Лебедев М.М., Раскатова М.В. Оптимизация скорости работы Android устройств с локальной базой данных // Международный журнал информационных технологий и энергоэффективности. – 2020. – Т. 5 № 2(16) с. – 3-21

Международный журнал информационных технологий и энергоэффективности

Сайт журнала:
http://www.openaccessscience.ru/index.php/ijcse/

УДК 004.4

ОПТИМИЗАЦИЯ СКОРОСТИ РАБОТЫ ANDROID УСТРОЙСТВ С ЛОКАЛЬНОЙ БАЗОЙ ДАННЫХ

[1]Лебедев М.М., [2]Раскатова М.В.
ФГБОУ ВО «НИУ «МЭИ», Москва, Россия (111250, г. Москва, ул. Красноказарменная, д.14), e-mail: 1) leb.dev96@gmail.com, 2) marina@raskatova.ru

В статье описываются варианты работы устройства Android с локальными базами данных. Рассматриваются различные режимы использования и функции для работы с данными. Проводятся тесты и измерения времени выполнения операций в различных условиях. Составляются рекомендации по локальному хранению данных и написанию программного кода для работы с SQLite на платформе Android.

Ключевые слова: Android, SQLite, SQLiteOpenHelper.

OPTMIZATION OF OPERATION SPEED ANDROID DEVICES WITH LOCAL DATABASE

[1]Lebedev M.M., [2]Raskatova M.V.

NRU «MPEI», Moscow, Russia (111250, Moscow, street Krasnokazarmennaya, 14), e-mail: 1) leb.dev96@gmail.com, 2) marina@raskatova.ru

This article describes different variants of working with local database on Android platform. Various usage modes and functions for working with data are considered. Tests are conducted and measurements are taken of operations' execution timings in different conditions. Also, recommendations on local data storing and code development for working with SQLite on Android are given.

Keywords: Android, SQLite, SQLiteOpenHelper

Для локального хранения данных на устройствах Android всегда используют базу данных SQLite. Для совершения операций над данными и работой с базой обычно в качестве решения для Java кода используют библиотеку SQLiteOpenHelper [1] или же Object Relational Mapping библиотеки. Однако, последние, как правило, имеют меньшую производительность, сильно выигрывая в читабельности и наглядности кода [2]. В данной статье мы рассмотрим случай, когда приложению необходимо работать с локальной базой данных на устройстве Android и обрабатывать значительные объёмы данных. Поэтому будет рассмотрена возможность оптимизации скорости работы с базой данных SQLite в сочетании с библиотекой SQLiteOpenHelper.

SQLite базы данных используют динамическое типизирование. Каждая колонка имеет только указание, данные какого формата следует в ней хранить, но не вводит жёсткое

ограничение. Хранимые в SQLite базах данные относят к одному из пяти следующих типов: NULL, INTEGER, REAL, TEXT, BLOB. Все прочие типы приводятся базой SQLite к этим пяти [3]. Рассмотрим, как тип хранимых данных может повлиять на скорость работы с базой данных.

Создать колонку, имеющую «рекомендацию» типа NULL невозможно и не имеет смысла, поэтому данный случай не следует рассматривать.

INTEGER в зависимости от величины значения может сжиматься для экономии места на диске, но в процессе обработки механизмами SQLite и INTEGER, и REAL представляются, как 8-байтное значение. Поэтому операции над обоими типами не отличаются по времени выполнения.

Согласно документации SQLite, длина значений, хранимых в типах TEXT и BLOB - не ограничена. Это означает, что объём данных, над которыми проводится операция может каждый раз быть разным, и что теоретически это может влиять на время выполнения операции. Длина данных типа TEXT будет зависеть от используемой кодировки, количества символов и длин кодов этих символов. К примеру, при использовании UTF-8 (используется по умолчанию), каждый символ, входящий в диапазон таблицы ASCII, будет иметь вес в 1 байт. То есть, операция над группой из 8 символов английского алфавита даёт равную нагрузку в сравнении с операцией над типами INTEGER и REAL. Зависимость скорости выполнения операции от размера вставляемых данных также следует проверить.

