# An Experimental Study of the Performance, Energy, and Programming Effort Trade-offs of Android Persistence Frameworks

Jing Pu

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Eli Tilevich, Chair
Barbara G. Ryder
Francisco Servant

July 1, 2016
Blacksburg, Virginia

Keywords: Energy Efficiency; Performance; Programming Effort; Orthogonal Persistence; Android;

# An Experimental Study of the Performance, Energy, and Programming Effort Trade-offs of Android Persistence Frameworks

Jing Pu

(ABSTRACT)

One of the fundamental building blocks of a mobile application is the ability to persist program data between different invocations. Referred to as *persistence*, this functionality is commonly implemented by means of persistence frameworks. When choosing a particular framework, Android—the most popular mobile platform—offers a wide variety of options to developers. Unfortunately, the energy, performance, and programming effort trade-offs of these frameworks are poorly understood, leaving the Android developer in the dark trying to select the most appropriate option for their applications.

To address this problem, this thesis reports on the results of the first systematic study of six Android persistence frameworks (i.e., ActiveAndroid, greenDAO, OrmLite, Sugar ORM, Android SQLite, and Realm Java) in their application to and performance with popular benchmarks, such as DaCapo. Having measured and analyzed the energy, performance, and programming effort trade-offs for each framework, we present a set of practical guidelines for the developer to choose between Android persistence frameworks. Our findings can also help the framework developers to optimize their products to meet the desired design objectives.

This thesis is based on research accepted for publication in the proceedings of the IEEE Modeling, Analysis, and Simulation of Computer and Telecommunication Systems Conference (MASCOTS 2016).

This research is supported by the National Science Foundation through the Grant CCF-1116565.

# Dedication

*To My Family For Their Love And Support*

# Acknowledgments

I would like to thank all of those who have contributed to this work.

In particular I am most grateful to Dr. Eli Tilevich, my supervisor, for his help and support which made this thesis possible. I am indebted to him not just for his scientific contribution but also for his motivating words, day after day, and his friendship.

I would like to thank the members of my thesis committee: Drs. Barbara G. Ryder, Francisco Servant and Eli Tilevich for their scientific and emotional support.

I would like to give special thanks to Zheng Song for his help and suggestion with my research project. I wish to express my sincere appreciation to Sharon Kinder-Potter in the office of Computer Science Department for her help and assistance.

My thanks goes to all my colleagues in the Computer Science Department for their help and advice which made my stay at Virginia Polytechnic Institute and State University a most pleasant experience. I would like to thank Newman Library for the job position provided to me and Anne Lawrence, Amanda French, Keith Gilbertson and other colleagues there for their assistance.

Finally, I would like to pay tribute to the constant support of my family, my relatives and my friends, without their love over the many years none of this would have been possible and whose sacrifice I can never repay. I want to thank my lovely mother for always being there. A very special thanks goes to my husband, Sai, for helping and encouraging me all the time.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Persistence, Android Programming, and Energy Efficiency

Any non-trivial application provides the ability to preserve and retrieve user data, both during the application session and across sessions. This ability is called persistence [58], and in object-oriented applications is commonly implemented by means of object-relational mapping (ORM) or object-oriented (OO) frameworks. These frameworks relieve the developer from the necessity to write raw SQL to interact with the underlying database engines and thus streamline the development process.

As mobile devices continue to replace desktops as the primary computing platform, Android is poised to win the mobile platform contest, taking the 82.8% share of the mobile market in 2015 [24] with more than 1.6 million applications developed thus far [26]. However, energy efficiency remains one of the key considerations when developing mobile applications [56, 64]. Consequently, in recent years researchers have focused their efforts on providing Android developers with insights that can be used to improve the energy efficiency of mobile applications. The research literature on the subject includes general program analysis and modeling approaches [51, 85, 47, 43, 81, 53, 62, 76, 77].

Despite all the progress made in understanding the energy impact of programming patterns and constructs, a notable omission in the research literature on the topic is energy behaviors of persistence frameworks. Although an major building block

of mobile applications, persistence has never been systematically studied in this context. Without understanding the energy consumption and performance trade-off of persistence, one cannot gain a comprehensive insight on the overall energy efficiency of modern mobile applications.

## 1.2 Analyzing the Trade-offs of Android Persistence Frameworks

When selecting persistence framework, performance and energy efficiency are the major concerns of mobile developers. Moreover, reducing the developing time and programming difficulty are always wanted by most developers. This thesis reports on the results of a comprehensive study we have conducted to measure and analyze the Performance, Energy, and Programming effort (PEP) trade-offs of popular Android persistence frameworks. To that end, we consider the persistence libraries most widely used in Android applications [71]. In particular, we study five widely used ORM persistence frameworks (ActiveAndroid, greenDAO, OrmLite, Sugar ORM, Android SQLite), and one OO persistence framework (Realm Java) as our experiment targets [25, 22, 1, 8, 21, 27]. These frameworks operate on top of the popular SQLite or Realm database engines.

The overriding goal of our study is to help Android mobile developers decide which persistence framework they should choose to achieve the desired performance/ energy / programming effort balance for the development scenario at hand. Depending on the amount of persistence functionality in a given application, the choice of a persistence framework may dramatically impact the levels of runtime performance and energy consumption. By precisely measuring and thoroughly analyzing the performance/energy/programming effort characteristics of alternative Android persistence frameworks, this study aims at obtaining a deeper understanding of the persistence's impact on the mobile software development ecosystem.

In our experiments, we apply these six persistence frameworks to different benchmarks, and then compare the resulting runtime performance, energy consumption, and programming effort [34]). Our benchmarks include an Android ORM benchmark designed to measure the performance of individual database operations as well as a well-known DaCapo H2 database benchmark [39]. In an effort to understand the noticeable performance and programming effort disparities between different persistence frameworks, we also introduce a recommendation model to help developers

select a suitable persistence framework for their applications.

