Mobile Data Store Platforms: Test Case based Performance Evaluation

Karim Kussainov and Bolatzhan Kumalakov
School of Science and Technology, Nazarbayev University, Kabanbay batyr 53, Astana, Republic of Kazakhstan

Keywords: Mobile Database, SQLite, Realm, SnappyDB, Performance Evaluation.

Abstract: Mobile applications are an important tool in knowledge management, as they collect and process massive amount of user data. Day-to-day usage of mobile services has rocketed by factors over the last decade. Average mobile device user installs multiple social network, messaging, professional and leisure applications. Saving and retrieving associated data becomes a challenging task in the light of the growing number of applications on a single device. While industry offers several well established platforms, such as BerkeleyDB and UnQLite, we examine comparatively poorly examined Realm and SnappyDB against industry standard - SQLite. In particular we are interested in performance and code maintainability, and use a test case in order to asses them. Results revile that SQLite shows the poorest performance, while Realm provides the most intuitive way of matching data to the application logic due to its object-oriented nature.

1 INTRODUCTION

The number of mobile services has rocketed by factors over the last decade. Average handheld device user installs multiple social network and messaging applications, apart from work related and, possibly, gaming software. Storing state variables, user preferences and other data segments puts pressure on mobile database engines, which have to cope with the workload in order to support desired user experience.

Industry offers a range of solutions. SQLite is a widely adopted mobile database management system (Kreibich, 2010). It implements relational data model, and natively supports industry standard - SQL language specification. All major mobile operating systems in the market support it, while Android comes with the pre-installed version by default.

Major players in the market offer at least a dozen alternative solutions, such as BerkeleyDB and UnQLite. Most of them are well tested and positioned against each other. NoSQL movement, however, also offers emerging, but promising SnappyDB (Hachicha, 2016) and Realm (Realm, 2016) database engines.

SnappyDB is an open source “key-value” database management system for Android OS. It is based on Googles LevelDB storage library, which was initially designed to overcome relational data model performance boundaries by eliminating notions of relation and referential integrity. It also makes advanced use of compression algorithms, thus, attempts to optimize disc space usage as well.

Realm, on the other hand, is a cross-platform, object-oriented database engine that can be deployed on Android and iOS devices. It requires storable objects to extend RealmObject class and define its access methods. The data is stored encapsulated, and access operations are performed in an object-oriented manner.

This paper presents comparative performance evaluation of SQLite, SnappyDB and Realm database engines. Results revile that SQLite shows the poorest performance, while Realm provides the most intuitive way of matching data to the application logic due to its object-oriented nature.

Remainder of the paper is structured as follows: Section 2 defines research design, including test cases, evaluation criteria and tools. Section 3 presents experimental results and corresponding discussion, while Section 4 concludes the paper.

2 RESEARCH DESIGN

Mobile databases are local storages, which operate on machines with limited computing and storage capabilities. In order to ensure realistic evaluation results, test runs are performed on the same mobile device, one after another. They do not overlap, but run on top of a realistic set of common user applications, such as WhatApp, Instagram and Telegram.

Kussainov, K. and Kumalakov, B.
Mobile Data Store Platforms: Test Case based Performance Evaluation.
DOI: 10.5220/0006032300950099
In *Proceedings of the 8th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2016) - Volume 3: KMIS*, pages 95-99
ISBN: 978-989-758-203-5

2.1 Experiment Design

2.1.1 Read and Write Performance Evaluation Design

Since mobile database systems follow centralized DBMS architecture pattern, concurrent queries are organized into a queue. The database engine then processes it using threads. This in the worst case is a sequential process, if majority of processes try to process the same data segment. Thus, instead of modelling concurrent processes that perform read/write operations, we utilize single test application that performs them with data segments of different size. Database object/table structure stores two fields/columns: planet and radius.

In order to test write performance we: first, initiate single entry of “mars” and “3000” values; second, perform multiple entries of 10000 random character long strings. Corresponding processing times are logged by the client application.

