Självständigt arbete på grundnivå

Independent degree project – first cycle

Datateknik
Computer engineering

Room vs. greenDAO for Android
- a comparative analysis of the performance of Room and greenDAO

Bernard Che Longho
MITTUNIVERSITETET

DSV Östersund

Examiner Dr. Felix Dobslaw Felix.Dobslaw@miun.se
Supervisor Raja-Khurram Shahzad Raja-Khurram.Shahzad@miun.se
Author Bernard Che Longho lobe1602@student.miun.se
Program Programvaruteknik, 180hp
Course DT113G, Självständigt Arbete
Field of study Computer Engineering
Semester, year HT, 2019
Abstract

Android developers often store relational data using databases. There are a couple of database frameworks from which an Android developer can choose. Developers usually consider application performance when selecting libraries and frameworks for their application. Unfortunately, there are limited studies that compare the performance of Android database frameworks. This study presents the results of a comparative analysis of the performance of two of the most popular SQLite-based database frameworks, Room Persistence Library (Room) and greenDAO. The Performance metrics used in this project were execution time for database create, read, update and delete (CRUD) operations, amount of RAM and CPU used by the device when performing the CRUD operations, and the effect of the application size when using Room or greenDAO. In order to achieve the objectives, three versions of an application was developed. One version used greenDAO, the other used Room while the third used SQLite for data storage. The results suggest that greenDAO performs better in all the database CRUD operations which could be explained by the fact that greenDAO does not cascade delete and update operations. Another finding is that the resulting application size was the same for both Room and greenDAO but was about 5% bigger than that for SQLite. The results also show that there was overall increase in RAM and CPU usage when using Room or greenDAO compared to SQLite. Future work should be done to compare relational models with database frameworks.

Keywords: Android, persistence, database, Room, greenDAO.
Sammanfattning

Android utvecklare oftast lagra objekt-orienterade data med hjälp av databaser. Det finns många databasramverk där Android utvecklare kan välja. Utvecklare oftast överväga prestanda när de välja bibliotek och ramverk för deras applikationer. Men det är tyvärr brist på undersökningar som jämföra de databasramverken för Android. För att bidra en lösning, presentera denna studie en jämförande analys av Room Persistence Library (Room) och greenDAO, två av dem mest använda SQLite-baserat databasramverk för Android. Dem prestandakriterier som studerats var exekveringstid för att lagra, läsa av, uppdatera, och ta bort data från databas (CRUD), RAM och CPU användning samt påverkan av applikations storlek när dessa ramverk används för utveckling. Metoden som användes i denna studie var att en Android applikation designerades och utvecklades i tre versioner, en använde Room, en annan använde greenDAO och en tredje använde SQLite som ramverk för data lagring. Den tredje utvecklades som bas för jämförelse. Resultatet visar att greenDAO är snabbare än Room för alla databas CRUD operationer, som kan förklaras av att greenDAO utför ingen databas cascading medan det gör Room. En annan rån är att det var ingen skillnad av applikations storlekt mellan Room och greenDAO. Däremot, användning av dessa ramverk resulterades i att applikations storleken var ungefär 5% större jämförde med den SQLite-baserade version. Resultatet visa också ett ökad storlek för RAM och CPU användning för den Room och greenDAO versioner jämförde med den SQLite versionen. Framtida studier ska undersöka hur databasramverk presterar med objekt-orienterad data.

Nyckelord: Android, databas, persistence, Room, greenDAO.
Table of Contents

Table of Tables ........................................................................................................ 7
Table of Figures ......................................................................................................... 8
Acronyms .................................................................................................................. 9
1 Introduction .......................................................................................................... 10
  1.1 Problem Statement ......................................................................................... 10
  1.2 Aim ................................................................................................................. 11
  1.3 Research questions ......................................................................................... 11
  1.4 Scope .............................................................................................................. 11
  1.5 Outline .......................................................................................................... 11
2 Background .......................................................................................................... 13
  2.1 Related Work ................................................................................................ 13
  2.2 Object-relational mapping .......................................................................... 13
  2.3 Database Cascade ......................................................................................... 15
  2.4 Some important terminology ....................................................................... 17
    2.4.1 Mobile Application success ................................................................. 17
    2.4.2 Persistence ............................................................................................... 17
    2.4.3 Application Performance ...................................................................... 17
3 Methodology ........................................................................................................ 18
  3.1 Approach ........................................................................................................ 18
    3.1.1 Application versions ............................................................................. 18
    3.1.2 Application model .................................................................................. 18
    3.1.3 Performance metrics ............................................................................. 18
  3.2 Implementation .............................................................................................. 19
    3.2.1 Functional Requirements ..................................................................... 19
    3.2.2 Non-functional Requirements ............................................................... 19
    3.2.3 Development environment ................................................................... 20
    3.2.4 Programming language ........................................................................ 20
    3.2.5 Device used for profiling ...................................................................... 20
    3.2.6 Experimental data .................................................................................. 20
4 Construction ........................................................................................................ 21
  4.1 Design ............................................................................................................. 21
    4.1.1 The Relational model ............................................................................ 21
    4.1.2 User interface .......................................................................................... 22
    4.1.3 Application overview ............................................................................ 23
  4.2 Database access ............................................................................................ 23
    4.2.1 SQLite version ......................................................................................... 23
    4.2.2 greenDAO version ................................................................................ 24
    4.2.3 Room version ........................................................................................ 25
  4.3 Benchmark .................................................................................................... 26
    4.3.1 Device preparation .................................................................................. 26
    4.3.2 Execution time ....................................................................................... 27
    4.3.3 Measuring CPU usage .......................................................................... 27
    4.3.4 Measuring RAM usage ......................................................................... 28
    4.3.5 Creating estimating the APK size ......................................................... 28
5 Results .................................................................................................................. 29
  5.1 Execution times ............................................................................................. 29
    5.1.1 Insert ....................................................................................................... 29
    5.1.2 Read ........................................................................................................ 29
    5.1.3 update ..................................................................................................... 30
    5.1.4 Delete ...................................................................................................... 30
5.2 CPU usage .................................................................31
5.3 RAM usage ..............................................................31
5.4 APK size .................................................................32

6 Discussion ..................................................................................33
   6.1 The results ........................................................................33
   6.2 The Implementation ..........................................................34
   6.3 Limitations and proposal for future work ..........................34
   6.4 Ethical considerations .......................................................34

7 Conclusion ..................................................................................35
References ....................................................................................36
Appendix A: Supplementary diagrams ........................................39
Appendix B: Raw data .................................................................41
Table of Tables

Properties of testing devices........................................................................................................20
Execution time to insert values into database..............................................................................41
Execution time to read data from database....................................................................................41
Execution time to update entries in the database..........................................................................42
Execution time to delete (all) items from the database....................................................................42
CPU usage (%) by different database frameworks...........................................................................43
RAM usage (MB) by different frameworks......................................................................................43
Table of Figures