Для работы с базой через библиотеку SQLiteOpenHelper следует создать класс-наследник, в котором будут переопределяться функции. Настройки для соединения с базой задаются в конструкторе класса. В простейшем случае, для задания настроек по умолчанию в конструктор следует передать контекст приложения, имя открываемой базы, фабрику курсоров (можно передать null) и текущую версию, с которой следует открыть базу. Разработаем необходимый класс и назовём его «DatabaseHelper». Конструктор с настройками по умолчанию будет выглядеть следующим образом:

```
private static final int DATABASE_VERSION = 1;
private static final String DATABASE_NAME = "appDB.db";
private static final String TAG = "UserLog.Database";

DatabaseHelper(Context context) {
    super(context, DATABASE_NAME, null, DATABASE_VERSION);
    Log.d(TAG, "DatabaseHelper: Constructor called");
}
```

Коллбэки создания базы и обновления её версии оставим пустыми, так как для каждого теста мы будем создавать таблицы заново, и их структура может отличаться от теста к тесту.

```
@Override
public void onCreate(SQLiteDatabase db) {
    Log.d(TAG, "onCreate: Creating app database");
}

@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    Log.d(TAG, "onUpgrade: Upgrading app database from version " + oldVersion +
```

```
" to " + newVersion);
}
```

Тестирование будем проводить с помощью инструмента юнит тестирования для платформы андроид AndroidJUnit4.

При работе с данными могут выполняться операции выборки (SELECT), обновления данных (UPDATE), записи новых данных (INSERT), удаления данных (DELETE). Для начала рассмотрим операции вставки новых данных.

Алгоритм теста следует сделать следующим:

- очистка БД от существующих таблиц (если таковые имеются);
- создание таблицы с указанными характеристиками;
- операция над данными с замером времени;
- вывод результата.

Для того, чтобы полученные временные значения были более достоверными, тесты необходимо проводить сериями. Для чего создание таблицы и операцию над данными следует поместить в тело цикла, и выполнить некоторое количество раз, после чего вычислить среднее арифметическое полученных данных. В каждом рассматриваемом в дальнейшем случае будем проводить по 100 тестов с одинаковыми условиями, а затем вычислять и записывать среднее арифметическое полученных значений.

1 Операции вставки данных

1.1 Вставка с помощью ContentValues

Проведём тесты времени выполнения операций с настройками по умолчанию. Проведём вставку значений через функцию класса SQLiteDatabase – insert(String table, String nullColumnHack, ContentValues values). Для вставки значений через эту функцию ей необходимо подавать имя таблицы и объект ContentValues, в который перед каждым вызовом insert необходимо разместить соответствие: «имя колонки» ↔ «значение». Напишем в классе DatabaseHelper функцию, которую будем вызывать из теста:

```
long insertViaContentValues(Integer value, int count) {
    ContentValues contentValues = new ContentValues();
    SQLiteDatabase sqLiteDatabase = getWritableDatabase();
    long time = getCurrentTime();
    for (int i = 0; i < count; i++) {
        contentValues.put(KEY_COLUMN_1, value);
        sqLiteDatabase.insert(TABLE_NAME, null, contentValues);
    }
    time = getDifference(time);
    Log.d(TAG, "insertViaContentValues: Done inserting " + count + " values in "
+ time + " ms");
    return time;
}
```

Кратко запишем условия выполнения (таблица 1) и результаты теста (таблица 2).

Таблица 1 – Условия выполнения теста

Настройки SQLiteOpenHelper	**Устройство**	**Структура БД**	**Операция**
По умолчанию	Pixel 2 API 24 (AVD)	Таблица: Колонка Integer	Вставка значений через ContentValues

Таблица 2 – Результаты теста

Количество операций подряд	**Среднее время на выполнение одной операции (мс)**
1	4,73000
10	4,50500
100	4,55500
1000	4,50835

Как можно увидеть из результатов теста – количество времени, затрачиваемого на операцию вставки при данных условиях, практически не изменяется с ростом количества идущих подряд операций. И на каждую операцию в среднем требуется около 4,6 мс.

1.2 Влияние количества колонок на операцию вставки данных

Узнаем, как количество колонок вставляемых данных влияет на скорость выполнения операций. Проведём аналогичные тесты, создав две одинаковых колонки типа Integer. Ожидаем увидеть, увеличение времени выполнения каждой операции.

Чтобы установить зависимость времени выполнения операции вставки от количества колонок, видоизменим функцию вставки для работы с переменным количеством колонок и проведём тесты с различным количеством.

```
long insertViaContentValues(Integer value, int count, int columnCount) {
    ContentValues contentValues = new ContentValues();
    SQLiteDatabase sqLiteDatabase = getWritableDatabase();
    long time = getCurrentTime();
    for (int i = 0; i < count; i++) {
        for (int j = 1; j <= columnCount; j++) {
            contentValues.put(KEY_COLUMN + j, value);
        }
        sqLiteDatabase.insert(TABLE_NAME, null, contentValues);
    }
    time = getDifference(time);
    Log.d(TAG, "insertViaContentValues: Done inserting " + count + " values in "
+ time + " ms");
    return time;
}
```

Так как SQLite базы данных имеют ограничение на количество колонок, а также параметров в выражении = 999, то вместо варианта с 1000 колонок, мы рассмотрим 999 в качестве максимума. Условия теста и результаты приведены в таблицах 3 и 4.

