**[Особенности работы с SQLite.](http://www.enterra.ru/blog/android_issues_with_sqlite/" \o "Android. Особенности работы с SQLite.)**

[**28**](http://www.enterra.ru/blog/android_issues_with_sqlite/)

Тепикин Егор

В настоящее время все большую популярность набирает OS Android. С ростом популярности платформы расширяется и круг задач, которые пользователь хочет решить здесь и сейчас. Среди задач встречаются такие, которые требуют обработки большого объема данных за приемлемое время, так, например, полнотекстовый поиск по базе. В качестве базы данных на OS Android используется SQLite. Для работы с SQLite существует пакет android.database.sqlite. Однако данный пакет содержит только набор инструментов для работы с базой. Он не является фреймворком, регламентирующим подход к реализации доступа к данным.  
  
На данный момент Google не предоставляет подробных рекомендаций по работе с базой данных. В официальной документации приводится лишь 2 простых примера, использующих SQLite (“NotePad” и “SearchableDictionary”). Поэтому программисты сами вырабатывают собственные подходы к реализации работы с базой данных, и, как результат, возникает множество различных способов – зачастую неверных.

Выработать правильный подход к реализации достаточно сложно. Основными проблемами являются слабо документированные и неочевидные особенности в классах пакета android.database.sqlite.  
Первыми признаками того, что нужно задуматься об архитектуре служат такие ошибки как:

1. database is locked – возникает при многопоточной записи в базу.
2. database is closed – может возникнуть при работе с базой из разных частей программы, например, Activity и Service.
3. corrupted database – возникает, если файл базы данных был испорчен либо пользователем, либо при неожиданном прерывании записи в базу (выключение телефона, ошибка OS, нехватка пространства, битые сектора на SD карте и т.д.)
4. низкая производительность при работе с базой данных – может возникнуть из-за внутренних блокировок, конкурирующих транзакциях, высоком уровне журналирования, отсутствии пакетной обработки.

Рассмотрим подробно как причины возникновения и возможные «неявные» проявления данных проблем, так и методы их решения.

**Проблема “database is locked” (она же многопоточность)**

У программистов часто возникают вопросы “Как лучше работать с *SQLiteOpenHelper*“. Действительно – поскольку к слою доступа к данным может обращаться практически любая часть программы (сервис, presenter, widget … ), то *SQLiteOpenHelper* должен быть доступен везде, где есть Context. Также встает вопрос, стоит ли для каждой части программы создавать свое соединение с базой, увеличится ли от этого скорость выполнения запросов. Возникают вопросы о многопоточном доступе к базе, и, конечно, о блокировках при записи.

Прежде всего, нужно отметить, что блокировки в SQLite выполнены на уровне файла. Это гарантирует блокировку для изменений из разных потоков и соединений. Причем читать базу может много потоков, а писать только один. Более подробно о блокировках в SQLite можно узнать из [документации SQLite](http://www.sqlite.org/lockingv3.html). Нас же интересует именно API, предоставляемое OS Android.

Писать в базу двумя потоками можно как из одного, так и из разных соединений с базой. Поскольку писать в базу может только одни поток, то возникает два варианта:

1. Если вы пишете из двух потоков одного соединения, то один поток будет ждать, пока закончит писать другой.
2. Если вы пишете из двух потоков разных соединений, то произойдет ошибка – ваши данные не запишутся в базу, а приложение вылетит с *SQLiteDatabaseLockedException*.

Становится очевидным, что приложение всегда должно иметь только один экземпляр *SQLiteOpenHelper*(именно открытого соединения), иначе в любой момент может возникнуть *SQLiteDatabaseLockedException*.

**Разные соединения в одном SQLiteOpenHelper**

Всем известно, что *SQLiteOpenHelper* имеет 2 метода предоставляющих доступ к базе*getReadableDatabase()* и *getWritableDatabase()*, соответственно для чтения и записи данных. Однако в большинстве случаев реальный сonnection один. Более того, это один и тот же объект:

*SQLiteOpenHelper.getReadableDatabase()==SQLiteOpenHelper.getWritableDatabase()*

Это означает, что нет никакой разницы, с помощью какого из этих методов читать данные. Но гораздо важнее другая недокументированная особенность – внутри класса *SQLiteDatabase* есть собственные блокировки – переменная mLock. Блокировки для записи на уровне объекта *SQLiteDatabase*, а поскольку на чтение и запись экземпляр *SQLiteDatabase* один, то чтение данных тоже блокируется. Особенно хорошо это видно при записи большого объема данных в транзакции.

