Performance analysis on Android SQLite database

Nikola Obradovic
Faculty of Electrical Engineering
University of Banja Luka
Banja Luka, Bosnia and Herzegovina
nikola.obradovic@etf.unibl.org

Aleksandar Kelec
Faculty of Electrical Engineering
University of Banja Luka
Banja Luka, Bosnia and Herzegovina
aleksandar.kelec@etf.unibl.org

Igor Dujlovic
Faculty of Electrical Engineering
University of Banja Luka
Banja Luka, Bosnia and Herzegovina
igor.dujlovic@etf.unibl.org

Abstract – **Database performance is a very important factor in the development of any application. Some applications need very fast feedback, others process a large amount of data and without the support of modern database management systems, this is almost impossible. In this paper main focus is to analyze SQLite Android relational database management system and test its performance in several scenarios: dealing with CRUD operations on unencrypted data, encrypted data, as well as concurrent access to the database. For this purpose, an application that demonstrates performance impacts on the SQLite database has been developed and published on the Google Play Store. Testing results show that operations over encrypted data take much longer than on unencrypted data. Furthermore, it is shown that a new SQLite locking and journaling mechanism provides concurrency with an efficiency of 40-44%.**

Keywords: Android, database, performance, encryption, concurrency

I. INTRODUCTION

The mobile market is continuously growing. Currently, there are billions of Android mobile devices. Android is a Linux-based mobile operating system developed by Google and OHA's Open Handset Alliance consortium. Based on market research, Android has become the most used operating system on mobile devices within a few years. The source code is publicly available, which means that anyone can modify it and adapt it to the products it produces. The number of available applications on the Google Play Store in December 2018. is 2.6 million [1], and this is statistics for only one Android market.

As this market is growing, hardware becomes more powerful and data computation possibilities from mobile devices are also increasing. Most of the Android applications need some kind of database management system, e.g. store user-specific data etc. as this is the most common way of storing and managing data. In most cases, these applications do not depend on database performance, but there are certainly some applications where database performance is an important factor. In this paper, the SQLite Android relational database management system (RDBMS) [2] is tested in several scenarios that are used in everyday applications. Relational databases are a subset of Android embeddable databases. Embeddable databases are self-contained, lightweight libraries with no server component. They also have limited resource requirements and low memory and power consumption. Most common types of embeddable databases are relational, objects, key-value pairs and XML documents. Besides SQLite, there are several more RDBMS for the Android platform, such as BerkeleyDB, AlaSQL, Interbase, Linter, etc [3]. Some of the previous mentioned RDBMS are faster than SQLite, but the main focus of this paper is SQLite RDBMS because many believe that SQLite is the most widely deployed SQL database in the world. The widest use of the SQLite relates to mobile applications (including Twitter and Facebook) [4, 5]. It is reported that SQLite is deployed on billions of different devices[6].

This paper deals with an analysis of the performance of the SQLite database when it comes to basic data operations, i.e. CRUD, and investigates whether it can respond to the demands of modern applications. It focuses mainly on the following three aspects: database operations on unencrypted data, encrypted data, as well as concurrent access to the database. Section 2 describes SQLite Android RDBMS as well as its position within the Android operating system. Performance tests and their results are discussed in section 3. This section also introduces an application, called DBInspector, which performs testing and performance measurement and conducts results. Section 4 focuses on related work and research to deal with mobile databases and performance impacts.

II. ANDROID AND SQLITE RDBMS

Android is one of the operating systems designed for mobile devices with advanced hardware capabilities. Like on Windows Mobile and Apple iPhone operating systems, Android has enabled the creation of diverse and advanced applications on mobile devices that are accompanied by a simple and rich development environment for their development. The biggest problem was to optimize it for working on mobile devices, which have limitations in memory, processing power and energy consumption. Previous issues have led Google to review the standard implementation of JVM (Java Virtual Machine) and create a Dalvik VM.