Таблица 3 – Условия выполнения теста

Настройки SQLiteOpenHelper	**Устройство**	**Структура БД**	**Операция**
По умолчанию	Pixel 2 API 24 (AVD)	Таблица: Колонка Integer ×X (переменное количество)	Вставка значений через ContentValues

Таблица 4 – Результаты теста

Количество операций подряд	**Среднее время на выполнение одной операции (мс)**						
	Количество колонок:	**1**	**10**	**100**	**200**	**500**	**999**
1	4,80000		4,80000	5,42000	7,13000	14,11000	36,20000
10	4,81100		4,47200	4,99200	5,74500	7,60300	9,30300
100	4,64560		4,54880	5,01020	5,74500	7,77990	9,20260
1000	4,62806		4,50898	4,92928	5,63511	7,78070	9,26118

Глядя на полученные результаты, можно отметить, что время выполнения вставки в одну колонку не сильно меньше времени вставки в 100 колонок. С учётом того, что в мобильных приложениях, как правило, таблицы имеют небольшое количество колонок, которое не часто превышает 100, можно сказать, что увеличение числа колонок обычно почти не сказывается на производительности операций вставки данных.

Также отметим, что с ростом количества колонок уменьшается время выполнения операции, если некоторое их количество идёт подряд, вероятно, из-за внутренних механизмов оптимизации SQLite.

1.3 Влияние типов данных и длины данных

Проведём тест со строковыми данными, изменяя их длину. Ожидаем, что вставка строки длиной 8 байт эквивалентна по нагрузке вставке одного целочисленного значения, а потому время выполнения операции должно быть одинаковым, однако с ростом длины строки нагрузка должна возрастать.

Чтобы сравнить получаемые результаты с результатами теста влияния количества колонок, протестируем вставку строк объёмами эквивалентными 1, 10, 100… значениям типа Integer, как в упомянутом тесте. Условия теста и результаты приведены в таблицах 5 и 6.

Таблица 5 – Условия выполнения теста

Настройки SQLiteOpenHelper	**Устройство**	**Структура БД**	**Операция**
По умолчанию	Pixel 2 API 24 (AVD)	Таблица: Колонка String	Вставка значений через ContentValues

Таблица 6 – Результаты теста

Количество операций подряд	**Среднее время на выполнение одной операции (мс)**						
	Длина строки в байтах:	**1*8**	**10*8**	**100*8**	**200*8**	**500*8**	**999*8**
1	4,89000		4,94000	5,13000	5,22000	5,11000	7,34000
10	4,52700		4,75800	5,08000	5,50000	6,11100	6,80700
100	4,53240		4,79780	5,19250	5,59230	6,29870	6,74740
1000	4,52082		4,67494	5,19367	5,52837	6,25584	6,63281

Получаемые результаты говорят нам о том, что увеличение объёма вставляемых за одну операцию данных, как и ожидалось, приводит к увеличению времени выполнения операции. Однако, куда выгоднее по времени вставить строку эквивалентную по объёму 100 целочисленным значениям, чем разместить в 100 колонок целочисленные значения. Наблюдаемое явление заставляет задуматься об использовании сериализации данных, например, в JSON строку. Если количество колонок действительно велико, это может положительно сказаться на производительности.

1.4 Объединение в одну транзакцию

Проведём тестирование времени выполнения операций вставки, используя механизм транзакций SQLite. Перепишем функцию так, чтобы каждый цикл операций был обёрнут в цельную транзакцию, и посмотрим на результаты (таблицы 7, 8).

```
long insertViaContentValuesWithTransaction(Integer value, int count, int
columnCount) {
    ContentValues contentValues = new ContentValues();
    SQLiteDatabase sqLiteDatabase = getWritableDatabase();
    long time = getCurrentTime();
    sqLiteDatabase.beginTransaction();
    try {
        for (int i = 0; i < count; i++) {
            for (int j = 1; j <= columnCount; j++) {
                contentValues.put(KEY_COLUMN + j, value);
            }
            sqLiteDatabase.insert(TABLE_NAME, null, contentValues);
        }
        sqLiteDatabase.setTransactionSuccessful();
```

```
    } finally {
        sqLiteDatabase.endTransaction();
    }
    time = getDifference(time);
    Log.d(TAG, "insertViaContentValuesWithTransaction: Done inserting " + count
+ " values in " + time + " ms");
    return time;
}
```