Figure 1  Code snippet, creating database using Room ................................. 14
Figure 2  Code snippet, creating a database using SQLite .............................. 14
Figure 3  Code snippet, inserting a Customer into the database using SQLite ...... 15
Figure 4  Code snippet, inserting a Customer into the database using Room ....... 15
Figure 5  Code snippet, inserting a Customer into the database using greenDAO .... 15
Figure 6  Model UML and database relationships ........................................... 21
Figure 7  Application user interface ............................................................. 22
Figure 8  Program overview ...................................................................... 23
Figure 9  UML, SQLite database access ........................................................ 24
Figure 10 UML, greenDAO database access ................................................ 25
Figure 11 UML, Room database access ........................................................ 26
Figure 12 Code snippet, example code snippet for measuring execution time ..... 27
Figure 13 Popup menu after application apk generation with Android Studio 3.4 28
Figure 14 Average time for inserting values into the database ....................... 29
Figure 15 Average time to read values from the database ............................. 29
Figure 16 Average time to update entries in the database ............................. 30
Figure 17 Average time to delete items from the database ............................ 30
Figure 18 CPU usage (%) by the different app versions ................................ 31
Figure 19 RAM usage (MB) by the different frameworks .............................. 31
Figure 20 Application size of the different app versions ............................... 32
Figure 21 UML, MethodTimer class diagram .............................................. 39
Figure 22 UML, Full program overview .................................................... 40
Acronyms

APK       Android Package
AVD       Android Virtual Device
CPU       Central Processing Unit
CRUD      Create, Read, Update, Delete
DAO       Data Access Object
GUI       Graphical User Interface
IDE       Integrated Development Environment
JSON      JavaScript Object Notation
JVM       Java Virtual Machine
ORM       Object Relational Mapping
RAM       Random Access Memory
SQL       Structured Programming Language
UI        User Interface
UML       Unified Modeling Language
1 Introduction

The use of smart phones has increased in recent decades, with the number of users estimated to reach five billion in 2019 [1]. These devices often have applications that can be used to create, store, and share data like text, pictures, or videos. Such data are often stored when the application is running (application session) and retrieved after the application was stopped. The ability to store and retrieve data between application sessions is called persistence [2]. Applications can be categorized into three main categories based on their operating system, Android, iPhone, Windows. Android application developers have four major ways of achieving persistence [3]. One of the four ways is using Shared Preferences [4] which is used to store relatively small amount of data like application preferences [5]. The second way of persisting data is by using the device’s internal memory (hard drive). The third form is using an externally mounted micro Secure Digital (SD) card. Both the second and third options are suitable for storing information like videos, images, and text files. The fourth option for data persistence in Android applications is using a database [6], [7]. Database storage is suitable for storing relational data i.e data that are tightly linked to each other such as data about a users and their scores in a gaming app. There are many database frameworks [8] from which Android application developers can choose. Some of these frameworks are, SQLite [9], Room Persistence Library (Room) [10], ActiveAndroid [11], greenDAO [12], Realm [13], and OrmLite [14]. This plethora of database frameworks implies that Android developers must choose a framework for storing relational data. In order to make this choice, developers have increasingly considered application performance as a major factor [15]. Application performance plays an important role in the user-perceived quality of applications [16], [17],[18]. Using third-party libraries can also increase the overall application size which in turn increases the memory usage of the application [19]. Unfortunately, there are limited empirical studies that compare the performance of the different Android database frameworks. Pu J. et al., [20] carried out one study that studied the performance, and programming effort trade-offs of SQLite, ActiveAndroid, greenDAO, Realm and OrmLite. They found that SQLite performs best while ActiveAndroid performs worst. They also concluded that greenDAO performed best among the Object Relational Models (ORMs). They however never compared the performance of Room which is the second most used database framework for Android [24]. This project adds to this knowledge of performance analysis by analyzing the performance of Room and greenDAO.

1.1 Problem Statement

In order to persist relational data in their applications, Android developers often have to choose among database frameworks such as OrmLite, ActiveAndroid, Room and greenDAO. Both Room and greenDAO map objects to SQLite databases. As of the writing of this report, greenDAO is the fourth most used framework for database management in Android [21], coming after Firebase [22], Room [23], and Realm [13]. To the best of the author’s knowledge, there is no empirical study that analyses the performance of Room and greenDAO using relational data models. Consequently developers cannot easily make an
educated choice on which framework to choose. This study is, therefore, the first to empirically analyze the performance of greenDAO and Room using a relational model.

1.2 Aim

The aim of this project is to measure the performance of Room and greenDAO when persisting relational data on Android applications. It is expected that the results will contribute to the existing knowledge of performance measure of Android database frameworks which may help guide Android developers to make an informed decision when choosing an ORM framework for their application.

1.3 Research questions

The research questions are,

- How much time does Room take to perform database CRUD(create, read, update, and delete) operations compared to greenDAO?
- How much RAM and CPU are used by an Android application when executing database operations with Room or greenDAO?
- How does the use of Room or greenDAO affect the Android application size in comparison to SQLite?

1.4 Scope

Throughout this report, the term performance is used to refer to the time taken to execute a task (execution time) as well as the memory required by the Android system to perform a task. The resulting application in this project had just buttons with informative texts that represent the actions to be performed. This minimal design is enough to answer the research questions and limits unnecessary lagging due to the loading of complex user interfaces (UI). One Android Virtual Device (AVD) was used for the measurements. The AVD was created solely for the project and had no other applications installed. Having such a “clean” system helps give an accurate representation of the results as an external phone would have many other applications that might affect the measurements due to background processes/updates. It is worth to note that performance comparison is between Room and greenDAO, the SQLite version was created to compare the application size.

1.5 Outline

So far, this report has presented the introductory chapter that contains a brief introduction to the topic, followed by the research problem and aim. In the next chapter, the essential terminologies, as well as related work on the topic, are presented. The methodology is presented in Chapter 3. Here, the approach, as well as the requirements for the application, are presented. Chapter four, details the technical aspects of the implementation and the performance measurements. The results of the work are presented in chapter five and discussed in chapter six. The conclusion, as well as proposed future work and ethical considerations, are presented in chapter seven which is then followed by the cited
references and appendix. The appendix contains relevant code snippets, tables, graphs, and figures cited in the preceding chapters.
2 Background

This section contains information that is needed to understand the context of this thesis. It starts by defining some key terminologies and then highlights related research on the thesis topic.

2.1 Related Work

Performance plays a vital role in the success of applications. In their study to characterize and detect performance bugs for smartphones, Liu Y et al., [24] found that the most common performance bugs are a graphical user interface (GUI) lagging, energy leak, and memory bloat. GUI lagging is the phenomenon where the user has to wait for some task to complete before doing anything on the device. If this lasts for long, the Android operating system sends an “Application Not Responding” (ANR) error. Memory bloat on its part occurs when an application requires more RAM or CPU resources than it is available for the device. These two bugs feature among the top factors that users give low ratings for applications [25]. Guerrouj, L. et al., [26] has shown that user app ratings are directly proportional to the success of the application. It is, therefore, important to build applications that respond fast and use less memory.

The size of an application also determines the likelihood of the application download. It has been demonstrated that the more an application is downloaded, the more likely it is to be downloaded by other users [27]. This implies that, if the developers have to choose a third-party library, they should be aware of the overall impact that the library has on the application size.