## 1.3 Research Questions

The specific questions we want to answer in this thesis are:

**RQ1.** How do popular Android persistence frameworks differ in terms of their respective features and capabilities?

**RQ2.** What is the relationship between the persistence framework's features and capabilities and the resulting performance, energy efficiency, and programming effort?

**RQ3.** How do the characteristics of an application's database functionality affect the performance of persistence frameworks?

**RQ4.** Which metrics should be measured to meaningfully assess the appropriateness of a persistence framework for a given mobile application scenario?

## 1.4 Research Findings

To answer **RQ1**, we analyze the documentation and implementation of the persistence frameworks under study to compare and contrast their features and capabilities. We find these Android persistence frameworks have significant differences in database engine supports, programming supports, programming abstraction patterns, relational feature supports, and execution modes.

To answer **RQ2** and **RQ3**, we measure each of the energy, performance, and programming effort metrics separately, compute their correlations, as well as analyze and interpret the results. On **RQ2**, for example, we find that programming abstraction pattern of the persistence framework is a key factor cause the overall performance difference. Programming support of the persistence framework has an impact on programming effort. On **RQ3**, we find that an application's dominant database operations (i.e., insert, update, select and delete), and query language complexity (i.e.,

JOIN query, aggregation or arithmetic operation, expression update, batch operations) affect the overall performance.

On **RQ4**, we use several important metrics to evaluate an application implemented by a specific persistence framework, which includes energy consumption, performance, Uncommented Lines of Code (ULOC), McCabe Cyclomatic Complexity, read/write database operation number and Dalvik VM method invocations. We also introduce a recommendation model and demonstrate its use by applying it to the benchmarks used for the measurements.

## 1.5 Research Contributions

Based on our experiment results, the main contributions of this thesis are as follows:

1 To the best of our knowledge, this is the first study to conduct empirical evaluation on the energy, performance, and programming effort trade-offs of widely used Android persistence frameworks.

2 Based on our experimental results, we offer a series of guidelines for Android mobile developers to select the most appropriate persistence framework for their mobile applications, and also to optimize their products for the mobile market. For example, ActiveAndroid or OrmLite fit well for applications processing relatively large data volumes in a read-write fashion. Our Android ORM experiments are based on the basic persist operation type—insert, select, update, or delete, whose findings can be applied to general database relevant applications. Our DaCapo experiments's conclusion can be applied to select-and-update operation dominant applications.

3 We introduce a recommendation model that take into account programming effort in addition to performance and energy efficiency, which is applied to our experimental benchmarks quantifying the trade-off between these three measurement dimensions for each persistence framework. This model provides a new means to evaluate the fitness of a persistence framework for a given mobile application scenario.

## 1.6 Thesis Roadmap

The rest of this thesis is organized as follows. Chapter 2 gives the definition of terms. Chapter 3 introduces the motivation of this study. Chapter 4 provides the background and related work information for this research. Chapter 5 introduces the history of Android persistence framework, as well as compares the features and capabilities of the studied persistence frameworks. Chapter 6 describes the design of our experimental study. Chapter 7 presents the study results, listing our findings and offering guidelines for Android developers. Chapter 8 presents our recommendation model. Chapter 9 discusses the threats to internal and external validity of our experimental results. Chapter 10 presents future work direction. Chapter 11 summarizes our conclusions.

# Chapter 2

# Terminology

This chapter defines the key technical terms used in this thesis.

**Relational Database:**
A relational database [44] is based on the relational model, which organizes data in terms of tuples and group tuples into relations. Relational Database systems use the SQL language for data manipulation. SQLite is a relational database engine for embedded devices, commonly used in Android mobile applications.

**Object Database:**
An object database [41] is based on the object-oriented model, which is designed to comply with the characteristics of object-oriented programming languages. It can be viewed as an extension to the application environment, the object nodes are maintained in memory pages. Realm is an embedded object database engine.

**Object-relational impedance mismatch:**
The differences between the object-oriented and relational models have been known as the object-relational impedance mismatch [54].

**Persistence framework:**
A persistence framework serves as a middleware layer that bridges the application logic with the database engines operations, automating the process of storing program data in a database.

**Data access object:**
A data access object (DAO) [40] is a design pattern used in some persistence frameworks (such as greenDAO, OrmLite) which provides public persistence operation interfaces without exposing database operation details (such as complex Java Database Connectivity (JDBC) code and non-portable SQL).

**Object-relational mapping (ORM) persistence framework:**
Object-relational mapping (ORM) [29] is a frequently-used solution for object-relational impedance mismatch, an incompatiblity between the programming models imposed by an object-oriented language and a relational database. The former uses object entities and attributes to fulfill data management tasks, while the latter organizes the data within tables and fields. An ORM persistence framework solves the mismatch problems including granularity mismatch, inheritance mismatch, identity mismatch, associations mismatch, and navigation mismatch [20]. Hibernate, Apache OpenJPA, greenDAO, OrmLite, Sugar ORM, Android SQLite, etc. are ORM persistence frameworks designed for JAVA PC or Android applications.

**Object-oriented (OO) persistence framework:**
An object-oriented persistence framework uses the same object model as the underlying object database. In the scope of this thesis, Realm Java is the only Android OO persistence framework we study, which works with Realm database.

**Table:**
In a relational database, a table represents an entity, which consists of a set of tuples (also known as rows) and attributes (also known as columns).

**Entity:**
An entity is a persistence domain object defined by object-oriented programming language in user application, used to map a corresponding database table, and a member variable corresponds to a table attribute, while each entity instance represents a tuple in the table.

**Database schema:**
In a relational database, a schema descriptively defines the structure, constraints of the tables, and the relationships among the tables.