Помимо внутренних блокировок класс *SQLiteDatabase* имеет еще одну интересную особенность: данный класс (до API 11) позволяет создавать транзакции только в режиме exclusive transaction. Из-за этого при активной работе с БД могут возникать задержки. Рассмотрим пример приложения, которое при первом старте должно в фоне скачать большой объем данных(~7000 строк содержащих BLOB) и сохранить их в базу. Если данные сохранять в транзакции, то сохранение занимает ~45сек, но при этом пользователь не может пользоваться приложением, поскольку все запросы на чтение заблокированы. Если же данные сохранять маленькими порциями, то процесс обновления данных затягивается на достаточно долгий срок (~10-15 минут), зато пользователь может все это время без каких-либо неудобств безболезненно пользоваться приложением. “Палка о двух концах” – или быстро, или удобно. Причины данной проблемы и некоторые выводы более подробно освещены в статье Kevin Galligan “Android Sqlite Locking”.

Как же бороться с данным “стандартным” поведением? В новых версиях Android, начиная с API 11, Google уже начали исправлять часть проблем связанных с работой *SQLiteDatabase* – были добавлены такие методы как:

*beginTransactionNonExclusive()* – создает транзакцию в “IMMEDIATE mode”.

*yieldIfContendedSafely()* – временно завершает транзакцию, для того, чтобы другие потоки могли выполнить свои задачи.

*isDatabaseIntegrityOk()* – проводит проверку целостности базы данных.

Более подробно описано в [документации](http://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html).

Однако для старых версий Android тоже необходим данный функционал.

**Решение**

Прежде всего, нужно отключить блокировки и разрешить чтение данных в любой ситуации.

*SQLiteDatabase.setLockingEnabled(false);* Отменяет использование внутренней блокировки запросов – на логическом уровне java класса (не связано с блокировками в терминах SQLite).

*SQLiteDatabase.execSQL(“PRAGMA read\_uncommitted = true;”);* Позволяет читать данные из кеша. По сути, изменяет уровень изоляции. Данный параметр должен устанавливаться для каждого соединения заново. Если соединений несколько, то оказывает действие только на то соединение, которое вызвало данную команду.

*SQLiteDatabase.execSQL(“PRAGMA synchronous=OFF”);* Изменить способ записи в базу – без “синхронизации”. При выключении данной опции, база данных может быть повреждена при неожиданном сбое системы, либо отключении питания. Однако согласно [документации SQLite](http://www.sqlite.org/pragma.html#pragma_synchronous) некоторые операции при выключении данной опции выполняются более чем в **50 раз** быстрее.

К сожалению не все PRAGMA поддерживаются в Android, например “PRAGMA locking\_mode = NORMAL” и “PRAGMA journal\_mode = OFF” и некоторые другие не поддерживаются. При попытке вызвать данные PRAGMA приложение вылетит с ошибкой.

В документации для метода *setLockingEnabled* сказано, что данный метод рекомендовано использовать лишь в том случае, если вы уверены, что вся работа с базой ведется из одного потока. Нам придется самим гарантировать, что в единицу времени будет выполняться только одна транзакция. Также вместо транзакций по умолчанию (exclusive transaction) нужно использовать immediate transaction. В старых версиях Android (ниже API 11) нет возможности создать immediate transaction через java обертку, однако сам SQLite данный функционал поддерживает. Для инициализации транзакции в immediate mode нужно выполнить следующий SQL запрос напрямую к базе данных,- например через метод execSQL.

*SQLiteDatabase.execSQL(“begin immediate transaction”);*

Поскольку инициализируем транзакцию мы прямым запросом, то и завершать её нужно аналогично:*SQLiteDatabase.execSQL(“commit transaction”);*

Осталось только реализовать свой TransactionManager, который будет инициировать и завершать транзакции нужного типа. Задача TransactionManager – гарантировать, что все запросы на изменение (insert, update, delete, DDL запросы) происходят из одного потока.

**Проблема “database is closed”**

При работе с базой из одной Activity через *SQLiteOpenHelper*, очевидно, что открывать базу нужно с открытием Activity, а закрывать при закрытии Activity. Но если с базой работает одновременно несколько Activity, несколько Service и часть данных расшаривает ContentProvider, то возникает вопрос: “когда следует открывать и закрывать соединение с базой?”. Если открывать и закрывать соединение после каждого запроса,- то скорость обращения к базе упадет в разы, а если открывать при старте приложения и закрывать при выходе,- то непонятно, когда мы выходим из приложения (а если сервис еще работает, или провайдер еще используется – остается только метод *Application.onTerminate()*). Но не один из этих методов не является верным. Соединение с базой может закрыться автоматически при следующих условиях:

Если несколько Activity независимо друг от друга будут открывать новые соединения, то может возникнуть ошибка, описанная в предыдущем пункте “database is locked”.