The core of the Android platform is the Linux kernel, responsible for device drivers, access to resources, power management, and other tasks performed by the operating system. Above the kernel, there is a group of C / C ++

978-1-5386-7073-6/19/$31.00 ©2019 IEEE

libraries, including SQLite, each of which has certain functionality required by the application frame on the level above (Figure 1).

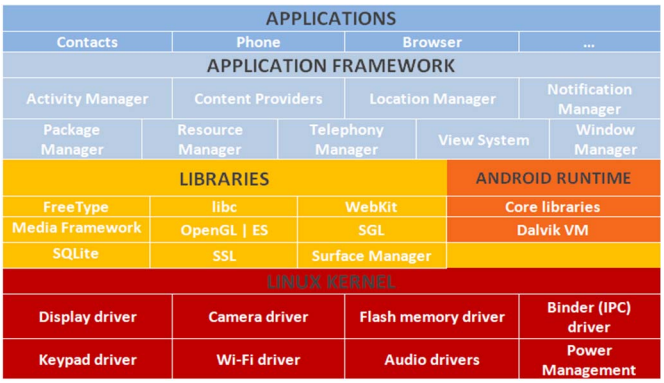

Figure 1. Android operating system architecture

SQLite is an open-source C library that implements a small, self-contained, serverless, zero-configuration, transactional SQL RDBMS and is designed for devices with limited hardware capabilities. It is also called a lighter version of SQL and it is available in most mainstream programming languages. This library supports almost all the relational database features and it has support for all major mobile operating systems: Android, iOS, Blackberry and Windows phone [7]. SQLite can be stored both on disk as well as in memory.

The application uses SQLite functionality through simple functional calls, which reduce the latency in database access because functional calls within one process are more efficient than interprocess communication. Complete database (definition, tables, indexes and data itself) is one interplatform file on the host machine. This simple design is achieved by locking the entire database file at the beginning of the transaction. This also means that once created an SQLite database on one device can be used on another device with an entirely different architecture by simply copying the cross-platform file [8].

SQLite implements most of the SQL-92 standard. For example, it supports the most complex searches, but also has some limitations, e.g. features as RIGHT and FULL OUTER JOIN, writing to VIEWs, GRANT and REVOKE are not supported, there is no complete ALTER TABLE support and no complete trigger support [9].

III. PERFORMANCE TESTS

Benchmarking of SQLite performance is a very complex task because it depends on a number of parameters, such as device hardware and OS version, file system type, database configuration, transaction-based queries, etc. Paper [10] shows that benchmarking of SQLite often leads to inconsistent results when it comes to the most used tools. Conducted results often differ by more than 50%. Therefore, this paper is not focused on the database environment and external factors, but on the raw data and database operations.

The basic approach in the database performance testing is the execution of CRUD operations over large datasets. In order to automate the testing process, the tool called DBInspector [11] has been developed. Figure 2 shows an application interface.

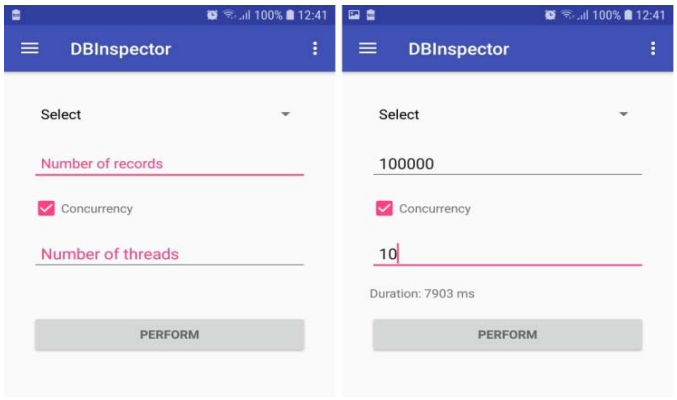

Figure 2. DBInspector testing tool

DBInspector provides CRUD operations on unencrypted data, as well as encrypted data and shows execution time for each operation. Additionally, it enables concurrent access to the database as well as the ability to choose whether the certain operation will be executed in a common transaction or not. The application has been published on the Google Play Store so everyone can use it.