**Performance:**
In this thesis, we use performance to refer to the time consumed by running a particular application or benchmark.

**Energy:**
Energy is defined as the electric charge that a battery will hold and the time a device will be allowed to run before the battery needs recharging.

**Programming effort:**
Programming effort refers to the amount of effort required to complete a specific programming task, always related to software complexity. A series of metrics depending on program size, control structure or characteristics of module interfaces can be used to estimate the programming effort, including Uncommented Lines of Code (ULOC), McCabe Cyclomatic Complexity [68], Coupling Between Object classes [65], Number of Static Methods, Number of Parameters, Number of Attributes, Nested Block Depth, etc.

**Benchmark:**
Benchmark is a standardized program used to evaluate a persistence framework's energy consumption and performance. In our study, we have two benchmarks: Android ORM benchmark and DaCapo benchmark.

# Chapter 3

# Motivation

Nowadays, developing a robust application without using any middleware is risky and high cost. This thesis focuses on Android persistence frameworks as its object of study. Because relatively few prior studies have concerned Android persistence frameworks, the ones dealing with the differences between the persistence mechanisms or persistence frameworks in Java desktop environments are particularly worthy of attention.

The designers of the Android platform have recognized the importance of persistence by including the SQLite database module in the standard Android image from 1.5 version. Ever since, this module has been used widely in Android applications [60]. According to our analysis of the most popular 550 applications hosted on GooglePlay, 400 of them (73%) involve interactions with the SQLite module. Android persistence frameworks have been introduced and refined to facilitate the creation of database-oriented applications.

The prior persistence studies have mainly focused on the comparison of time consumption. However, as the energy demands of mobile applications continue to exceed the battery capacities of mobile devices, one cannot neglect energy efficiency as a key concern when analyzing the overall performance of persistence frameworks on mobile platforms.

One major goal of this thesis is to understand the energy consumption of Android application when using different persistence frameworks. Studies have shown that a series of software development related factors can significantly influence the energy consumption of a software system [86] (e.g., design patterns involved, the Model-

View-Controller architecture [45], information hiding, implementation of persistence layers, code obfuscation [19], refactoring, and data structure usage).

# Chapter 4

# Related Work

To set the context for the research presented in this thesis, this chapter first gives an overview of the main technologies used and then discusses the related state of the art. The major sections include the discussion of prior studies on persistence framework, performance and energy efficiency, and programming effort which are related to our study. This chapter also introduces the history of persistence framework, and then compares major Android persistence functionality, as it is realized by means of embedded database engines and Android persistence frameworks in our study.

This thesis evaluates the programming effort required to implement the benchmarks using different Android persistence frameworks. The results of this evaluation can be helpful in estimating how the choice of a particular persistence solution can affect programmer productivity.

## 4.1 Persistence Framework

Jordan [57] compares and contrasts major Java persistence mechanisms, including Java Object Serialization (JOS), JavaBeans Persistence (JBP), Orthogonal Persistence (OPJ), JDBC, Java Data Objects (JDO), and Enterprise JavaBeans (EJB). He also proposes a set of criteria, including performance, scalability, reusability, transaction support, and operational complexity, for evaluating the persistence mechanisms. His work aims at evaluating the native persistence support provided by the Java language rather than that provided by external third-party persistence frameworks.

Phutela [75] discusses the pros (e.g., transparent object-oriented mapping mechanism, less programming effort, powerful customized query language, and performance optimization) and cons (e.g., learning cost, overhead caused by framework and data complexity) of the ORM framework Hibernate, comparing it with JDBC, which is a naive Java database connectivity API. This work studies the available ORM features without evaluating their performance or programmability characteristics. Bhatti, et. al. [38] compare the performance of three open source Java ORM tools—Hibernate, Ebean and TopLinkhave—using a simple benchmark with the insert/query/update/delete database operations, similar to our Android ORM benchmark design. As compared to our benchmarks, however, this work leaves out the scalability and complexity of database structure considerations.

## 4.2 Performance and Energy Efficiency

Here we list the primary branches in Android energy research area from hardware to software, and then emphatically introduce the prior research efforts on the Android application-layer energy consumption analysis which are most related to our research category.

Studies [89, 59, 55] focus on estimating the energy consumption of mobile hardware components, such as CPU, LCD, GPS, Wi-Fi, audio, etc. These approaches manually or automatically model the power consumption of different architectures to understand the run-time behavior of energy-intensive hardware components. In recent years, the research focus has shifted more toward reducing the energy consumption of mobile applications, which has become a critical problem in modern software design.

Some early research focuses on the instruction level power estimation techniques [84, 80, 69], which require a complete instruction level trace analysis at runtime. These estimation models always involve large computing overhead, which can commonly lead to considerable measurement inaccuracies.

Studies [37, 82, 47] use system energy modeling methods to estimate the energy consumption at subroutine or system-call levels. Their estimation results are impacted by the accuracy, rate and overhead of collecting the measurements at runtime, and also depend on the selection and adaptability of the regression model in place.

Some researchers aim at understanding the application-level energy consumption. Geoffrey, et. al. [51] combines program analysis (which tracks run-time traversing

path and energy-related information) and pre-instruction energy modeling (which provides individual energy estimation function for each distinct hardware component) techniques to estimate the overall energy consumption of an application. Studies [81, 53, 62, 76, 77] discuss the impact of software design patterns on the application energy consumption.

Other studies are devoted to energy-aware programming. One representative approach introduces the Energy Types (ET) model, a programming abstraction that can be used to build energy-efficient applications [43]. This model helps build the energy management strategies by instrumenting the phase and mode information into a programming language directly. This work also evaluates the validity of ET model by measuring the performance and energy consumption of ET applications.

Compared to these prior works, although our studies also focus on application energy consumption, our emphasis is on the energy impact of the middleware layer. We investigate the relationship between the features of persistence frameworks and the overall application energy consumption, runtime performance, and programming effort.