Если открывать соединение с базой при старте приложения и закрывать при *Application.onTerminate()*, то соединение с базой может закрыться само при очередном вызове *Cursor.getCount()* или *Cursor.onMove()*. Если детально просмотреть исходный код соответствующих классов, то видно, что при определенной комбинации условий в конечном итоге будет вызван метод *SQLiteDatabase.onAllReferencesReleased()*, который вызывает нативный метод *dbclose()*. Более детально данная проблема освещена [здесь](http://stackoverflow.com/questions/6535908/android-sqlite-sqliteopenhelper-error-illegalstateexception-db-already-clo), последовательность вызовов и необходимые условия описаны [тут](http://darutk-oboegaki.blogspot.com/2011/03/sqlitedatabase-is-closed-automatically.html).

Возможно, это одна из причин, по которой “ManagedCursor” объявили “Deprecated”.

Данная проблема широко известна и для ее решения предложено множество способов.

**Вариант 1**

При каждом обращении к базе проверять,- закрыта база или нет, и если закрыта, то переоткрывать её заново.

public synchronized SQLiteDatabase getReadableDatabase() {          
        SQLiteDatabase db;  
        try {  
            db = super.getReadableDatabase();  
        }   
        catch (SQLiteException e) {  
            Log.d(Constants.DEBUG\_TAG, e.getMessage());              
            db = reopenDatabase(dbFile.getAbsolutePath(), null);  
        }  
        return db;  
    }

У данного метода есть очевидный недостаток – если мы обратились к базе, а затем сохранили ссылку на уже открытый экземпляр, и используем полученный экземпляр, не вызывая*SQLiteDatabase.getReadableDatabase()*, то данный метод не сработает.

**Вариант 2**

Принудительно добавить фиктивную ссылку на базу и держать её пока база используется*SQLiteClosable.acquireReference()*;

Но в этом случае нужно самостоятельно закрывать базу предварительно очистив все ссылки созданные вручную. Но все равно количество ссылок может стать равным нулю, поэтому нужно непрерывно следить за количеством ссылок и пополнять их при необходимости. Но этот метод является не самым удачным.

**Вариант 3**

Закрывать и открывать базу после каждого запроса. Данный метод не надежен, поскольку возможно создание двух независимых соединений на запись и произойдет ошибка. Если данный подход использовать только для чтения, то ошибки не возникнет, но значительно замедлит работу приложения (особенно если обращений к базе много).

**Вариант 4**

Использовать ContentProvider для доступа к базе. Причем желательно использовать именно один провайдер – это легко реализовать, поскольку ему можно добавить неограниченное количество Uri. Суть заключается в том, что *ContentProvider* сам следит за состоянием базы данных. А вопрос о том, когда базу пара закрывать ложится на OS – она сама удалит старые провайдеры из памяти и вернет их при первой необходимости.

Про работу с *ContentProvider* подробное описание есть на [официальном сайте](http://developer.android.com/guide/topics/providers/content-provider-creating.html).

**Проблема “corrupted database”**

На Android телефонах очень мало места отводится под приложения и это место нужно беречь, иначе пользователь пожертвует вашим приложением в пользу очередной игрушки. Почти все приложения используют базу для хранения данных, и если база слишком большая, то её очень желательно хранить на SD карте. Старые версии Android (2.2 и ниже) не позволяют создавать базу на SD карте через стандартные средства *SQLiteOpenHelper*, но это можно обойти, если использовать *AndroidConnectionSource* от ORMLite.

Нужно помнить – все, что видно пользователю может быть удалено. Пользователь может удалить или иным образом испортить файл базы данных, может вынуть SD карту из телефона, и многое другое. Но база может быть испорчена не только пользователем. Телефон – устройство с ненадежным питанием – часть данных может не записаться (особенно актуально – если отключено журналирование), база может быть повреждена при скачивании или при использовании предустановленной базы и т.д. Более подробно, о причинах порчи базы можно узнать из статьи [“How To Corrupt An SQLite Database File”](http://www.sqlite.org/howtocorrupt.html).