Studies on the performance of database frameworks are limited. One study that attempts to address this is that done by Pu J. et al., [20]. They aimed to understand the energy requirement (battery usage), the execution time for CRUD operations by SQLite, ActiveAndroid, greenDAO, OrmLite, Sugar ORM and Java Realm, as well as the programming effort trade-offs when using any of these database frameworks. They used a Java database benchmark, the DaCapo [28] as well a set of microbenchmarks [29] for the measurements. According to their experiment, SQLite performed best followed by greenDAO, OrmLite, Java Realm, Sugar ORM, and ActiveAndroid (worst). Another attempt to benchmark the performance of different ORMs have been done by greenrobot [30]. They use a simple non-relational model to measure execution time for CRUD operations for different ORM frameworks. Their results show that greenDAO is fastest to perform Insert, Update, and Read operations than the other ORM frameworks [31].

2.2 Object-relational mapping

Object Relational Mapping (ORM) is a programming technique that enables query and manipulation of relational data from a database [32]. Martin Fowler refers to it as a layer of software that helps map objects to a database [33]. The main advantages us using ORMs is the easy in mapping plain objects to the database object and writing of less code. Both Room and greenDAO are frameworks that use SQLite underneath. They conceptually differ
from SQLite in that, the programmer does not require to use raw SQL syntax for querying and manipulating data from the database. To illustrate this, consider a situation where the in a Customer-Order relational system (as that of this thesis), creating the database in Room can be accomplished using

```java
//Database(entities = [Customer.class, Product.class, Order.class,
//  OrderProduct.class], version = 1) // database tables and version
abstract class AppDatabase extends RoomDatabase {
  private final static String databaseName = "customer_order_room";
  private static AppDatabase instance;
  static AppDatabase getInstance(final Context context) {
    if (instance == null) {
      synchronized (AppDatabase.class) {
        if (instance == null) {
          instance = Room.databaseBuilder(context, AppDatabase.class,
            databaseName).build();
        }
      }
    }
    return instance;
  }
}
```

![Figure 1: Code snippet, creating database table using Room](image)

A corresponding code snippet to create the databases in SQLite is as shown below (for brevity, the statement for creating Order-Product table is cropped).

```java
private final static String CREATE_CUSTOMER_TABLE = "CREATE TABLE IF NOT EXISTS tbl_customer {
  customer_id INTEGER PRIMARY KEY,
  customer_name TEXT NOT NULL,
  customer_addr TEXT
};
private final static String CREATE_PRODUCT_TABLE = "CREATE TABLE IF NOT EXISTS tbl_product {
  product_id INTEGER PRIMARY KEY,
  product_name TEXT NOT NULL,
  product_desc TEXT
};
private final static String CREATE_ORDER_TABLE = "CREATE TABLE IF NOT EXISTS tbl_order {
  order_id INTEGER PRIMARY KEY,
  order_customer INTEGER NOT NULL,
  order_date DATETIME DEFAULT current_timestamp,
  FOREIGN KEY(order_customer) REFERENCES tbl_customer(customer_id) ON DELETE CASCADE
};
public void onCreate(SQLiteDatabase db) {
  db.execSQL(CREATE_CUSTOMER_TABLE);
  db.execSQL(CREATE_PRODUCT_TABLE);
  db.execSQL(CREATE_ORDER_TABLE);
  db.execSQL(CREATE_ORDER_PRODUCT_TABLE);
}
```

![Figure 2: Code snippet, creating a database table using SQLite](image)

From these, one can almost immediately see the advantages of using ORM compared to using SQLite. Figure 2 contains many boiler-plate codes that one has to write for SQLite whereas an ORM like Room requires just a few lines of code to achieve same purpose. This, however, comes with some effort which includes a learning curve on how to set the project to use the framework and how to annotate the entities for the framework to understand and manipulate. Some would argue that this is a one-time learning and an overall long-term gain since subsequent use of the library will require less effort. Another advantage of using ORMs is that getting an object from the database requires less code
when using an ORM than SQLite. As an example, inserting a Customer object into the database can be accomplished in SQLite using the following snippet,

```java
public void insertCustomer(final Customer customer) {
    ContentValues contentValues = new ContentValues();
    dbWriter.beginTransaction();
    contentValues.put(COL_CUSTOMER_ID, customer.getId());
    contentValues.put(COL_CUSTOMER_NAME, customer.getName());
    contentValues.put(COL_CUSTOMER_ADDRESS, customer.getAddress());
    dbWriter.insertWithOnConflict(TB_CUSTOMER, null, contentValues, SQLiteDatabase.CONFLICT_REPLACE);
    contentValues.clear();
    dbWriter.setTransactionSuccessful();
    dbWriter.endTransaction();
}
```

**Figure 3: Code snippet, inserting a Customer into the database using SQLite**

From this snippet, one has to extract the values of the Customer object, put them in ContentValues and call the SQLiteDatabase (dbWriter) and then insert. This is done in Room and greenDAO with a few lines of code.

```java
public void insertCustomer(final Customer customer) {
    customerDao.insertCustomer(customer);
}
```

**Figure 4: Code snippet, inserting a Customer into the database using Room**

```java
public void insertCustomer(final Customer customer) {
    daosession.getCustomerDao().insertOrReplaceInTx(customer);
}
```

**Figure 5: Code snippet, inserting a Customer into the database using greenDAO**

Database operations are simplified when using ORMs. However, ORMs can also lead to an increase in application size which in turn can affect the application performance negatively [19]. Another difference between ORMs and SQLite is that SQLite provides no compile-time verification of SQL queries whereas the ORMs provide compile-time verification, limiting the possibility of writing wrongly formed SQL statements.

### 2.3 Database Cascade

An essential part of all relational databases is the ability to propagate changes from an entity to the underlying dependencies. This feature of relational databases is referred to as database cascade or cascading. A typical example of how cascading works is that for a relational system like that used in this project, if a user deletes an Order, the Products in the Order have to be deleted. As an extension, if a Customer object is deleted, then the Orders made by the Customer are deleted and consequently the products. Different database frameworks implement this feature differently. Cascading is enforced in Room using annotation tags. The snippet below shows the relationship between a Customer and an Order and specifies the cascading rule.
The `onUpdate = ForeignKey.CASCADE` above declares the cascading rule between Customer and Order objects. It means that if the `customer_id` value in the customer table (`tb_customer`) is updated, then the value of `order_customer` the order table (`tb_order`) has to be updated.

The version of greenDAO (version 3.2.2) used in this project does not support database cascading rules. If this is desired, the developer must write the code for this. For example if all Orders belonging to a Customer has to be deleted, then when a Customer object is deleted, the developer has to do like this:

[Start of pseudo code]
1. Get the id of the Customer and store in a variable, customer
2. Delete the Customer
3. For each order in the list of Orders
   if order.getCustomerId() == customer
      delete(order)
[End of pseudo code]

The corresponding snippet for SQLite specifying the cascading rule between an Order and Customer is:

```java
private final static String CREATE_ORDER_TABLE = "CREATE TABLE IF NOT EXISTS tb_order (" +
"order_id INTEGER PRIMARY KEY, " +
"order_customer INTEGER NOT NULL," +
"order_date DATETIME DEFAULT current_timestamp," +
"FOREIGN KEY(order_customer) REFERENCES tb_customer(customer_id) " +
"ON DELETE CASCADE ON UPDATE CASCADE);";
```

Then,
```java
@override
public void onCreate(final SQLiteDatabase db) {
    db.execSQL(CREATE_ORDER_TABLE);
}
```
This shows that the column `order_customer` in the order table in the database (`tb_order`) is related to the column `customer_id` in the customer table in the database (`tb_customer`) and that if the value of `customer_id` is updated, then `order_customer` should be updated. The code snippets above shows that database cascading rules are possible to be declared at compile time for Room but not for greenDAO.