## 4.3 Programming Effort

Various metrics have been proposed in prior studies. Halstead [50] proposes a method which calculates software metrics such as program vocabulary, length, difficulty and effort by identifying measurable properties (number of operators and operands) and their mutual relationships in a given software application. McCabe et al. [68, 72, 49] introduce the McCabes cyclomatic complexity metric which measures the cyclomatic number–the maximum number of linearly independent circuits in a program control graph. Oviedo [74] presents a software complexity indicator which combines control-flow metric and data-flow metric together; it considers the number of edges in the control flow graph, and the number of variables referenced but not defined in the program. Studies [61, 88] compares several complexity metrics, which include McCabes Cyclomatic Complexity metric, Halsteads programming effort, statement count and Oviedos data flow complexity. Studies [66, 48, 35, 52, 42] explore metrics for object-oriented software. These metrics take into account object-oriented language features, such as encapsulation, inheritance, coupling and polymorphism. Compared to the related prior studies, this thesis estimates the programming effort required by Android Persistence frameworks using two program metrics, such as Uncommented Lines of Code and the McCabe Cyclomatic Complexity.

# Chapter 5

# Android Persistence Frameworks

## 5.1 The Evolution of Persistence Frameworks

Android persistence frameworks are rooted in Java persistence. The concept of Java persistence was first introduced in the specification of the transient Java keyword [57]. Java began to support persistence by providing different persistence mechanisms. The most famous ones are JDBC and Enterprise JavaBeans (EJB) [33], which use entity beans containing database manipulation logic to manage the relational data in applications. Persistence frameworks were introduced later to provide lightweight persistence solutions rather than heavyweight entity beans. A persistence framework usually focuses itself on two aspects: 1) It provides a set of APIs for accessing, persisting, and managing data. It separates the application business needs and the underlying database accesses. 2) It specifies a special query syntax that replaces the relational SQL query syntax.

Hibernate [36] is a full-featured ORM persistence framework, invented by Gavin King in 2001. The Hibernate ORM 5.1.0.Final version [10], launched in February 2016, continues to play a significant role in the promotion and evolution of Java persistence. It not only facilitates the development of subsequent persistence frameworks, such as Apache OpenJPA [46] which is implemented based on Java persistence API specification, and it also acts as a cornerstone for the development of Android persistence frameworks.

## 5.2 Android Persistence Framework Overview

In this study, we use six popular Android persistence frameworks, we will introduce them in turn.

**Android SQLite:**
The Android platform uses SQLite as its default database engine and has a built-in SQLite database management framework in its SDK. This framework handles the maintenance of the active SQLite database connection pool, concurrency protection, and makes a simple encapsulation over the underlying database access interfaces. It provides public user interfaces with relational SQL statement style in nature instead of focusing on object-oriented characteristics.

**ActiveAndroid:**
It is an ORM framework whose distinguishing characteristic is its use of the active record pattern [70], which maps entity classes to database tables, defines properties that correspond to database table columns, and provides database manipulating object interfaces, such as insert, update, and delete. With ActiveAndroid, developers can save and retrieve SQLite database records in one table without considering setup configurations or writing a single SQL statement if their database model is simple.

**greenDAO:**
greenDAO provides an automatic generator tool to map relational entities and their DAOs (Data Access Object), as well as object oriented programming interfaces. The framework uses the DAO layer to encapsulate various operations, such as insert, update, delete, query builder, load, refresh and batch operations in DAO abstraction. Developers can manipulate table entities and persist them via the corresponding DAO. This framework offers some advanced ORM features like session cache, eager loading, and active entities besides basic requirements.

**OrmLite:**
Object Relational Mapping Lite (OrmLite) framework isolates the database operations in abstract DAO classes; it supports compiled statements for reusability, and provides transaction management for atomicity and also for increasing the speed of batch operations.

**Sugar ORM:**
This ORM framework also uses active record style APIs. It provides a simple way to map object operations to SQLite database operations.

**Java Realm:**
Unlike the ORM frameworks discussed above, which are developed on top on SQLite database engine, Java Realm framework is published with the Realm embedded cross-platform database engine, constructed completely using an Object-Oriented model [79] instead of Relational model. This framework along with the Realm engine handle object relations directly, while trying to avoid the complexity of object-relational mapping and achieve maximum zero-copy and minimum deserialization within operations, while also being ACID-compliant. When developers access the entity properties, they in fact access the raw data in the Realm database without extra mapping or frequent memory copying.

## 5.3 Android Persistence Framework Feature Comparison

Our studied Android persistence frameworks differ in a variety of ways. We focus on their feature supports, as they may impact energy consumption, performance, and programming effort. These features include database engines, programming support (e.g., object and schema auto generation), programming abstractions (e.g., data access method support, relationships, raw query interfaces, batch operations, complex updates, and aggregation operations), relational features support (e.g., key/index structure, SQL join operations, etc.), and execution modes (e.g., transactions and caching). Table 5.1 compares these differences of the persistence frameworks used in our study.

We study the following features:

1 **Database Engine:**
Database engine can be "SQLite" or "Realm". Five of the studied persistence frameworks use SQLite [73], an ACID (Atomic, Consistent, Isolated, and Durable) and SQL standard-compliant relational database engine. The remaining framework uses Realm [22], an object-oriented database engine whose design goal is to provide functionality equivalent to relational engines.

2 **Object Code Generation:**
Object Code Generation can be "With code generator" or "Without code generator". Some frameworks feature code generators that relieve the mobile developer from the necessity to write Java classes that represent the relational schema in place.

3 **Schema Generation:**
Schema Generation can be "Manual" or "Auto". At the initialization stage, persistence frameworks employ different strategies to generate the database schema. Android SQLite requires raw SQL statements to create database tables, while OrmLite provides a special API call. greenDAO generates a special DAO class that includes the schema. The remaining frameworks automatically extract the table schema from the programmer defined entity classes.