Following tests have been conducted by the Samsung Galaxy A5 with octa-core 1.9 GHz CPU and Android 8.0 (code name Oreo) operating system.

A. *Unencrypted data*

First test scenario includes database operations on plain text, i.e. unencrypted data. A database with a simple table containing three columns (one textual and two integers) has been created for a testing purpose. An execution time of all CRUD operations for different numbers of records has been measured and the result is shown in Figure 3.

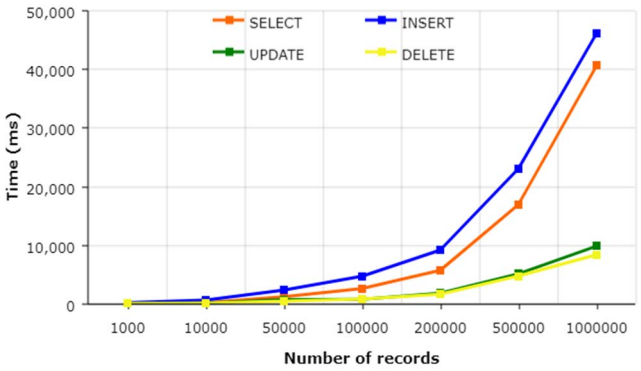

Figure 3. CRUD operations with unencrypted data

Figure 3 shows that as the number of records increases, operations INSERT and SELECT become significantly more demanding, i.e. the time needed for their execution increases

exponentially. However, an execution time of the UPDATE and DELETE increases much slower and is more linear.

When it comes to SELECT queries, it is well known that creating and maintaining indexes in the database could significantly improve the performance of data retrieval. However, indexes also come with some overhead [12]: they will use additional space and can increase the time it takes to insert or update records because SQLite needs to keep the indexes up to date. Therefore, in this test case indexes were not used in order to measure the execution time of each operation as precise as possible.

In addition, special attention should be paid on the way of execution of the certain operation, i.e. if all records are processed within a common transaction or within separated transactions. SQLite documentation says that changes to the database can be made only within a transaction. Any command that changes the database will automatically start a new transaction if one is not already in progress. Transactions started in that way are committed when the last query finishes [13]. So, if the number of transactions is minimized, then the disk access is also minimized and performance is maximized. Figure 4 shows the difference in performance when all records are inserted into the database within a single transaction compared to when each record is inserted with a separate transaction. This is also provided by the DBInspector. Tests conducted in [14] show that a single transaction is more than ten times faster (which can also be seen in Figure 4).

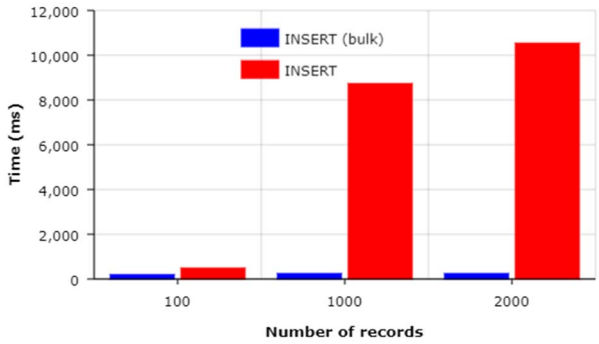

Figure 4. Insert operation: single transaction vs. separated transactions

B. *Encrypted data*

Second test scenario includes database operations on encrypted data. This scenario is also very important and sensible because many applications need to secure their data and store it in encrypted form. For data encryption, DBInspector uses the SQLCipher library [15]. SQLCipher is an SQLite extension that provides encryption of database files using 256-bit AES.

Figure 5 shows an execution time of CRUD operations for different numbers of encrypted records. It is shown that DELETE and UPDATE take a long time as opposed to SELECT. For UPDATE, this can be explained by the fact that each record needs to be decrypted first, then updated and then encrypted again. However, it is not known why the deletion takes so long. One of the possible explanations is that the SQLite performs the reorganization of remaining data after deletion and their afresh decryption/encryption. Another explanation for the poor performance of DELETE and UPDATE could be non-indexed columns. As mentioned before, indexes were not used in this research. Given that UPDATE and DELETE operations have WHERE clause, SQLCipher has to do a table scan when searching for a non-indexed column. This could be quite slow because every record has to be decrypted.