Таблица 7 – Условия выполнения теста

Настройки SQLiteOpenHelper	Устройство	Структура БД	Операция
По умолчанию	Pixel 2 API 24 (AVD)	Таблица: Колонка Integer	Вставка значений через ContentValues в единой транзакции

Таблица 8 – Результаты теста

Количество операций подряд	Среднее время на выполнение одной операции (мс)
1	5,210000
10	0,697000
100	0,199300
1000	0,147860
10000	0,141041

Время выполнения операций резко сократилось. Особенно это заметно по результатам вставки большого количества строк, причём чем больше вставок, тем быстрее выполняется каждая из них. Становится очевидно, что вставку нескольких строк по возможности всегда лучше объединять в единую транзакцию, так как, например, при проведении тестов вставка 100 строк без транзакции заняла примерно 450 мс, а в единой транзакции – всего 20 мс.

1.5 ExecSQL

Протестируем иной способ вставки данных – через выполнение функции ExecSQL. Данный метод компилирует и выполняет любой запрос к БД, кроме SELECT.

```
long insertViaExecSQL(String value, int count) {
    SQLiteDatabase sqLiteDatabase = getWritableDatabase();
    long time = getCurrentTime();
    String query = "Insert into " + TABLE_NAME + " (" + KEY_COLUMN_1 + ") values
(?)";
    for (int i = 0; i < count; i++) {
        sqLiteDatabase.execSQL(query, new String[] {value});
    }
    time = getDifference(time);
    Log.d(TAG, "insertViaExecSQL: Done inserting " + count + " values in " +
time + " ms");
```

```
    return time;
}
```

Условия теста и результаты приведены в таблицах 9 и 10.

Таблица 9 – Условия выполнения теста

Настройки SQLiteOpenHelper	**Устройство**	**Структура БД**	**Операция**
По умолчанию	Pixel 2 API 24 (AVD)	Таблица: Колонка Integer	Вставка значений через ExecSQL

Таблица 10 – Результаты теста

Количество операций подряд	**Среднее время на выполнение одной операции (мс)**
1	4,91000
10	4,60600
100	4,61530
1000	4,63406

Очевидно, что данный способ оказался малоэффективен, если не менее эффективен, чем вставка через ContentValues.

1.6 Prepared Statements

Протестируем вставку данных, используя механизм прекомпилированных запросов. Такие запросы компилируются единожды, а в последствии могут быть переиспользованы базой некоторое количество раз подряд. Их рекомендуется использовать, если вы собираетесь выполнить некоторую повторяющуюся операцию, например, вставку нескольких строк. Прекомпилированные запросы являются предпочтительными при использовании повторяющихся операций, к тому же их использование сокращает количество используемой памяти [4].

```
long insertViaPrecompiledStatement(Integer value, int count) {
    SQLiteDatabase sqLiteDatabase = getWritableDatabase();
    long time = getCurrentTime();
    String query = "Insert into " + TABLE_NAME + " (" + KEY_COLUMN_1 + ") values
(?)";
    SQLiteStatement sqLiteStatement = sqLiteDatabase.compileStatement(query);
    for (int i = 0; i < count; i++) {
        sqLiteStatement.bindLong(1, value);
        sqLiteStatement.executeInsert();
    }
    time = getDifference(time);
    Log.d(TAG, "insertViaPrecompiledStatement: Done inserting " + count + "
values in " + time + " ms");
    return time;
}
```

Условия теста и результаты приведены в таблицах 11 и 12.

Таблица 11 – Условия выполнения теста

Настройки SQLiteOpenHelper	Устройство	Структура БД	Операция
По умолчанию	Pixel 2 API 24 (AVD)	Таблица: Колонка Integer	Вставка значений через PrecompiledStatement

Таблица 12 – Результаты теста

Количество операций подряд	Среднее время на выполнение одной операции (мс)
1	4,89000
10	4,43400
100	4,45830
1000	4,45320

В результатах теста заметно небольшое сокращение времени выполнения, по сравнению с обычной вставкой через ContentValues.