4 **Data Access Method:**
Data Access Method can be "DAO", "Relational" or "Hybrid". DAO (Data Access Object) is a well-known abstraction strategy for database access that provides a unified object-oriented, entity-based persistence operation set (insert, update, delete, query, etc.). greenDAO, Sugar ORM and OrmLite provide the DAO layer, while Android SQLite adopts a relational rather than DAO database manipulating interface. ActiveAndroid and Realm Java provide a hybrid strategy—both DAO and SQL builder APIs.

5 **Relationship Support:**
Relationship Support can be "One-To-One", "One-To-Many" or "Many-to-Many". greenDAO and Realm Java support all three relationships. ActiveAndroid lacks support for Many-to-Many, while Sugar ORM only supports One-To-Many. Android SQLite and OrmLite lack support for relationships, requiring the programmer to write explicit SQL join operations.

6 **Raw Query Interface Support:**
Raw Query Interface Support can be "Yes", or "No". Raw queries use naive SQL statements, thus deviating from pure object-orientation to execute complex database operations on multiple tables, nesting queries and aggregation functions. Android SQLite, greenDAO and OrmLite all provide this functionality.

7 **Batch Operations:**
Batch Operations can be "Batch Insert", "Batch Update" or "Batch Delete". Batch operations commit several same-type database changes at once, thus improving performance. greenDAO, OrmLite and Sugar ORM provides batch mech-

anisms for insert, update and delete. Realm Java provides batch inserts only, and the remaining two frameworks lack this functionality.

8 **Complex Update Support:**
Complex Update Support can be "Relational", or "Object". Typically, there are two kinds of database update operations—update columns to given values, or update columns based on arithmetic expressions. Android SQLite and ActiveAndroid can only use raw SQL manipulation interface to support expression updates. greenDAO, Sugar ORM and Java Realm support complex updates via entity field modification. OrmLite is the only framework which provides both the value update and expression update abstractions.

9 **Aggregation Support:**
Aggregation Support can be "All", or "Partial". Aggregating data in a relational database enables the statistical analysis over a set of records. Different frameworks selectively implement aggregation functionality. Android SQLite and OrmLite support all of the aggregation functions via a raw SQL interface. Realm Java and Sugar ORM provide an aggregation subset in the entity layer. ActiveAndroid and greenDAO support only the COUNT aggregation.

10 **Key/Index Structure:**
Key/Index Structure can be "Primary Key", "Index" or "Foreign Key". Key/Index structure identifies individual records, indicates table correlations, and increases the execution speed. Android SQLite fully supports the database constraints—single or multiple primary keys (PK), index and foreign key (FK). ActiveAndroid supports integer single PK, unique, index and FK. greenDAO supports integer single PK, unique, index. OrmLite supports single PK, index, FK. Sugar ORM supports integer single PK, unique. Realm Java supports string or Integer single PK, index.

11 **SQL JOIN Support:**
SQL JOIN Support can be "Yes", or "No". SQL JOIN clause combines data from two or more relational tables. Android SQLite only supports raw JOIN SQL. ActiveAndroid incorporates JOIN in its object query interface. DAOs of greenDAO and OrmLite provide the JOIN operation. Sugar ORM and Realm Java lack this support.

12 **Transaction Support:**
Transaction Support can be "Yes", or "No". Transactions perform a sequence

of operations as a single logical execution unit. All the studied engines with the exception of greenDAO and Sugar ORM provide full transactional support.

13 **Cache Support:**
Cache Support can be "Yes", or "No". Some persistence frameworks provide this advanced feature to maintain persisted entities in memory for sped-up future accesses, at the cost of extra processing required to initialize the cache pool. OrmLite, ActiveAndroid, greenDAO support caching.

Table 5.1: Persistence framework feature comparison

| **Features** | **Android SQLite** | **Active Android** | **greenDAO** | **OrmLite** | **Sugar ORM** | **Java Realm** |
|---|---|---|---|---|---|---|
| **Database Engine** | SQLite | SQLite | SQLite | SQLite | SQLite | Realm |
| **Object Code Generation** | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| **Schema Generation** | Manual | Auto | Auto | Manual | Auto | Auto |
| **Data Access Method** | Relational | Hybrid | DAO | DAO | DAO | Hybrid |
| **Relationship Support** | ✗ | One-To-One, One-To-Many | One-To-One, One-To-Many, Many-to-Many | ✗ | One-To-Many | One-To-One, One-To-Many, Many-to-Many |
| **Raw Query Interface Support** | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ |
| **Batch Operations** | ✗ | ✗ | batch insert, batch update, batch delete | batch insert, batch update, batch delete | batch insert, batch update, batch delete | batch insert |
| **Complex Update Support** | Relational | Relational | Object | Relational | Object | Object |
| **Aggregation Support** | All | COUNT | COUNT | All | COUNT, FIRST, LAST | MAX, MIN, SUM, AVERAGE, FIRST, LAST |
| **Key/Index Structure** | primary key, index, foreign key | int single primary key, unique, index, foreign key | int single primary key, unique, index | single primary key, index, foreign key | int single primary key, unique | string or int single primary key, index |
| **SQL JOIN Support** | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ |
| **Transaction Support** | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ |
| **Caching Support** | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ |