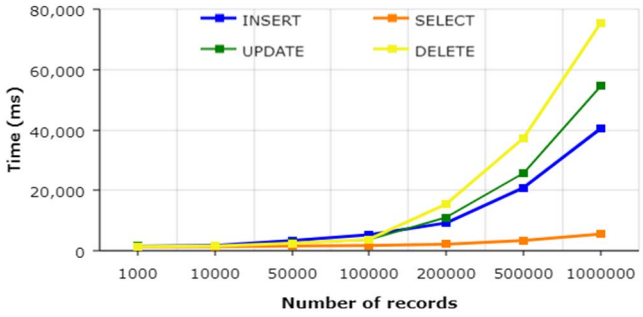

Figure 5. CRUD operations with encrypted data

C. *Concurrency*

The third test scenario deals with concurrent access to the SQLite database. From the SQLite version 3.0, it is introduced a new locking and journaling mechanism. The main goal of the new mechanism is improved concurrency and reduced writer starvation issue. The locking mechanism provides five states in which database file can be: UNLOCKED, SHARED, RESERVED, PENDING and EXCLUSIVE [16]. In short words, it means that any number of processes can read from a database file at the same time (SHARED). However, while reading is in progress, other threads are not allowed to perform operations of writing. If some process wants to write to the database file, it must to sign up (RESERVED, PENDING) and wait for all readers to end their sessions. RESERVED lock can be held only by one process at a time. During this period any other process can read from the database.

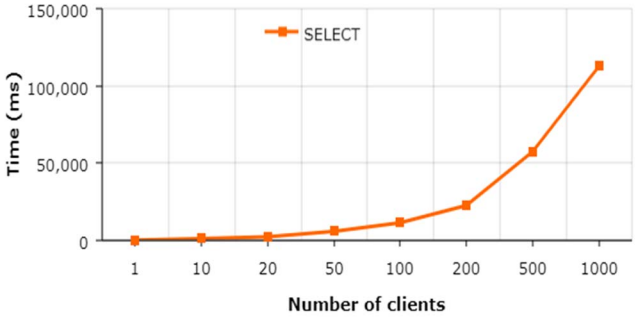

Figure 6. Concurrent database access with SELECT operation

Considering those facts, it makes sense to test concurrency only with the SELECT operation. Figure 6 shows an execution time of SELECT operation for different numbers of threads, i.e. database clients. Each client intended to read ten thousand records from the database at the same time. The testing shows

that SQLite locking and journaling mechanism does not produce excellent results when it comes to concurrency. Actually, based on test results, it can be calculated that the concurrency level is about 40-44%. The concurrency level can be explained as how much time it takes more to execute the same operation for multiple clients in relation to one client.

IV. RELATED WORK

To our best knowledge, there are no plenty research papers related to performance measurement of the SQLite database especially when it comes to a comparison on CRUD operations for unencrypted and encrypted data, as well as performance on concurrent access to the database. However, there are some papers dealing with SQLite benchmarking. They mostly use benchmarking tools like Mobibench, RLbench, Androbench to evaluate performance [10].

Purohith et al. [10] have shown that the SQLite benchmarking is very complex and that many parameters affect the results of testing. In some examples, tunning of database parameters can produce a 28X difference in performance, and this could lead to incomplete and misleading evaluations. Even benchmarking tools show up to 50% difference in performance when default configurations are used [13].

Tuan et al. [17] have dealt with IO characteristics of SQLite transaction in Android platform. Their study has revealed that SQL transactions are extremely inefficient.

Lim et al. [18] have developed Androtrace, IO trace and analysis framework, and shown that SQLite could cause a lot of IOs overhead while data and journal files are committing to the storage. Additionally, they showed that write operation is dominant when it comes to database performance.