2.4 Some important terminology

2.4.1 Mobile Application success

Users are usually the target for most mobile applications. Therefore, the acceptance and perception of the end-user are essential metrics for application success. Users express their satisfaction with an application through star ratings [16]. A higher rating implies a more successful application.

2.4.2 Persistence

Persistence generally refers to “object and process characteristics that continue to exist even after the process that created it ceases or the machine it is running on is powered off” [2]. In simpler terms, it means allowing the application data to remain after the application closes. For information to persist, it often saved in a permanent storage medium such as the hard disk or database. In this thesis, persistence will be limited to data storage using databases.

2.4.3 Application Performance

Software performance is defined as how well the software performs given the system resources. Some of the metrics used to measure performance are listed below.

- **Response time**: The total time lapse between an inquiry on a system and the response to that inquiry or the time-lapse for completing a task from the request [32]. Developers aim at having an application with a low response time as possible.

- **Processor Usage**: The percentage of CPU resources used by the system in completing a software task. The less CPU used by the application, the better the app since the device and OS can do more for the user.

- **Memory Usage**: The amount of RAM used an application while completing a task. An application that uses much RAM to perform a task is generally not desirable because, firstly, the host device might not be able to perform other background tasks in parallel which can lead to a total freeze on some devices with small amount of RAM.

- **Throughput**: The amount of completed work over a specific time [34]. The more work an application does under a specific time, the better the application.

Performance metrics for this project are response time, CPU, and RAM usage.
3 Methodology

This section contains the implementation details of the project. It highlights the tools, used as well as the constrains of the thesis.

3.1 Approach

It is worthy of note that the aim of this project was to measure the performance of two ORM frameworks for Android. Therefore, the design and implementation of the application is restricted to this rather than building a complex UI.

3.1.1 Application versions

Three versions of an Android application containing a relational model was designed and implemented. One version of the application used greenDAO and the other used Room for data persistence. The third version which served as the basis for comparison used SQLite. The SQLite version was considered as a base for comparison because both of the tested ORMs use SQLite under the hood.

3.1.2 Application model

The application model was a simplified version of a customer order system as detailed in chapter 4.1.1. It mimics a real-world situation where a customer can make one or more orders and for each order, there can be more one or more products. This model was chosen as it has most of the features of a relational database, i.e. one-to-one relationships (only one Customer can make an Order), one-to-many relationships (an Order can contain many Products and a Customer can make many orders). The data for the project was generated using another maven-based Java project written by the author of this report [35].

3.1.3 Performance metrics

On the question of time and memory measurements, these were done five times for each app and the average time and memory for the measurements recorded. The number of runs was arbitrarily chosen. However, running the tests five times limits errors due to the cold start of the Java Virtual Machine(JVM). Time taken to perform the CRUD operations was measured and saved to a file on the device storage. Database CRUD operations were chosen because these constitute the fundamental operations that an application will perform. These measurements were done using a helper class called MethodTimer which reports the execution time in nanoseconds as well as the milliseconds and seconds equivalents. MethodTimer also saves the results in a JSON file on the device's external storage, making it easier to extract and analyze the results. It was initially intended to use TimingLogger [42], an in-built method timer included in the Android SDK and reports the results in Android Studio's logcat which could be both error-prone and time-consuming. Therefore, writing the results to file for later processing the motivation for creating MethodTimer (please see Figure 21 in Appendix A). The full implementation of the MethodTimer can be found in the project source repository [43] at
appVersion/io.blongho.github.appVersion.util where appVersion stands for room, greendao or sqlite.

With regards to memory usage, the Android Profiler [37] was started whenever the application starts and the memory usage recorded. Android Profiler is a built-in tool in Android Studio which can give a real-time value for the RAM and CPU usage of an application. It is commonly used by developers for profiling their applications [27].

As regards the application size, Android Package (APK) was generated for each application using Android Studio and the size analyzed as detailed in section 4.3.5.

3.2 Implementation

3.2.1 Functional Requirements

The application should be able to

- Present a UI with buttons containing texts that depict the intended action. This limits ambiguity.
- Display the time taken for initializing the database in Android Studio Logcat so that the developer can assure that there is an instance of the database opened before carrying out any database operation.
- Display the time taken to perform any of database CRUD (create, read, update and delete operations) in Android Studio Logcat so that this can be copied and tabulated for visual appraisal in the final report.
- Save all the measurements in the last two points to a JSON file whose name reflects the application version as well as the number of entries used for the CRUD operations. Saving to file makes it easier to run all tests and then analyze the results at once. The results can also be read at a later time if need be. Being in JSON has the advantage that it is easy for humans to read and write\(^1\) as well as easy to be parsed using any JSON parser e.g. Gson\(^2\). The saved file name contains the number of entries for the CRUD operations in order to distinguish the files in the directory.

3.2.2 Non-functional Requirements

The main non-functional requirements for the project are

- The application should have a minimum target Android SDK of 21. This is needed for the optimal use of the Android Profiler [38].
- The application source code and testing data should be in a public repository so that others can clone and test it for themselves.

\(^1\) [https://www.json.org/](https://www.json.org/)
\(^2\) An open-source lightweight Java serialization/serialization library to convert Java Objects into JSON and back. [https://github.com/google/gson](https://github.com/google/gson)
3.2.3 Development environment

This project was carried out in a computer running Windows 10 Pro which is the only computer accessible to the author of this report. Android Studio version 3.4 [39] was the IDE used for the project. Android Studio was chosen because it is a free platform for Android development. This IDE has many tools needed for debugging Android applications as well as the Android Profiler [37] used in measuring the CPU and RAM usage for this project. Android Studio also offers the possibility to create Android Virtual Devices [40] as well as generating the APK.

3.2.4 Programming language

The programming language used for the project was the Java Standard Edition Development Kit 8 [41]. This language was chosen because the author has a good understanding of it as well as the fact that Java is one of the recommended languages for Android development.

3.2.5 Device used for profiling

The following are the specifications of the AVD used in profiling the applications.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android version (API level)</td>
<td>7(24)</td>
</tr>
<tr>
<td>RAM</td>
<td>4GB</td>
</tr>
<tr>
<td>Heap size</td>
<td>512 MB</td>
</tr>
<tr>
<td>Size, resolution, pixel density</td>
<td>5” 1080×1920</td>
</tr>
<tr>
<td>CPU cores</td>
<td>4</td>
</tr>
<tr>
<td>Internal Storage</td>
<td>8 GB</td>
</tr>
</tbody>
</table>

Table 1: Properties of testing devices

3.2.6 Experimental data

The data for the project was generated with the help of java-faker [42], an open-source Java library for generating fictive data. It is available at bitbucket [35]. The data were stored as JSON files. These were transferred into the app/res/raw directory in the different Android application versions. This data was generated because the author could not find any suitable relational data that fits the model of this project.
4 Construction

This section describes the implementation details as much as possible in a way that it can be replicated. There is no intention in making this section serve as a tutorial as that would mean a repetition of already well-written tutorials and documentations for the various features. However, where necessary, some code snippets and UMLs are shown. The author encourages the reader to follow the documentation links as well as the project source code [43] as they contain all the needed information to replicate this project. During the implementation of the different application versions, their respective documentations were read and where necessary, more information was sought online from blog tutorials and forums.