# Chapter 6

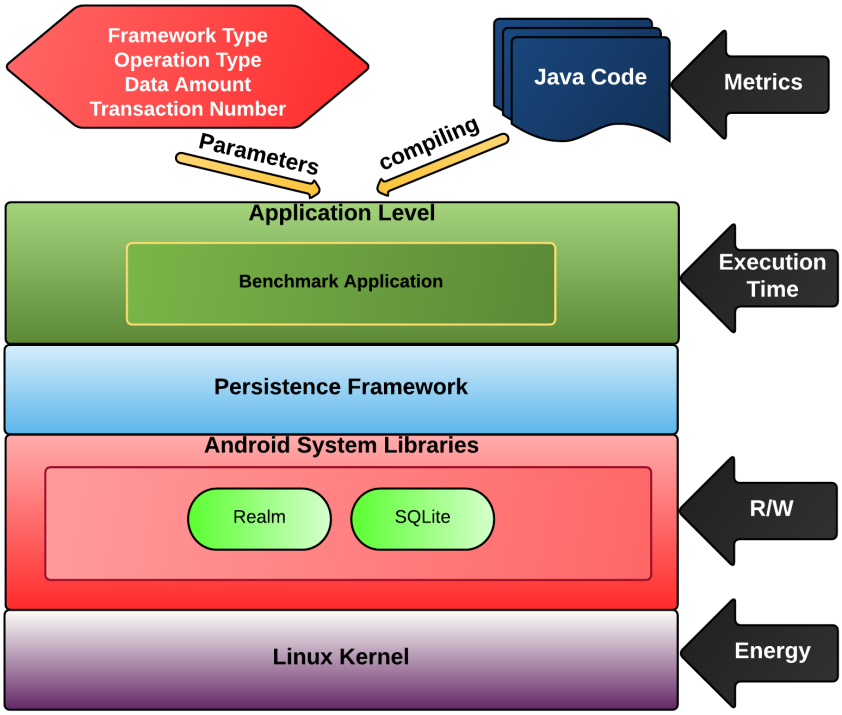
# Experiment Design

In this chapter, we explain the main design decisions we have made in designing our experiments. In particular, we discuss the benchmarks, the measurement variables, and the experimental parameters.

## 6.1 Android Persistence Framework Experiment Architecture

Figure. 6.1 shows the architecture of our Android persistence framework experiment. As we can see in an Android system [87], persistence framework layer locates in the middleware layer, which is between Android system libraries and application level. We executed our benchmarks over six persistence frameworks. We have a set of benchmark input parameters such as framework type, operation type, data amount and transaction number. We measure application energy consumption by attaching device to a power monitor, calculate the number of underlying database read/write operations by hooking SQLite library's operation interfaces, use power monitor and logs to get the execution time of each benchmark run, and also use the Java code metrics to estimate the programming effort of each benchmark.

Figure 6.1: Android Persistence Framework Experiment Architecture




## 6.2 Android Persistence Framework Selection

In this thesis, we choose and evaluate six Android persistence frameworks: Android SQLite, ActiveAndroid, greenDAO, OrmLite, Sugar ORM, and Realm Java as our research subjects based on the following considerations:

1 All of them are designed for the Android platform, backed up by the SQLite and Realm database engines, which are customized for mobile devices, with limited resources, including battery power, memory, and processor. The database engine maintains a schema in memory or on disk, and the Android persistence framework provides a programming interface for the application to interact with the database engine.

2 Android SQLite is the official persistence framework (also known as the JDBC

driver) for Android and its natively supported SQLite database engine. Java Realm is the only Object-oriented persistence framework for Realm database engine. ActiveAndroid, greenDAO, OrmLite, and Sugar ORM are commonly acknowledged as some of the best third-party Android ORM [29] libraries for working with the Android SQLite database engine.

3 These six persistence frameworks have some common strengths, when compared to other open-source Android persistence frameworks.

   (a) They are widely used in Android applications, and provide stable and optimal functionality.

   (b) All of them have stable versions published, have detailed documentation, and also have good community support.

   (c) All of them provide parameterized statements to prevent SQL injection security problems, especially, greenDAO and Java Realm support additional database encryption mechanisms.

   (d) In addition, most of them are actively maintained. Specifically, Android SQLite is upgraded and published with other Android APIs. ActiveAndroid was first released in 2010 with Apache Version 2.0, and the last update was in 2014. greenDAO published the first version in 2011, and released the latest version in 2016 with Apache Version 2.0. OrmLite was invented in 2010, the latest version was released in 2013 with ISC License, and its GitHub repository is still keeping active in 2016. Sugar ORM released its first version in 2012 with Apache Version 2.0, which is still regularly maintained up to 2016. Realm Java was released in 2014 and had the latest 1.0.0 version published in 2016.

Next,

## 6.3 Benchmark Selection

DaCapo H2 [39] is a well-known Java database benchmark that interacts with the H2 Database Engine via JDBC. To adapt this benchmark for our experiments, we replace H2 with SQLite or Realm. This benchmark manipulates a considerable volume of data to emulate bank transactions. The benchmark includes: 1) a complex

schema and non-trivial functionality, obtained from a real-world production environment. The database structure is complex (12 tables, with 120 table columns and 11 relationships between tables), while the database operations simulate the running of heavy-workload database-oriented applications; 2) complex database operations that require: multiple-table query, aggregations, and combined operations.

However, using the DaCapo benchmark alone would leave unanswered the questions of the performance of persistence frameworks under the low data volumes with simple schema conditions. To establish a baseline for our evaluation, we thus designed a set of micro benchmarks, referred to as *the Android ORM Benchmark*, which features a simple database schema with few data records. Specifically, this benchmark's database structure includes 2 tables comprising 11 table columns, and a varying small number of data records. Besides, this Android ORM benchmark comprises the fundamental database operation invocations "create table", "insert", "delete", "select", "update". As the database operations in many mobile applications tend to be rather simple, the Android ORM benchmark's results present valuable insights for application developers.

Note that database operations differ from database operation invocations. The invocations refer to calling the interfaces provided by the persistence framework (e.g., "insert", "select", "update" and "delete"). However, each invocation can result in multiple database operations (e.g., `android...SQLiteStatement.executeInsert()`).