Read performance, on the other hand, is tested using two experiments. First queried number of entries in the database and returned their total number. Prior to execution, SQLite and Realm got 30000 randomly generated entries per entity/RealmObject. SnappyBD, however, searches values by its key prefix. Thus, in order to generate 30000 values it required generating corresponding number of unique keys, while in order to perform the operation it is not necessary to apply count function to the number of returned values. In SQLite query results had to be processed using following method:

```
cursor.getCount();
```

For the second experiment we query all the planets, where radius is less than 4000. Data set contained 12000 objects/keys-value pairs that satisfy the constraint. Similarly to the previous experiment, SQLite and Realm utilize build-in functionality:

```
select * from planets Where radius<4000;
```

for SQL and

```
realm.where(Planet.class).lessThan(radius, 4000)
.findAll();
```

for Realm respectively. SnappyDB, on the other hand, adds the key to the list every time it comes across one that starts with “Planet,” which also corresponds to a value less than 12000.

Second, we measure RAM consumption during 10000 character long string value write operations execution. In order to do so we use native android memory investigator and Cool Tool application. Both report memory usage, and we were interested in average and maximum consumption rate.

Finally, we alternate test application functionality and measure how many lines of code had to be modified in terms of database access. Which includes database query commands, response handling and data matching.

2.1.2 Test Application Design

Test application implements three core functions: adding entry, querying data and deleting them. The use case diagram (Figure 1) illustrates the concept using UML 2.0 use case diagram.

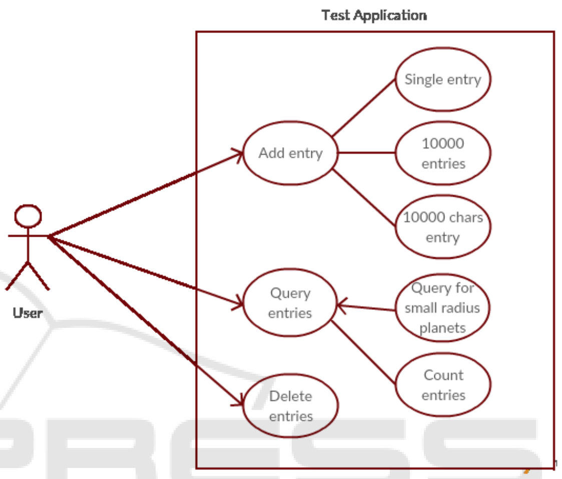


Figure 1: Figure presents UML 2.0 Use case diagram, which visualizes Test application functionality.

For the experiment proposes we developed three Android applications, which implement the same functionality. Most of the code was reused to guarantee that the only difference is in data store handling part.

Figure 2 illustrates how write operation is performed on its implementation level.

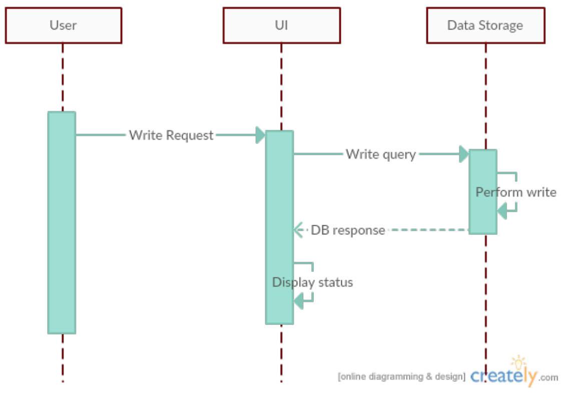


Figure 2: Figure presents UML 2.0 Sequence diagram for ”Add entry” procedure.

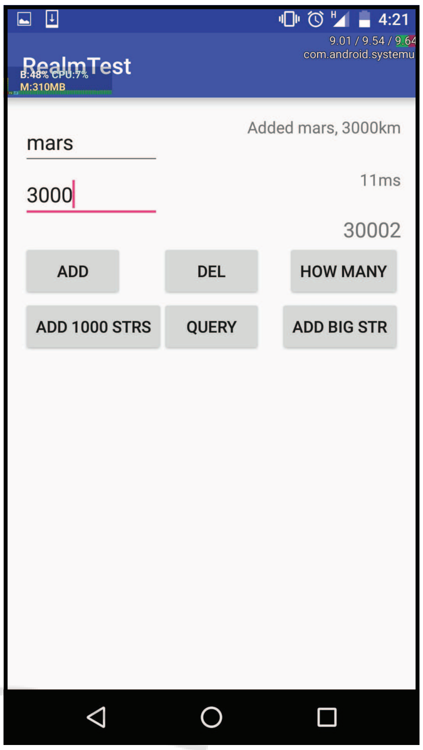
2.2 Implementation Details

Test applications were developed using Android Studio version 1.5.1, and installed on LG Nexus 5. The device runs Android 6.0.1, and has quad-core Snapdragon 800 CPU (2.26 MHz) with 2 GB of RAM. SQLite database engine came with the operating system, while SnappyDB and Realm had to be installed manually.