1.7 Журналирование

Попробуем изменить настройки открытия БД, обернув конструктор DatabaseHelper в фабричный статический метод.

```
@RequiresApi(api = Build.VERSION_CODES.P)
static DatabaseHelper databaseHelperFactoryMethod(Context context) {
    SQLiteDatabase.OpenParams.Builder builder = new
SQLiteDatabase.OpenParams.Builder();
    builder.setJournalMode("TRUNCATE");
    return new DatabaseHelper(context, builder.build());
}

@RequiresApi(api = Build.VERSION_CODES.P)
DatabaseHelper(Context context, SQLiteDatabase.OpenParams openParams) {
    super(context, DATABASE_NAME, DATABASE_VERSION, openParams);
    Log.d(TAG, "DatabaseHelper: Constructor called");
}
```

К сожалению, вызов такого конструктора требует минимального уровня API = 28, что заставит нас поднять минимальный уровень API для использования приложения, и, возможно, ощутимо сократить круг мобильных устройств, которым оно будет доступно. Поэтому рассматривать возможность изменения параметров открытия базы следует исключительно в случаях, когда известно, что приложение будет использоваться более современными моделями устройств.

Изменим параметр журналирования. Журналирование БД может работать в одном из нескольких доступных режимов.

- DELETE – Стандартное поведение, при котором журнал отката удаляется по завершению транзакции.
- TRUNCATE – По завершению транзакции файл журнала обрезается до нулевой длины.
- PERSIST – Файл журнала остаётся в системе, но его заголовок заполняется нулями.
- MEMORY – Заставляет поместить файл отката в оперативную память, что повышает скорость, но может привести к повреждению БД.
- WAL – Использует Write Ahead Logging. Полезно при использовании БД одновременно несколькими процессами.
- OFF – Отключает журнал отката, из-за чего функция ROLLBACK будет вести себя непредсказуемо, не рекомендуется к использованию.

Не следует использовать режимы MEMORY и OFF в реальных условиях эксплуатации приложения, так как это ставит под угрозу целостность данных в базе, а потому их рассмотрение также можно опустить. Условия теста и результаты приведены в таблицах 13 и 14.

Таблица 13 – Условия выполнения теста

Настройки SQLiteOpenHelper	**Устройство**	**Структура БД**	**Операция**
Различные режимы журналирования	Pixel 2 API 28 (AVD)	Таблица: Колонка Integer	Вставка значений через ContentValues

Таблица 14 – Результаты теста

Количество операций подряд	**Среднее время на выполнение одной операции (мс)**				
	Используемый режим:	**DELETE**	**TRUNCATE**	**PERSIST**	**WAL**
1	6,24000		8,56000	7,45000	1,62000
10	5,68200		8,07700	6,93200	1,22100
100	5,61160		7,99180	6,86410	1,28850
1000	5,51070		7,82960	6,81806	1,27050

Результаты тестов показывают, что наиболее эффективно использовать режим WAL для осуществления вставки записей. Аналогичные результаты были получены при выполнении тестов вставки через ContentValues с использованием транзакций. Условия теста и результаты приведены в таблицах 15 и 16.

Таблица 15 – Результаты теста вставок в единой транзакции

Количество операций подряд	**Среднее время на выполнение одной операции (мс)**				
	Используемый режим:	**DELETE**	**TRUNCATE**	**PERSIST**	**WAL**
1	6,88000		8,99000	7,11000	1,97000
10	0,74100		0,95700	0,79200	0,27100
100	0,17840		0,20000	0,18660	0,13200
1000	0,12178		0,12275	0,12114	0,11309

1.8 Write Ahead Logging

При тестировании различных режимов журналирования было выявлено сокращение времени выполнения операций вставки данных в несколько раз при использовании режима WAL (Write Ahead Logging). Открытие базы с нестандартными опциями доступно только на более новых версиях платформы Android, что доставляет неудобства при разработке приложений для широкого круга пользователей. Однако режим WAL можно включить или отключить в ходе выполнения кода программы в любой момент времени. Согласно официальной документации SQLite [5], данный режим имеет следующие наиболее значимые особенности:

- WAL значительно быстрее в большинстве сценариев работы БД
- WAL повышает эффективность параллельного использования БД, потому как читатели не блокируют писателей, а также верно и обратное
- Величина страницы не может быть изменена в этом режиме
- WAL может быть на 1-2% менее эффективен при частых чтениях и редких записях

Проведём тестирование вставки через ContentValues без транзакции и с транзакцией. Для отслеживания влияния режима попробуем провести тест без него (в режиме по умолчанию), затем включим его в начале операции и выключим в конце, а затем установим его до выполнения тестов. Условия теста и результаты приведены в таблицах 16 - 18.