Research conducted by [19] has compared ZeroDB cryptographic databases with SQLite among others. The authors have shown that the communication overhead of ZeroDB could lead to significant drops in performance compared to the non-cryptographic databases, in particular for write and update operations. This confirms the results obtained in this paper when it comes to the comparison of operations over unencrypted and encrypted data.

V. CONCLUSION

This paper shows what can be expected from the SQLite database when performing basic operations over large datasets. Special attention was paid on the comparison of performance when it comes to unencrypted and encrypted data. Testing results show that operations over encrypted data take much longer than on unencrypted data. Furthermore, it is shown that INSERT and SELECT operations take significantly longer than UPDATE and DELETE when it comes to unencrypted data. However, when it comes to encrypted data the results are almost the opposite. Additionally, testing about concurrency shows that a new SQLite locking and journaling mechanism provides concurrency with an efficiency of 40-44% which cannot be particularly worthy of laud.

As the next step in research, our goal is to conduct a detailed analysis to determine why UPDATE and DELETE operations are slower with encrypted data. Also, the plan is to extend available options in the application, based on recommendations in [10], in order to determine what differences will be shown for CRUD operations when database parameters vary. In the updated application the user will be allowed to define parameters in order to conduct a customized benchmark.

REFERENCES

[1] https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, last access: 15.01.2019.

[2] SQLite. SQLite transactional SQL database engine. http://www.sqlite.org/, last access: 15.01.2019.

[3] http://greenrobot.org/news/mobile-databases-sqlite-alternatives-and-nosql-for-android-and-ios/, last access: 15.01.2019.

[4] Why to use SQLite in Android. https://www.quora.com/Why-should-we-use-SQLite-in-Android-development, last access: 15.01.2019.

[5] SQLite in Android. http://www.grokkingandroid.com/sqlite-in-android, last access: 15.01.2019.

[6] SQLite. Most Widely Deployed SQL Database Engine. https://www.sqlite.org/mostdeployed.html, last access: 15.01.2019.

[7] Features of SQLite, https://www.sqlite.org/features.html, last access: 17.01.2019.

[8] Single-file Cross-platform Database, https://www.sqlite.org/onefile.html, last access: 17.01.2019.

[9] SQL Features That SQLite Does Not Implement, https://sqlite.org/omitted.html, last access: 17.01.2019.

[10] D. Purohith, J. Mohan, and V. Chidambaram, "The Dangers and Complexities of SQLite Benchmarking," Department of Computer Science, University of Texas at Austin, 2017, Conference: the 8th Asia-Pacific Workshop.

[11] A. Kelec, DBInspector, https://play.google.com/store/apps/details?id=org.unibl.etf.dbinspector, last access: 15.01.2019.

[12] Squeezing Performance from SQLite: Indexes? Indexes!, https://medium.com/@JasonWyatt/squeezing-performance-from-sqlite-indexes-indexes-c4e175f3c346, last access: 15.01.2019.

[13] SQL As Understood By SQLite, https://sqlite.org/lang_transaction.html, last access: 15.01.2019.

[14] Squeezing Performance from SQLite: Insertions, https://medium.com/@JasonWyatt/squeezing-performance-from-sqlite-insertions-971aff98eef2, last access: 15.01.2019.

[15] SQLCipher, https://guardianproject.info/code/sqlcipher/, last access: 15.01.2019.

[16] https://www.sqlite.org/lockingv3.html, last access: 15.01.2019.

[17] D. Q. Tuan, S. Cheon, and Y. Won, "On the io characteristics of the sqlite transactions, " in Proceedings of the International Conference on Mobile Software Engineering and Systems, MOBILESoft '16, pp. 214–224, New York, NY, USA, 2016. ACM.

[18] E. Lim, S. Lee, and Y. Won, "Androtrace: framework for tracing and analyzing IOs on Android, " in Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads, pp. 1-8, October 04-04, 2015, Monterey, California