We performed following operations in order to configure the environment:

- set "multiDexEnabled" to *true* in application level "build.gradl" configuration file to facilitate work with the SQLite database.
- add SnappyDB (version 0.5.0) .jar file to the "jniLibs", and corresponding field to the *dependencies* field in application level "build.gradl".
- install Realm (version 0.88.3) plugin, and add class path dependency to the project level "build.gradl".

Initial application design includes EditText fields, TextViews and several buttons. EditText fields are used to add an entry to the database. TextViews display results such as status logs and time consumed. Every operation is written as a button handler. Figure 3 presents a screenshot of the testing client application.



Added mars, 3000km

mars

11ms

3000

30002

Figure 3: Figure presents a screenshot of the client application, which performs test entries and logs the execution time. When the data is collected it is loaded to the MS Excel software to derive statistical data.

3 EXPERIMENTAL RESULTS

3.1 Test Results

3.1.1 Memory Consumption

The experiment is designed in such a way, that it is impossible to completely separate RAM usage of the database engine and common user applications. Nonetheless, given that we avoided active use of these applications (only the background services run constantly) it is possible to assume equal conditions. When in stand-by mode average memory use of the mobile device ranged from 200 KB to 2 MB. Table 1 presents memory consumption data during the tests.

Table 1: Table presents memory consumption data.

Platform name	Average	Maximum
SQLite	81 MB	115 MB
SnappyDB	12 MB	66 MB
Realm	15 MB	69 MB

RealmTest and SnappyTest applications used maximum of 60-70 MB, while SQLite consumed up to 115 MB.

SnappyDB showed the lowest memory consumption of about 12 MB and Realm needed up to 15MB on average. SQLite showed different results in a different tests, ranging from 24 to 130 MB. Refreshing rate of the app is one second and both SnappyDB and Realm usually use less than one second.

3.1.2 Read/Write Performance

Difference in performance when writing single entry is very little. SnappyDB shows the slowest result, there is no sharp decline in performance for writing a string of 10000 characters and a number. However, there is a considerable difference in performance during high amount of writing operations. While Realm and Snappy manage to push 10000 data pairs within 1 second, it takes about 80000 seconds for SQLite to do the same. Please, note the logarithmic scale in the Figure 4.

Realm is an undisputed leader in querying operations. It takes 8 ms to count 30000 items and 3-4 seconds to perform a simple query. SnappyDB needs

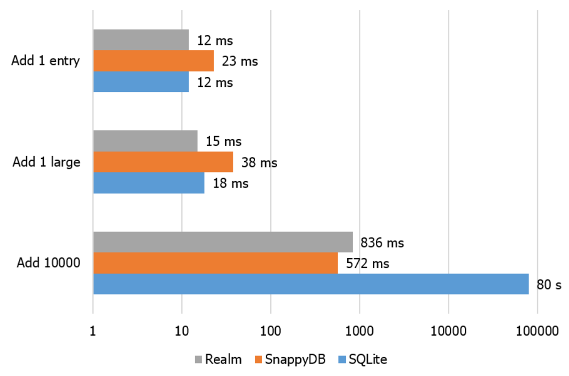


Figure 4: Figure visualizes write performance data.

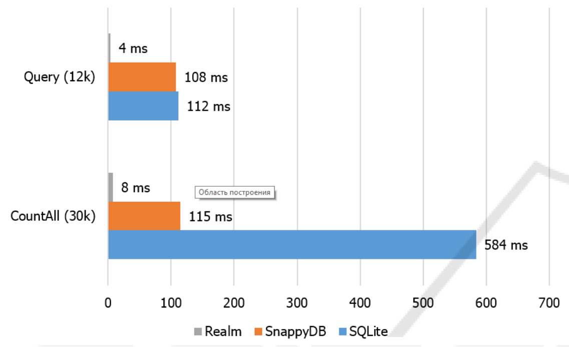


Figure 5: Figure visualizes read performance data.

