrnul(), иначе реализация SQLite strchrnull().

HAVE\_USLEEP, для Юникс.

HAVE\_UTIME, для Юникс.

SQLITE\_BYTEORDER=1234 для LE, 4321 для BE и 0, если порядок байт определяется во время выполнения.

**ОПЦИИ ДЛЯ УСТАНОВКИ ИЗНАЧАЛЬНЫХ ЗНАЧЕНИЙ**

SQLITE\_DEFAULT\_AUTOMATIC\_INDEX, определяет изначальную установку для PRAGMA automatic index для новых соединений с БД.

SQLITE\_DEFAULT\_AUTOVACUUM, если 0,то отключен вакуум, если 1, то полный вакуум, если 2, то инкрементальный.

SQLITE\_DEFAULT\_CACHE\_SIZE — максимальный размер устанавливает для кеш-страницы для каждой прикрепленной БД.

SQLITE\_DEFAULT\_FILE\_FORMAT — также указывается в заголовке БД.

SQLITE\_DEFAULT\_FILE\_PERMISSIONS=N — для Юникс.

SQLITE\_DEFAULT\_FOREIGN\_KEYS, если 1, то включается ограничения для FK

SQLITE\_DEFAULT\_MMAP\_SIZE, устанавливает ограничение по размеру памяти, используемой для ММАР

SQLITE\_DEFAULT\_JOURNAL\_SIZE\_LIMIT, устанавливает ограничение по размеру журнала отката и ЛОЗа.

SQLITE\_DEFAULT\_LOCKING\_MODE, при 1 — EXCLUSIVE, при 0 — NORMAL.

SQLITE\_DEFAULT\_LOOKASIDE — устанавливает размер lookaside memory allocator

SQLITE\_DEFAULT\_MEMSTATUS —

SQLITE\_DEFAULT\_PCACHE\_INITSZ