4.1 Design

The application design was such that all the application versions should be as similar as possible in terms of method calls as well as classes. This was done in order to control for any differences in performance that might result from programming differences. The following was the same across the application versions

- the application models (Figure 6)
- the same UI (Figure 7)
- the same method calls during test (Figure 10)
- same amount of data in the app/raw/ directory (Figure 11)

The only programming difference among the application versions is the database logic. This difference is because each framework has a different way of accessing the database. In addition to these considerations, there was extensive use of threading in order to minimize overloading of the main thread when running the tests. Using threading can help boost the application performance [44].

4.1.1 The Relational model

The UML diagram and relationship between the application models is as shown below. Getters and setters are omitted for brevity.

![UML diagram](image)

*Figure 6: Model UML and database relationships*
This model mimics a real-world store where Customer can make zero or many Orders and each Order contain at least one Product. For full details of the implementation of these models, please see the project repository [43].

4.1.2 User interface

All the applications had the same user interface except for the application name.

![Application user interface](image)

*Figure 7: Application user interface*

The display shown in Figure 7 above is contained in a layout file of the application. It was constructed as follows. The root is a LinearLayout, vertically oriented that matches the parent width in both height and width, followed by a Button whose width matches the parent and height that “wraps the content” (create the database button). This is followed by two nested LinearLayout that have orientation set to horizontal and height set to 100dp. These have three child views, a Button, followed by a View which helps gives the space between the other Button. Following these two LinearLayouts is a Button(Drop all entries). This layout is the only layout file in the project and it is displayed in the MainActivity. The only difference in presentation in the application name as evident in the Figure 7.
4.1.3 Application overview

The various methods and fields contained in the classes of Figure 8 can be seen in (Please see Figure 22 of Appendix A). The MainActivity has an instance of the Test class which has implementations as defined by the interface, TestSuiteInterface. ResultsFileWriter is a utility class that saves the results to file. It is used by MethodTimer. MethodTimer is called for each of the methods that the Test class implements. To perform the data Create operation, the contents are read from the file and returned as a String object. Reading the contents from the file is not measured as this does not constitute part of the objectives. Timing is done when the method for saving the data is called. Version-specific implementations of the CRUD operations are in the package "io.blongho.github.xxx.database" where "xxx" stands for room, sqlite, greenDAO for the different versions of the application.

4.2 Database access

4.2.1 SQLite version

The implementation of this version of the application was done following the documentation of SQLite for Android [7]. The source code is found at the branch “sqlite” in the project's repository [43].

SQLite has been part of the Android SDK since the beginning of Android. Therefore, there were no dependencies included in this version of the application. All one does is extend SQLiteOpenHelper, a helper class that manages database creation and version management. In this project, the SQLiteOpenHelper class's “onCreate”, “onUpgrade”, “onDowngrade” as well as “onOpen” were overridden with logic to perform the required tasks as the names suggest. Performing CRUD operations with SQLite requires considerable effort from the programmer and a good understanding of the structured programming language (SQL) syntax. In this project, all possible database
operations for the project’s objectives were written in a class called DatabaseManager as shown in the UML below.

![DatabaseManager UML Diagram]

**Figure 9: UML, SQLite database access**

### 4.2.2 greenDAO version

This project was based on greenDAO version 3.2.2, the latest release at the time of carrying out this study. Implementation of the greenDAO version was done by following greenDAO’s documentation [45] and the source code is found at the branch “greendao” in the project repository [43]. The appropriate dependencies and [46] the entities annotated following the documentation [57]. After the annotations and the rebuilt (Android Studio, Build>Rebuild project), greenDAO auto-generates the necessary data access objects (DAO) for each entity. Getting an entity’s DAO is achieved by getting an instance of the DaoSession. These were then used to carry out CRUD operations on the entities annotated with the “@Entity” tag. The implementation UML is shown below.

---

3 These can be seen in the project repository at

[build/generated/source/greendao/io/blongho/github/greendao/model]
This project was built using Room version 2.1.0-beta01 which was the latest release when this project was done. The procedure was similar to that of greenDAO, which is the addition of library dependencies and annotation of entities [6]. The package diagram is shown below.
4.3 Benchmark

The project used 1000, 5000, 10000, and 12000 values to test for CRUD execution time. These values were used to see if there is any variation in execution time with respect to the number of entries. These values were stored in the "app/res/raw" directory. The files could also be placed on a remote server but downloading tasks from the internet would require extra permission to be added in the application manifest which would add unnecessary overhead in programming efforts. Internet download also uses more memory and time than getting assets from the device.

4.3.1 Device preparation

The testing devices detailed in section 3.2.5 were created for the sole purpose of this project. No other applications were installed on the devices prior to the tests. This is to limit device memory pressure due to some background tasks running from another application on the device. The devices were all set to flight mode before and during all the tests. Setting the device in flight mode is also a way of limiting device memory pressure due to background update on the Android OS software and services. Device memory pressure can result to killing of low priority processes by the Android Low Memory Killer Daemon\(^4\) \[47\].

\(^4\) A process that monitors the memory state of running processes in the Android OS and kills least essential processes if there is high memory demand in order to keep the system at an acceptable performance level \[47\].
4.3.2 Execution time

Execution times were measured when the application was run on the AVD (please see 3.2.5) MethodTimer was used to measure the execution time for the database CRUD operations. MethodTimer was used to measure the execution time for the database CRUD operations. MethodTimer uses System.nanoTime() which returns the value of the Java Virtual Machine in nanoseconds precision. For each CRUD operation, an instance of MethodTimer is created and its start() method is called just before the timing is required and stopped immediately after the expected code snippet has executed. The execution time for the task is displayed on the AndroidStudio Logcat as well as saved in a JSON file in the format “name_number.json” where name represents the application name and number represents the number of values used during the measurement. For example, deleting all Customer objects from the greenDAO version was accomplished using,

```
// Run potentially long-running actions on background thread

public void delete() {
    ExecutorService threadPool = Executors.newCachedThreadPool().submit() -> {
        final int numberOfCustomers = (int) dbSession.getCustomerDao().count();
        final MethodTimer timer = new MethodTimer("Deleting " + numberOfCustomers + " customers");
        timer.start();
        dbSession.getCustomerDao().deleteAll();
        timer.stop();
        timer.showResults();
        return null;
    }
}
```

Figure 12: Code snippet, example code snippet for measuring execution time

The output in the Android Studio's logcat is

```
24149-24289/io.blongho.github.greendao I/Deleting 5000 customers: 24582800 ns (24 ms, 0 s)
```

The output of Figure 12 in the device in /sdcard/greendao_5_000.json is

```
{
    "method": "Deleting 5000 customers",
    "milliTime": 24,
    "nanoTime": 24582800,
    "secTime": 0
}
```

4.3.3 Measuring CPU usage

CPU measurements were done following the documentation [48]. The measurements are initiated by clicking on the “profile” menu in the bottom menu of Android Studio (assuming the default layout of the IDE). When a CRUD operation was executed, the Android Profiler was monitored until the memory usage of the app reaches about 1% or less. This was done in order to allow for a “fresh” run after the system must have de-allocated the memory that was allocated for the application and action. The peak for each region was noted.
4.3.4 Measuring RAM usage

The RAM was measured in a similar manner to the CPU. From the Android Profiler, reading the RAM usage is done by switching the menu to “MEMORY.” The amount of RAM in megabyte used by the application can be read as this is also measured while the Profiler measures the CPU.