100-120 ms in order to perform the same operations. SnappyDB exhibits no difference between 30000 and 12000 elements to count. It takes about half of a second for SQLite to count 30000 rows in one entity and about 112 ms to perform a simple query and write result to the StringBuffer.

Finally, the standard VM heap size for Nexus 5 is fixed at 192 MB. In order to have reliable result, it was manually increased through android.manifest file to 512 MB.

3.1.3 Code Maintanability

Realm uses separate java classes for each type of objects (+20 lines in this example). It uses 1 line query. Snappy querying is more complex due to the database initialization and try-catch methods. On the other hand, there is a simplest adding procedure in SnappyDB. In Realm there are several lines of code with respect to class setters. In SQLite there is only one line of code.

3.2 Discussion

From the presented results we can see that platforms show close code maintainability and memory usage

Table 2: Code complexity (LoC).

Platform	AppTotal	Add entry	Query
SQLite	165	1	1
SnappyTest	186	3	5
RealmTest	159+20	6	1

characteristics. The only exception is the SQLite engine, which in extreme cases uses eight times as much RAM as the other ones. Of course, it may be justifiably argued that we consider highly unlikely scenarios, which do not normally accrue in the real world. Nonetheless, we try to take into account the high rate of mobile database load explained earlier.

From the performance perspective SQLite also shown poorer performance. In particular, it takes almost twice the time for performing write operations, and five times as much time for read operations. We are also aware that there are SQLite modifications that show better performance than the standard edition, such as (Bakibayev et al., 2012). However, installing and configuring them is not a standard procedure when the cell phone loses its service level. Hence, they are of limited access to a common user.

SnappyDB has limited functionality, but it seems to be the fastest in terms of reading/writing myriad of entries. While it is mostly due to its data model simplification (the simplest among considered), there is additional work for the software engineer when matching data segments to object fields of the program. Realm has also proven to be an alternative to SQLite database. It is also easier to match data segments, because it makes use of object encapsulation.

Nonetheless, within the described performance map choosing one database engine the other depends on the dataset. Some data do not fit into two-dimensional row-column structure, while extracting JSON segments create unnecessary computational load (Dash, 2013). Relational databases are suitable for complex queries, but lack flexibility, which results in difficulties when database scheme changes significantly. Hence, SQLite is a viable option for small to medium datasets and solutions with low concurrency. It is also widely accepted among mobile developer community.

4 CONCLUSION

Paper presents case based comparative evaluation of SQLite, SnappyDB and Realm mobile data store engines. Test results revile that all of the solutions distribute close complexity in terms of code maintainability and memory consumption.

In terms of performance, in extreme cases - such as writing and reading large entries - SQLite showed the poorest results, while Realm and SnappyDB are significantly faster. Nonetheless, SQLite better suits applications with medium datasets and low concurrency.

Mobile applications are an important tool in knowledge management, as they collect and process user side configurations, logs, etc. Information reviled in this studies may be used when engineering such systems to facilitate better data retrieval and save when needed.

ACKNOWLEDGEMENTS

This work is supported by Nazarbayev University (Astana, Kazakhstan) under its internal Social Policy Grant scheme.

REFERENCES

Bakibayev, N., Olteanu, D., and Zavodny, J. (2012). Demonstration of the FDB query engine for factorised databases. *PVLDB*, 5(12):1950–1953.

Dash, J. (2013). RDBMS vs. NoSQL: How do you pick? http://zdnet.com/article/rdbms-vs-nosql-how-do-you-pick/. [Online; accessed 11-August-2016].

Hachicha, N. (2016). A fast and lightweight key/value database library for android. http://www.snappydb.com/. [Online; accessed 20-April-2016].

Kreibich, J. A. (2010). *Using SQLite - Small. Fast. Reliable. Choose any Three*. O'Reilly.

Realm (2016). Realm is a replacement for sqlite and core data. https://realm.io/. [Online; accessed 25-April-2016].