Таблица 16 – Условия выполнения теста

Настройки SQLiteOpenHelper	**Устройство**	**Структура БД**	**Операция**
По умолчанию, включение WAL «на ходу», постоянно включённый WAL	Pixel 2 API 24 (AVD)	Таблица: Колонка Integer	Вставка значений через ContentValues с транзакцией и без

Таблица 17 – Результаты теста без транзакции

Количество операций подряд	**Среднее время на выполнение одной операции (мс)**			
	Используемый режим:	**По умолчанию**	**WAL на время выполнения операции**	**Постоянно включённый WAL**
1	4,98000		34,0300	1,31000
10	4,62500		4,31000	0,99100
100	4,60180		2,60100	0,98470
1000	4,52814		1,16059	0,95566

Таблица 18 – Результаты теста в единой транзакции

Количество операций подряд	**Среднее время на выполнение одной операции (мс)**			
	Используемый режим:	**По умолчанию**	**WAL на время выполнения операции**	**Постоянно включённый WAL**
1	5,10000		34,3600	1,66000
10	0,64900		2,54100	0,25300
100	0,19610		0,36960	0,15340
1000	0,14653		0,16629	0,13960

На основе произведённых тестов, можно заметить, что включение и отключение данного режима требует некоторого времени, поэтому его не стоит включать или отключать во время выполнения операций. Очевидно также, что использование режима на постоянной основе действительно даёт уменьшение времени выполнения операций вставки. При написании кода следует учесть, какие сценарии работы с базой данных будут использоваться наиболее часто. Так как, согласно документации, время чтения может быть увеличено на несколько процентов, а, согласно проведённым тестам, время выполнения операций вставки может сократиться до нескольких раз, режим WAL можно смело рекомендовать к включению в приложениях, чаще (или в равной степени часто) использующих операции вставки, чем операции получения данных.

2 Операции чтения

2.1 Чтение с помощью Query

Рассмотрим операции чтения из БД. Функция Query библиотеки SQLite – одна из функций, дающих возможность прочитать данные из базы. По вызову функции из БД выбираются данные, подходящие по параметрам поиска, а в код программы передаётся курсор для навигации по результатам. Перед выполнением операций чтения необходимо создать таблицу и заполнить её данными. Проделаем это перед выполнением тестов: создадим таблицу с одной целочисленной колонкой и вставим туда 1000 записей, после чего будем выполнять уже сам тест чтения. Условия теста и результаты приведены в таблицах 19 и 20.

Таблица 19 – Условия выполнения теста

Настройки SQLiteOpenHelper	**Устройство**	**Структура БД**	**Операция**
По умолчанию	Pixel 2 API 24 (AVD)	Таблица: Колонка Integer 1000 записей	Чтение значений с помощью функции Query

Таблица 20 – Результаты теста без транзакции

Количество операций подряд	**Среднее время на выполнение одной операции (мс)**
1	1,03000
10	0,10300
100	0,01220
1000	0,00288

По результатам теста можно заметить, что чем больше строк считывается подряд, тем меньше времени занимает каждая операция. Это объясняется тем, что на выполнение получения результата тратится всегда несколько больше времени, чем занимает непосредственно каждая операция считывания данных курсором.

2.2 Влияние количества записей в таблице

Увеличим число записей в таблице и сравним результаты, при этом считывать мы будем также только некоторое количество записей сверху. Условия теста и результаты приведены в таблицах 21 и 22.

Таблица 21 – Условия выполнения теста

Настройки SQLiteOpenHelper	**Устройство**	**Структура БД**	**Операция**
По умолчанию	Pixel 2 API 24 (AVD)	Таблица: Колонка Integer Переменное количество записей	Чтение значений с помощью функции Query

Таблица 22 – Результаты теста

Количество операций подряд	**Среднее время на выполнение одной операции (мс)**				
	Количество записей в таблице:	**1000**	**10000**	**100000**	**1000000**
1	1,10000		14,7200	816,370	1666,69
10	0,10800		1,46500	81,7810	166,476
100	0,01280		0,14690	8,17270	16,6423
1000	0,00284		0,01628	0,81900	1,66530
10000	0,00029		0,00358	0,08368	0,16830

По результатам теста можно отметить, что среднее время выполнения одной операции уменьшается примерно во столько же раз, во сколько и увеличивается количество операций. Это означает, что увеличение количества считываемых данных почти не влияет на время выполнения одной операции чтения, а вот увеличение записей в таблице и, как следствие, количества результатов поиска резко негативно сказывается на времени чтения. То есть не столь важно, читаете ли вы одну запись из огромной таблицы или 100, чтение займёт примерно одинаковое количество времени.

2.3 Влияние количества колонок

Установим количество записей – 10000. Увеличим количество колонок и видоизменим функцию получения записей, чтобы она могла работать с переменным количеством колонок. Условия теста и результаты приведены в таблицах 23 и 24.