4.3.5 Creating estimating the APK size

For each application, the APK was generated using Android Studio as follows. While the project is opened in Android Studio, click on Build>Build Bundle(s)/APKs>Build APK

Android Studio then builds the APK and a pop-up menu shows like that in Figure 8 below. To view the APK size and associated files, the following was done click Build> Analyze APK> go/to/direction/of/designed/apk/and/select

This gives a detailed outlook of the APK size as shown in the figure below.

Figure 13: Popup menu after application apk generation with Android Studio 3.4
5 Results

The results for the thesis are presented here. The interpretation and discussion is in the discussion chapter.

5.1 Execution times

5.1.1 Insert

The graph below shows the average time in milliseconds taken to insert values into the database by the different frameworks. Smaller means better. The raw data for this table is found in Table 2 in Appendix B.

Figure 14: Average time for inserting values into the database

5.1.2 Read

The graph below shows the average time it takes to read data from the database. Smaller time means better. Please see Table 3 in Appendix B for raw data.

Figure 15: Average time to read values from the database
5.1.3 update

The graph below shows the average time taken to update values into the database. Smaller means better. The raw data for this graph is in Table 4 of Appendix B.

![Average execution time for database update](image)

*Figure 16: Average time to update entries in the database*

5.1.4 Delete

The graph below shows the average time taken to delete all items from the database. Small means better. The data is in Table 5 of Appendix B.

![Avg. Time to execute database delete](image)

*Figure 17: Average time to delete items from the database*
5.2 CPU usage

Below is the average amount of CPU used by the various database frameworks. Please see raw data in Table 6 of Appendix B.

![CPU usage (%)](image)

*Figure 18: CPU usage (%) by the different app versions*

5.3 RAM usage

The graph below shows the amount of RAM used by each database framework Table 7 of Appendix B contains the raw data.

![RAM usage (MB) by different database frameworks](image)

*Figure 19: RAM usage (MB) by different frameworks*
5.4 APK size

The graph below shows the resulting APK from the different application versions.

Figure 20: Application size of the different app versions
6 Discussion

6.1 The results

The results of this study show that the execution time for database insert and read increases with an increasing number of rows for all the frameworks and that greenDAO is overall faster than Room for all database CRUD operations. This result confirms the benchmark by greenrobot [30] though their tests involved non-relational models. It, however, contrasts the results from the study done by Pu J. et al., [20] who showed that SQLite performed better than any of their studied frameworks including greenDAO. Their performance measure was, however, included other metrics such as energy consumption and programming effort. However, though not part of the objectives of this project, the author of this report can confirm that greenDAO is better than SQLite and Room when it comes to programming effort, measured in terms of uncommented lines of code and amount of time needed to set up the system and the logic. Therefore if the production time is important for the developer and the application does a lot of read-write operations to the database, greenDAO comes as the database of choice.

Another interesting result from this study is the amount of time that Room takes to execute both delete and update operations. Compared to greenDAO and SQLite, Room takes considerably long time for carrying out these operations. This could be explained by the fact that greenDAO does not have any explicit mechanism to enforce database cascading rules [49], [50]. The model used in this project was such that an Order must be made by a Customer and that the rows of OrderProduct table contained foreign keys from the Order. Therefore, a delete of a Customer should result in an implicit delete of an Order and eventual delete of a row in the OrderProduct table. Room does this but neither greenDAO nor SQLite does so. The reason why SQLite does not do so is unknown to the author because these cascading rules were declared (Please see project source code [43]). This means that if cascading is necessary, the developer must explicitly provide the logic for deleting dependent tables which unfortunately means more coding for the developer and some copy-paste code.

As concerns the APK size, the results show that there is a 5% increase in APK size when either Room or greenDAO. That there is an increase in APK size as a result of the inclusion of these libraries is not unexpected [19]. This was confirmed in this project as both ORMs involved the inclusion of some dependencies which generated some framework-specific classes when the application is built, an example being the data access objects generated by greenDAO [51].

The results from the CPU usage show that Room uses more CPU than the other frameworks. This can partly be explained by the fact that Room takes longer to execute these operations. The same trend was observed for the RAM usage whose explanation can be the same as for CPU usage. These suggest that application size does indeed increases the memory usage of an application.
6.2 The Implementation

The author chose a relational model consisting of four tables with a total of twelve rows across the database. Other benchmarking studies used a different model. The benchmark by greenrobot uses a non-relational model which showed that greenDAO is fastest in database CRUD operations [30]. The study by Pu J et al., [20] used a similar model together with the DaCapo framework [28] which is used to measure both database CRUD operations as well as power consumption. Power consumption was not part of the objectives of this project and therefore, a simpler model was created. The author, however, developed a sufficiently complicated relational model that can be used for benchmarking any ORM framework. Despite these differences, the results of the study can still contribute to the knowledge of performance analysis. Furthermore, as far as the author is aware, no other previous study has been done that compares Room and greenDAO on relational models. This, therefore makes this work unique as the first of its kind to empirically establish the performance difference between Room and greenDAO on relational models.

6.3 Limitations and proposal for future work

One of the limitations of this study is the number of frameworks studied. It would have been interesting to see how the other ORMs performed compared to Room and greenDAO for the model. Another limitation is that these results could have resulted from programming bias due to the author's knowledge of the framework. However, the author made considerable effort reading the documentation as well as consulted many books and online resources during the implementation phase. The author did not do the measurements on a physical device which could mimic a real-life situation. Using a physical device could, however, be a limitation on its own as each device is unique in terms of the manufacturer as well as the number of applications already installed on the device. Measuring the execution time in a "controlled" environment seemed justifiable.

Future work should seek to understand how ORM frameworks perform with a relational data model. This project provides a good starting point for such a model. The relational data model created as part of this project can be used by others in measuring the performance of different ORMs. It will also be interesting to see how these frameworks perform for databases with less than 1000 rows.

6.4 Ethical considerations

There are no personal identifiable data contained in this project. All data models were generated randomly using an open-source Java library [42]. The author also releases this work in the public domain and readers are free to use any part of the work for any purpose provided they cite the original author as it is done in academic writing.
7 Conclusion

The study shows that greenDAO performed overall best for all database CRUD operations compared to Room and SQLite. These results could, however, be different if Room did not implicitly cascade update and delete actions. Developers should therefore be aware of the fact that greenDAO does not implicitly cascade delete and update operations. On the other hand, Room performs reasonably well for databases with items up to 5000 entries which is more likely to be the case for many real-world applications. Given that the application size is the same for both Room and greenDAO and that Room enforces database cascading rules, the author recommends Room for persisting relational data when cascading is desired. However, if the application uses simple, non-relational data, it is preferable to use greenDAO.
Room vs. greenDAO for Android
A comparative analysis of the performance of Room and greenDAO

Jan 18, 2020

References


Room vs. greenDAO for Android

A comparative analysis of the performance of Room and greenDAO

Jan 18, 2020


[45] M. Junginger, greenDAO is a light & fast ORM solution for Android that maps objects to SQLite databases.: greenrobot/greenDAO. 2019.