Если разработчик никак не реализовал алгоритм восстановления базы, то Android сам создаст базу заново. Но бывают случаи, когда базу можно восстановить. Самый простой способ – запросить данные из всех доступных таблиц и вставить в новую базу. Но чаще достаточно выполнить команду [“VACUUM”](http://sqlite.org/lang_vacuum.html) – данный метод пересоздает базу и восстанавливает максимум данных.

Очень часто есть необходимость создать приложение с предустановленными данными. Для этого можно собрать готовую базу и положить в папку raw, а в момент установки приложения база будет скопирована на устройство. Файл с базой лучше расположить именно в папке raw. Папка assets кажется более удобной, поскольку подвергается сжатию, но из данной папки невозможно получить данные объемом более 1 мб ([[см. здесь](http://stackoverflow.com/questions/2860157/load-files-bigger-than-1m-from-assets-folder)), и поэтому приходится разбивать базу на отдельные файлы по 1мб – что весьма неудобно. Важно, что базу всегда нужно собирать именно на эмуляторе самой младшей из поддерживаемых версий. Поскольку если собрать предустановленную базу на Android 2.3, то на Android 2.2 возникнет ошибка “corrupted database”, хотя на устройствах 2.3 и выше база будет работать корректно.

**Оптимизация запросов**

Скорость выполнения запросов складывается из множества факторов, но наиболее важными из них являются оптимизация самого запроса и структура базы данных. Для оптимизации запросов есть множество стандартных методов, которые достаточно легко найти в интернете, поэтому перечислим особенности оптимизации именно для SQLite. Для краткости оформим их в виде тезисов.  
Не нужно писать запросы, которые возвращают более 1000 строк или данные объемом более 1мб. Всегда используйте оператор limit. Если запрос возвращает более 1000 строк, то будет выдано предупреждение в лог, либо произойдет падение приложения – это зависит от количества свободной памяти и самого устройства. Если нужно отображать длинный список есть два решения:

a) нужно запрашивать список частями, а затем соединять при помощи *android.database.CursorJoiner*.

b) Реализовывать авто дополняемый список на интерфейсе (список с догрузкой).

Один запрос выполняется значительно быстрее, чем 2 отдельных. Лучше использовать join, но выполнить 1 запрос. Очень важен порядок ограничений на join – чтобы не получилось декартово произведение, из которого потом выбираются строки по оператору where.

Если нужно что-то изменить в базе – делайте это в транзакции. Это не только гарантирует целостность данных, но и значительно ускорит выполнение задачи. Дело в том, что при любом изменении в базе данных рядом с файлом базы создается файл изменений. Если вы делает 100 insert, то 100 раз создастся и удалится файл изменений, а если 100 insert находятся в транзакции, то файл создастся всего 1 раз.

Если нужно создать таблицу из уже существующих данных, то используйте INSERT AS SELECT (не выполняйте отдельные INSERT) – это значительно ускорит время выполнения.

Если вы получили из базы много данных за раз, и этот “большой” запрос повторяется не часто, то очищайте ненужную память *SQLiteDatabase.releaseMemory()*.

В операторе where нужно сначала писать более простые условия.

*SELECT \* FROM tablename WHERE col1 LIKE ‘%string%’ AND col2 = 123456*

работает в 3-4 раза медленнее чем

*SELECT \* FROM tablename WHERE col2 = 123456 AND col1 LIKE ‘%string%’*

Правильное индексирование таблиц ускоряет выполнение запросов в 5-7 раз. Индексировать нужно в первую очередь те поля, по которым идет join, затем те по которым идет поиск. Причем лучше указывать направление работы индекса – например

*CREATE INDEX index\_name ON table\_name (column\_name ASC).*

Для больших таблиц с поиском используйте FTS3 это значительно ускоряет текстовый поиск по таблице. Вместо LIKE используйте MATCH, но помните что по умолчанию MATCH работает как поиск слова целиком, а не подстроки. [Описание FTS3](http://www.sqlite.org/fts3.html).

**Заключение**

В данной статье перечислены основные проблемы при работе с SQLite в Android. К сожалению, в API еще существует очень много пробелов, для решения ряда вопросов отсутствует необходимая документация, а также в процессе работы периодически выявляются ошибки в самой системе. Но радует тот факт, что с каждой новой версией Android API становится все гибче и полнее, ошибки исправляются, а документация расширяется.

Хотите использовать наш опыт в [разработке приложений на Android](http://www.enterra.ru/android)? Начните с [оценки Вашего проекта](http://www.enterra.ru/other-pages/contact-form-submit-your-project-request.html)прямо сейчас!