Таблица 23 – Условия выполнения теста

Настройки SQLiteOpenHelper	**Устройство**	**Структура БД**	**Операция**
По умолчанию	Pixel 2 API 24 (AVD)	Таблица: Переменное количество колонок Integer 10000 записей	Чтение значений с помощью функции Query

Таблица 24 – Результаты теста

Количество операций подряд	**Среднее время на выполнение одной операции (мс)**					
	Количество колонок:	**1**	**10**	**100**	**200**	**500**
1	14,2500		70,6900	144,520	255,560	599,32001
10	1,41100		7,02800	14,5220	25,2710	59,97800
100	0,14120		0,71460	1,53440	2,68870	6,47050
1000	0,01578		0,08110	0,24959	0,52306	1,39602
10000	0,00347		0,02124	0,21253	0,49730	2,01422

2.4 Влияние объёма данных

Проведём тест аналогичный тестам записи. Будем считывать строки различной длины, постепенно увеличивая объём, соблюдая его эквивалентность объёму некоторого количества целочисленных значений. Будем сравнивать эффективность выполнения операций над длинными строками с операциями над большим количеством целочисленных значений. Условия теста и результаты приведены в таблицах 25 и 26.

Таблица 25 – Условия выполнения теста

Настройки SQLiteOpenHelper	**Устройство**	**Структура БД**	**Операция**
По умолчанию	Pixel 2 API 24 (AVD)	Таблица: Колонка String 10000 записей	Чтение значений с помощью функции Query

Таблица 26 – Результаты теста

Количество операций подряд	**Среднее время на выполнение одной операции (мс)**					
	Длина строки в байтах:	**1*8**	**10*8**	**100*8**	**200*8**	**500*8**
1	15,3700		16,6000	13,80000	19,73000	32,48000
10	1,52900		1,65200	1,37400	1,99700	3,31000
100	0,15530		0,16840	0,15510	0,23360	0,41400
1000	0,01833		0,02124	0,03339	0,05790	0,13778
10000	0,00501		0,00713	0,02688	0,05431	0,14571

Если сравнить результаты теста с результатами чтения из множества колонок, то можно отметить значительное уменьшение времени на выполнение операций. Это ещё один плюс в пользу сериализации данных в единую строку, так как и операции чтения, и операции вставки с одной длинной строкой куда выгоднее операций над множеством колонок.

2.5 Объединение в единую транзакцию

Объединим операции чтения данных в одну транзакцию, аналогично, как было сделано в операциях вставки в единой транзакции. Условия теста и результаты приведены в таблицах 27 и 28.

Таблица 27 – Условия выполнения теста

Настройки SQLiteOpenHelper	**Устройство**	**Структура БД**	**Операция**
По умолчанию	Pixel 2 API 24 (AVD)	Таблица: Колонка Integer Переменное количество записей	Чтение значений с помощью функции Query в единой транзакции

Таблица 28 – Результаты теста

Количество операций подряд	**Среднее время на выполнение одной операции (мс)**				
	Количество записей в таблице:	1000	10000	100000	1000000
1	1,47000		14,5400	824,760	1707,85
10	0,14000		1,43600	82,4720	171,430
100	0,01340		0,14540	8,26910	17,0568
1000	0,00312		0,01621	0,82716	1,70652
10000	0,00031		0,00350	0,08453	0,17345

Время выполнения операций осталось практически неизменным. Поэтому данный подход не имеет смысла применять на операциях чтения.

2.6 RawQuery

Существует альтернативная функция для получения данных из БД, которая также возвращает курсор для чтения и страницу результатов. Это функция RawQuery. Она принимает на вход «сырой» SQL запрос, который компилирует, подставляя переданные параметры, если таковые имеются, и затем выполняет. Условия теста и результаты приведены в таблицах 29 и 30.

Таблица 29 – Условия выполнения теста

Настройки SQLiteOpenHelper	**Устройство**	**Структура БД**	**Операция**
По умолчанию	Pixel 2 API 24 (AVD)	Таблица: Колонка Integer Переменное количество записей	Чтение значений с помощью функции RawQuery

Таблица 30 – Результаты теста

Количество операций подряд	**Среднее время на выполнение одной операции (мс)**				
	Количество записей в таблице:	**1000**	**10000**	**100000**	**1000000**
1	1,22000		14,3400	838,299	1717,10
10	0,12700		1,42300	83,4710	172,027
100	0,01600		0,14580	8,39010	17,2879
1000	0,00577		0,01829	0,84256	1,72294
10000	0,00059		0,00599	0,08863	0,17584

В целом, можно утверждать, что использование данной функции не оказало значимого положительного эффекта на время выполнения операций. В основном, результаты оказались даже немного хуже, чем при использовании обычной функции Query.