Appendix A: Supplementary diagrams

**Figure 21: UML, MethodTimer class diagram**

Measures the execution time of a method and shows same in Android studio logcat as well as save result to file

```
MethodTimer
+FILE_NAME: String = test.json
-method: String
-start: long
-stop: long
-resultsMap: Map<String, Long>
-fileWriter: ResultsFileWriter

+start(): void
+stop(): void
+setTag(tag:String): void
+showResults(): void
```

```
ResultsFileWriter
-method: String
-nanoTime: long
-milliTime: long
-secTime: long
-filenName: String

-createFile(): File
+writeToFile(T:Generic type): void
```
Figure 22: UML, Full program overview
Appendix B: Raw data

In the tables below, R1, 2...5 stands for first, second… fifth test execution

<table>
<thead>
<tr>
<th></th>
<th>1000</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>R4</td>
<td>R5</td>
</tr>
<tr>
<td>SQLite</td>
<td>184</td>
<td>188</td>
<td>248</td>
<td>185</td>
<td>200</td>
<td>201</td>
</tr>
<tr>
<td>Room</td>
<td>168</td>
<td>129</td>
<td>118</td>
<td>100</td>
<td>123</td>
<td>127.6</td>
</tr>
<tr>
<td>greenDAO</td>
<td>134</td>
<td>147</td>
<td>157</td>
<td>166</td>
<td>137</td>
<td>148.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLite</td>
<td>827</td>
<td>837</td>
<td>840</td>
<td>802</td>
<td>833</td>
<td>827.8</td>
</tr>
<tr>
<td>Room</td>
<td>488</td>
<td>438</td>
<td>431</td>
<td>440</td>
<td>395</td>
<td>438.4</td>
</tr>
<tr>
<td>greenDAO</td>
<td>477</td>
<td>482</td>
<td>487</td>
<td>501</td>
<td>498</td>
<td>489</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLite</td>
<td>1608</td>
<td>1602</td>
<td>1934</td>
<td>1565</td>
<td>1606</td>
<td>1663</td>
</tr>
<tr>
<td>Room</td>
<td>955</td>
<td>901</td>
<td>932</td>
<td>905</td>
<td>938</td>
<td>926.2</td>
</tr>
<tr>
<td>greenDAO</td>
<td>1193</td>
<td>894</td>
<td>922</td>
<td>904</td>
<td>933</td>
<td>969.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLite</td>
<td>1965</td>
<td>1832</td>
<td>1907</td>
<td>1911</td>
<td>1891</td>
<td>1901.2</td>
</tr>
<tr>
<td>Room</td>
<td>1086</td>
<td>1070</td>
<td>1018</td>
<td>1007</td>
<td>1104</td>
<td>1057</td>
</tr>
<tr>
<td>greenDAO</td>
<td>1106</td>
<td>1039</td>
<td>1166</td>
<td>1104</td>
<td>1099</td>
<td>1102.8</td>
</tr>
</tbody>
</table>

Table 2: Execution time to insert values into database

<table>
<thead>
<tr>
<th></th>
<th>1000</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>R4</td>
<td>R5</td>
</tr>
<tr>
<td>Room</td>
<td>30</td>
<td>11</td>
<td>20</td>
<td>8</td>
<td>20</td>
<td>17.8</td>
</tr>
<tr>
<td>greenDAO</td>
<td>23</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>2</td>
<td>7.6</td>
</tr>
<tr>
<td>SQLite</td>
<td>11</td>
<td>19</td>
<td>28</td>
<td>11</td>
<td>14</td>
<td>16.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Room</td>
<td>51</td>
<td>41</td>
<td>54</td>
<td>52</td>
<td>61</td>
<td>51.8</td>
</tr>
<tr>
<td>greenDAO</td>
<td>33</td>
<td>21</td>
<td>13</td>
<td>34</td>
<td>11</td>
<td>22.4</td>
</tr>
<tr>
<td>SQLite</td>
<td>84</td>
<td>32</td>
<td>47</td>
<td>54</td>
<td>34</td>
<td>50.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Room</td>
<td>121</td>
<td>105</td>
<td>78</td>
<td>87</td>
<td>103</td>
<td>98.8</td>
</tr>
<tr>
<td>greenDAO</td>
<td>59</td>
<td>35</td>
<td>31</td>
<td>40</td>
<td>54</td>
<td>43.8</td>
</tr>
<tr>
<td>SQLite</td>
<td>126</td>
<td>88</td>
<td>95</td>
<td>86</td>
<td>87</td>
<td>96.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Room</td>
<td>148</td>
<td>110</td>
<td>148</td>
<td>101</td>
<td>120</td>
<td>125.4</td>
</tr>
<tr>
<td>greenDAO</td>
<td>87</td>
<td>56</td>
<td>61</td>
<td>44</td>
<td>41</td>
<td>57.8</td>
</tr>
<tr>
<td>SQLite</td>
<td>150</td>
<td>131</td>
<td>139</td>
<td>122</td>
<td>108</td>
<td>130</td>
</tr>
</tbody>
</table>
## Room vs. greenDAO for Android

A comparative analysis of the performance of Room and greenDAO

Jan 18, 2020

### Table 4: Execution time to update entries in the database

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>227</td>
<td>204</td>
<td>184</td>
<td>288</td>
<td>189</td>
<td>218.4</td>
</tr>
<tr>
<td>Room</td>
<td>371</td>
<td>467</td>
<td>350</td>
<td>363</td>
<td>453</td>
<td>400.8</td>
</tr>
<tr>
<td>greenDAO</td>
<td>155</td>
<td>139</td>
<td>117</td>
<td>117</td>
<td>136</td>
<td>132.8</td>
</tr>
<tr>
<td>SQLite</td>
<td>895</td>
<td>823</td>
<td>865</td>
<td>822</td>
<td>917</td>
<td>864.4</td>
</tr>
<tr>
<td>Room</td>
<td>7026</td>
<td>7216</td>
<td>6558</td>
<td>6937</td>
<td>6912</td>
<td>6929.8</td>
</tr>
<tr>
<td>greenDAO</td>
<td>636</td>
<td>520</td>
<td>488</td>
<td>586</td>
<td>508</td>
<td>547.6</td>
</tr>
<tr>
<td>SQLite</td>
<td>1740</td>
<td>1636</td>
<td>1691</td>
<td>1753</td>
<td>1695</td>
<td>1703</td>
</tr>
<tr>
<td>Room</td>
<td>25821</td>
<td>26175</td>
<td>25811</td>
<td>25855</td>
<td>25363</td>
<td>25805</td>
</tr>
<tr>
<td>greenDAO</td>
<td>1010</td>
<td>978</td>
<td>966</td>
<td>976</td>
<td>970</td>
<td>980</td>
</tr>
<tr>
<td>SQLite</td>
<td>2135</td>
<td>2134</td>
<td>2020</td>
<td>2015</td>
<td>1970</td>
<td>2054.8</td>
</tr>
<tr>
<td>Room</td>
<td>36213</td>
<td>35956</td>
<td>36939</td>
<td>36350</td>
<td>36312</td>
<td>36354</td>
</tr>
<tr>
<td>greenDAO</td>
<td>1190</td>
<td>1171</td>
<td>1240</td>
<td>1541</td>
<td>1135</td>
<td>1255.4</td>
</tr>
</tbody>
</table>