## 6.4 Parameters and Dependent Variables

Our experimental setup comprises a mobile application that uses each of the benchmarked frameworks to execute both DaCapo and the Android ORM Benchmark[1]. Our experiments leave all settings fixed while varying the persistence frameworks.

Next, we explain the variables used to evaluate the performance, energy consumption, and programming effort of the studied persistence frameworks. We also describe how these variables are obtained.

[1]All the code used in our experiments can be downloaded from `https://github.com/AmberPoo1/PEPBench`.

### 6.4.1 Performance

- **Overall Execution Time:** is the time elapsed from the point when a database transaction is triggered to the point when it stops.
- **Read/Write Database Operation Number:** is obtained by hooking into the SQLite operation interfaces provided by the Android System Library. We focus on comparing the Read/Write numbers only on SQLite-based frameworks ((i.e., ActiveAndroid, greenDAO, OrmLite and Sugar ORM). The write operations include executing SQLs that are used to "insert", "delete" and "update", while the read operations include only "select". When performing the same combination of transactions, the differences in Read/Write number is the output of how different persistence frameworks interpret database operation invocations. The read/write ratio can also impact the energy consumption.

  We hook all SQLite operation interfaces. For those interfaces provided for a certain type of database operation, we mark them as "Read" or "Write"; for those interfaces provided for general SQL execution, we search for certain keywords (e.g., insert, update, select and delete), in the SQL strings, and further mark them as "Read" or "Write".

### 6.4.2 Energy

Energy Consumption is obtained by monitoring the real-time current of the Android device's battery.

Equation 6.1 is used to calculate the overall energy consumption [31], where $\bar{I}$ is the average current ($mA$), and $t$ is the time window ($ms$). Energy consumption can be measured by Micro-ampere-hour [6, 9], which indicates the amount of energy charge in a battery that will allow one micro ampere of current to flow for one hour.

$$E = \frac{\bar{I} * t}{3600s/hour} \tag{6.1}$$

The equation shows that the energy consumption is proportional to the execution time, as well as to the current required by the device's hardware (e.g., CPU, memory, network, and hard disk). For the persistence frameworks, the differences of energy consumption reflect not only the execution time differences, but also the dif-

ferent CPU workload and hard disk R/W operations required to process database operations.

### 6.4.3 Programming Effort

- **Uncommented Line of Codes (ULOC):** reflects the programming effort in terms of how much lines of code the programmer has to write to use each persistence framework.
- **The McCabe Cyclomatic Complexity:** estimates the programming effort expended to implement each benchmark using different persistence frameworks. This metric calculates the number of flows through a piece of code based on the control flow graph. The McCabe numbers indicate the degree of code understanding difficulty and probability of defect containment.

### 6.4.4 Benchmark Input Parameters

Next, we introduce the input parameters for different benchmarks. For the DaCapo benchmark, we want to explore the performance boundary of different persistence frameworks under a heavy workload. Therefore, we vary the amount of total transactions to a large scale, and record the overall time taken and energy consumed.

For the Android ORM benchmark, we study the "initialize", "insert", "select", "update" and "delete" invocations in turn. We change the number of transactions for the last four invocations, so for the "select", "update" and "delete" invocations, the amount of data records also changes. Therefore, the input parameters for the Android ORM benchmark is a set of two parameters, {NUMBER OF TRANSACTIONS, AMOUNT OF DATA RECORDS}.

## 6.5 Experimental Hardware and Tools

All the measurements are conducted on an LG LS740 smartphone, with 1GB of RAM, 8GB of ROM and 1.2GHz quad-core Qualcomm Snapdragon 400 processor [83], running Android 4.4.2 KitKat operating system. The device has a 3000mAh removable Lithium Ion battery.

All experiments are executed as the only load on the device's OS. We run each benchmark 5 times within the same environment, with the first two runs to warm up the system, and the reported data as the average of the last 3 runs. We clear cached data before each run.

We use the Monsoon Power Monitor [17] to monitor the current and voltage of the device battery, as shown in Figure 6.2. Figure 6.3 shows the software user interface connects to Monsoon Power Monitor, which provides instant and average informations such as energy consumption, sampling time, power, current and voltage.

Figure 6.2: Monsoon Mobile Device Power Monitor's main channel measurement connection