2.7 Write Ahead Logging

Особенности данного режима уже были рассмотрены ранее при рассмотрении операций вставки записей. Следует включить режим и проверить, не падает ли эффективность выполнения операций считывания более, чем это заявлено в документации (1-2%). Условия теста и результаты приведены в таблицах 31 и 32.

Таблица 31 – Условия выполнения теста

Настройки SQLiteOpenHelper	**Устройство**	**Структура БД**	**Операция**
Включённый режим WAL	Pixel 2 API 24 (AVD)	Таблица: Колонка Integer Переменное количество записей	Чтение значений с помощью функции Query

Таблица 32 – Результаты теста

Количество операций подряд	**Среднее время на выполнение одной операции (мс)**				
	Количество записей в таблице:	**1000**	**10000**	**100000**	**1000000**
1	1,17000		14,5000	829,690	1710,03
10	0,11300		1,45100	82,9590	171,332
100	0,01630		0,14710	8,30550	17,0311
1000	0,00571		0,01851	0,83657	1,71319
10000	0,00057		0,00607	0,08829	0,17441

Очевидно, что время выполнения операций чтения действительно практически не изменилось, как и было указано в документации.

Операции вставки строк следует оборачивать в транзакции. Это сильно ускоряет процесс выполнения операции, особенно если требуется большое количество вставок.

Чтобы ещё более сократить время выполнения операции вставок следует включить режим WAL на постоянной основе, однако следует предварительно рассмотреть наиболее часто используемые операции при работе с БД, так как наибольшая выгода использования этого режима – при частых вставках в базу.

Для вставки данных рекомендуется использовать функцию insert библиотеки SQLiteDatabase или прекомпилированные запросы (последние наиболее предпочтительны). Для чтения следует использовать функции query и rawQuery.

Большое количество колонок показало негативное влияние как на время выполнения вставки данных, так и на выполнение операций чтения данных. В случаях с большим количеством колонок следует рассмотреть возможность сериализации данных и их упаковки в одну.

Список литературы

1. SQLiteOpenHelper | Для разработчиков Android | Android Developers / Android developers – Режим доступа: https://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper, свободный. (Дата обращения: 11.04.2020 г.).
2. Марашан М. В. Сравнение производительности ORM-библиотек как критерия выбора для работы с базой данных SQLite на устройствах с ОС Android // Молодой учёный. — 2016. — №26. — С. 146-149. — Режим доступа: https://moluch.ru/archive/130/36100/, свободный. (Дата обращения: 22.03.2020 г.).
3. Sunny K. A, Vikash K. K., Android SQLite Essentials. United Kingdom: Packt 2014. 110p.
4. SQLite Performance and Prepared Statements -- Visual Studio Magazine [Электронный ресурс] / Visual Studio Magazine – Режим доступа:

https://visualstudiomagazine.com/articles/2014/03/01/sqlite-performance-and-prepared-statements.aspx, свободный. (Дата обращения: 10.04.2020 г.).
5. Write-Ahead Logging [Электронный ресурс] / SQLite Org – Режим доступа: https://www.sqlite.org/wal.html, свободный. (Дата обращения: 04.04.2020 г.).

References

1. SQLiteOpenHelper | For Android Developers | Android Developers / Android developers – Access mode: https://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper, free. (Request date: 11.04.2020 y.).
2. Marshan M. V. ORM libraries performance comparison as a selection criterion for working with the SQLite database on Android devices // Young scientist. — 2016. — №26. — P. 146-149. — Access mode: https://moluch.ru/archive/130/36100/, free. (Request date: 22.03.2020 y.).
3. Sunny K. A, Vikash K. K., Android SQLite Essentials. United Kingdom: Packt 2014. 110p.
4. SQLite Performance and Prepared Statements -- Visual Studio Magazine [Electronic resource] / Visual Studio Magazine – Access mode: https://visualstudiomagazine.com/articles/2014/03/01/sqlite-performance-and-prepared-statements.aspx, free. (Request date: 10.04.2020 y.).
5. Write-Ahead Logging [Electronic resource] / SQLite Org – Access mode: https://www.sqlite.org/wal.html, free. (Request date: 04.04.2020 y.).
1. The use of corrective capabilities of codes to ensure fault tolerance. Petlevanny S.V., Sagdeev A.K. Modern high technology. 2007. No.4. P. 41-42