### Table 5: Execution time to delete (all) items from the database

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Room</td>
<td>375</td>
<td>374</td>
<td>360</td>
<td>370</td>
<td>367</td>
<td>369.2</td>
</tr>
<tr>
<td>greenDAO</td>
<td>8</td>
<td>25</td>
<td>11</td>
<td>21</td>
<td>26</td>
<td>18.2</td>
</tr>
<tr>
<td>SQLite</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>12</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>Room</td>
<td>8245</td>
<td>8516</td>
<td>8182</td>
<td>8443</td>
<td>8193</td>
<td>8315.8</td>
</tr>
<tr>
<td>greenDAO</td>
<td>13</td>
<td>35</td>
<td>23</td>
<td>29</td>
<td>33</td>
<td>26.6</td>
</tr>
<tr>
<td>SQLite</td>
<td>26</td>
<td>12</td>
<td>11</td>
<td>27</td>
<td>10</td>
<td>17.2</td>
</tr>
<tr>
<td>Room</td>
<td>33596</td>
<td>33523</td>
<td>34117</td>
<td>32669</td>
<td>32469</td>
<td>33274.8</td>
</tr>
<tr>
<td>greenDAO</td>
<td>26</td>
<td>19</td>
<td>27</td>
<td>21</td>
<td>20</td>
<td>22.6</td>
</tr>
<tr>
<td>SQLite</td>
<td>8</td>
<td>8</td>
<td>22</td>
<td>25</td>
<td>20</td>
<td>16.6</td>
</tr>
<tr>
<td>Room</td>
<td>49700</td>
<td>49864</td>
<td>48314</td>
<td>47121</td>
<td>47200</td>
<td>48439.8</td>
</tr>
<tr>
<td>greenDAO</td>
<td>37</td>
<td>23</td>
<td>41</td>
<td>39</td>
<td>21</td>
<td>32.2</td>
</tr>
<tr>
<td>SQLite</td>
<td>22</td>
<td>22</td>
<td>20</td>
<td>14</td>
<td>14</td>
<td>18.4</td>
</tr>
</tbody>
</table>

Table 4: Execution time to update entries in the database

Table 5: Execution time to delete (all) items from the database
## Room vs. greenDAO for Android

A comparative analysis of the performance of Room and greenDAO

---

### Table 6: CPU usage (%) by different database frameworks

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Insert</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLite</td>
<td>24.6</td>
<td>25</td>
<td>29.9</td>
<td>24.9</td>
<td>25.4</td>
<td>25.96</td>
</tr>
<tr>
<td>Room</td>
<td>29.5</td>
<td>43.6</td>
<td>34.4</td>
<td>40.9</td>
<td>41.9</td>
<td>38.06</td>
</tr>
<tr>
<td>greenDAO</td>
<td>33.9</td>
<td>32</td>
<td>30.5</td>
<td>39.3</td>
<td>34.4</td>
<td>34.02</td>
</tr>
<tr>
<td><strong>Read</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLite</td>
<td>33.3</td>
<td>34.3</td>
<td>29.6</td>
<td>29.7</td>
<td>30</td>
<td>31.38</td>
</tr>
<tr>
<td>Room</td>
<td>52.6</td>
<td>57.2</td>
<td>49.5</td>
<td>57.1</td>
<td>47.5</td>
<td>52.78</td>
</tr>
<tr>
<td>greenDAO</td>
<td>34.2</td>
<td>39.9</td>
<td>33</td>
<td>40.9</td>
<td>43.7</td>
<td>38.34</td>
</tr>
<tr>
<td><strong>Update</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLite</td>
<td>24</td>
<td>27.8</td>
<td>26.9</td>
<td>27.1</td>
<td>29</td>
<td>26.96</td>
</tr>
<tr>
<td>Room</td>
<td>26.8</td>
<td>27.1</td>
<td>35.9</td>
<td>38.5</td>
<td>36.2</td>
<td>32.9</td>
</tr>
<tr>
<td>greenDAO</td>
<td>30</td>
<td>27.8</td>
<td>26.4</td>
<td>28</td>
<td>28.1</td>
<td>28.06</td>
</tr>
<tr>
<td><strong>Delete</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLite</td>
<td>30.2</td>
<td>31.8</td>
<td>38.2</td>
<td>39.7</td>
<td>33.5</td>
<td>34.68</td>
</tr>
<tr>
<td>Room</td>
<td>47.2</td>
<td>34.8</td>
<td>36.6</td>
<td>44</td>
<td>42</td>
<td>40.92</td>
</tr>
<tr>
<td>greenDAO</td>
<td>25.7</td>
<td>27.5</td>
<td>26</td>
<td>28</td>
<td>26.9</td>
<td>26.82</td>
</tr>
</tbody>
</table>

Table 6: CPU usage (%) by different database frameworks

### Table 7: RAM usage (MB) by different frameworks

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Insert</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLite</td>
<td>26</td>
<td>25.4</td>
<td>23.9</td>
<td>29.3</td>
<td>25</td>
<td>25.92</td>
</tr>
<tr>
<td>Room</td>
<td>27.7</td>
<td>28.2</td>
<td>26.3</td>
<td>26.4</td>
<td>29.9</td>
<td>27.7</td>
</tr>
<tr>
<td>greenDAO</td>
<td>34.9</td>
<td>31.4</td>
<td>30.1</td>
<td>30.6</td>
<td>34.5</td>
<td>32.3</td>
</tr>
<tr>
<td><strong>Read</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLite</td>
<td>24</td>
<td>26</td>
<td>28.5</td>
<td>29.9</td>
<td>30.1</td>
<td>27.7</td>
</tr>
<tr>
<td>Room</td>
<td>33.5</td>
<td>34</td>
<td>35.2</td>
<td>36.1</td>
<td>27.8</td>
<td>33.32</td>
</tr>
<tr>
<td>greenDAO</td>
<td>29</td>
<td>27.5</td>
<td>32.9</td>
<td>30</td>
<td>29.7</td>
<td>29.82</td>
</tr>
<tr>
<td><strong>Update</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLite</td>
<td>26</td>
<td>28.2</td>
<td>24.2</td>
<td>24.6</td>
<td>25.3</td>
<td>25.66</td>
</tr>
<tr>
<td>Room</td>
<td>35.8</td>
<td>36.3</td>
<td>42.1</td>
<td>30.8</td>
<td>45.7</td>
<td>38.14</td>
</tr>
<tr>
<td>greenDAO</td>
<td>27.1</td>
<td>27.4</td>
<td>27.3</td>
<td>25.3</td>
<td>25.4</td>
<td>26.5</td>
</tr>
<tr>
<td><strong>Delete</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLite</td>
<td>25.6</td>
<td>22.1</td>
<td>29.8</td>
<td>26.9</td>
<td>24.6</td>
<td>25.8</td>
</tr>
<tr>
<td>Room</td>
<td>47.2</td>
<td>35.8</td>
<td>36.6</td>
<td>39.8</td>
<td>33.5</td>
<td>38.58</td>
</tr>
<tr>
<td>greenDAO</td>
<td>25.9</td>
<td>26.2</td>
<td>25.9</td>
<td>25.8</td>
<td>26</td>
<td>25.96</td>
</tr>
</tbody>
</table>

Table 7: RAM usage (MB) by different frameworks