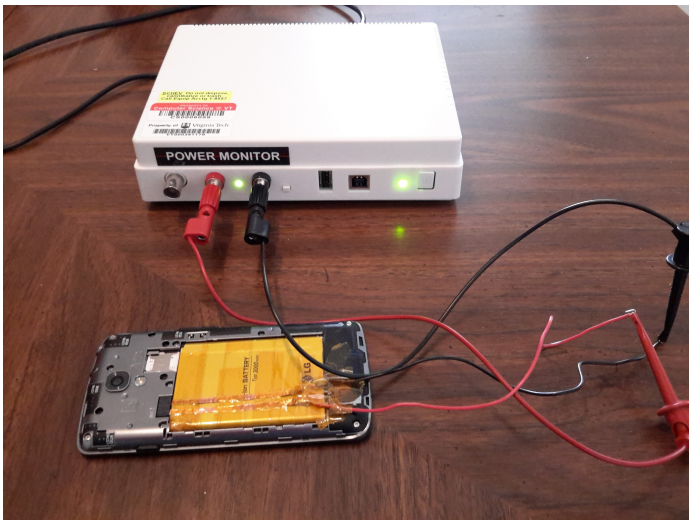
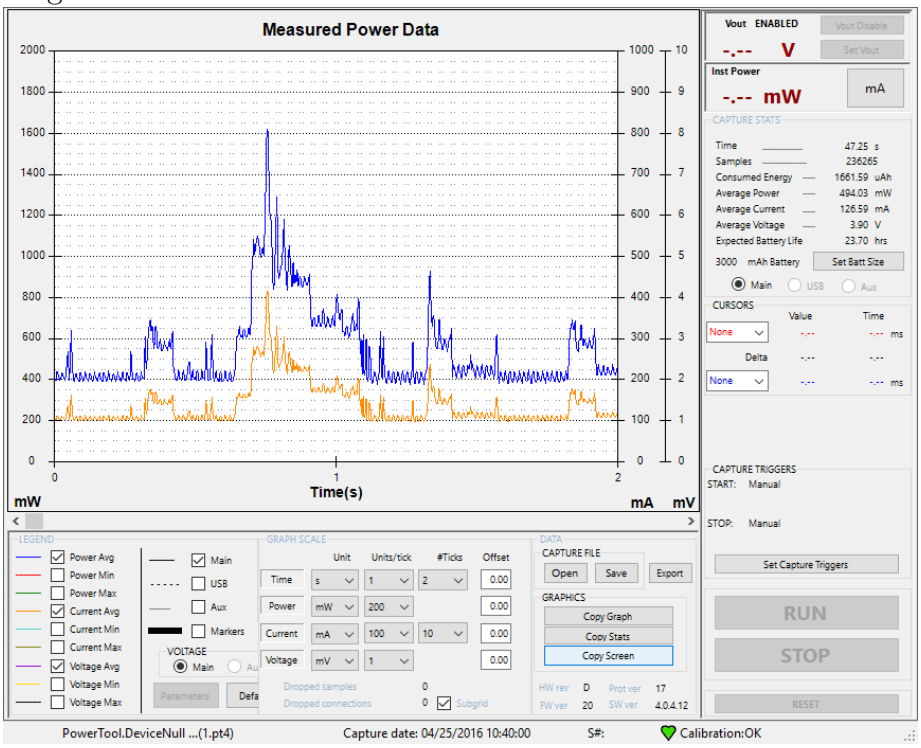


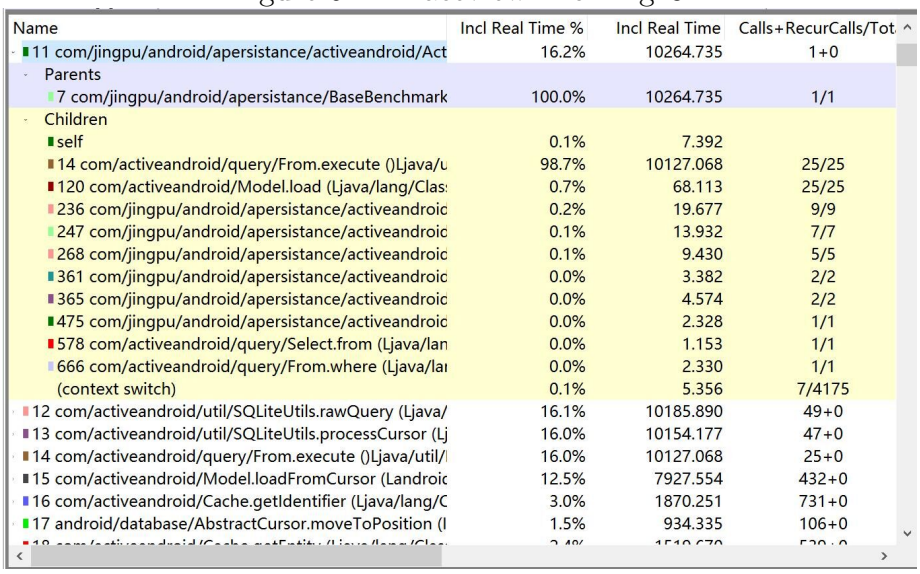
Figure 6.3: Monsoon Mobile Device Power Monitor's Software UI




We use Xposed framework [32] instrument our benchmark code to compare the underlying database calls (database read and write) between five SQLite-based persistence frameworks—ActiveAndroid, greenDAO, OrmLite, Sugar ORM, and Android SQLite. Java Realm framework is an exception because it's based on Realm engine, and its implementation is fully object-oriented.

To understand the Dalvik VM method invocations, we use Traceview [30] (Figure 6.4), an Android profiler that makes it possible to explore the impact of programming abstractions on the overall performance. Unfortunately, only the Android ORM benchmark is suitable for this exploration, due to the Traceview scalability limitations.

Figure 6.4: Traceview Profiling UI



| Name | Incl Real Time % | Incl Real Time | Calls+RecurCalls/Tot |
|---|---|---|---|
| 11 com/jingpu/android/apersistance/activeandroid/Act | 16.2% | 10264.735 | 1+0 |
| Parents | | | |
| 7 com/jingpu/android/apersistance/BaseBenchmark | 100.0% | 10264.735 | 1/1 |
| Children | | | |
| self | 0.1% | 7.392 | |
| 14 com/activeandroid/query/From.execute ()Ljava/u | 98.7% | 10127.068 | 25/25 |
| 120 com/activeandroid/Model.load (Ljava/lang/Clas | 0.7% | 68.113 | 25/25 |
| 236 com/jingpu/android/apersistance/activeandroic | 0.2% | 19.677 | 9/9 |
| 247 com/jingpu/android/apersistance/activeandroic | 0.1% | 13.932 | 7/7 |
| 268 com/jingpu/android/apersistance/activeandroic | 0.1% | 9.430 | 5/5 |
| 361 com/jingpu/android/apersistance/activeandroic | 0.0% | 3.382 | 2/2 |
| 365 com/jingpu/android/apersistance/activeandroic | 0.0% | 4.574 | 2/2 |
| 475 com/jingpu/android/apersistance/activeandroic | 0.0% | 2.328 | 1/1 |
| 578 com/activeandroid/query/Select.from (Ljava/lan | 0.0% | 1.153 | 1/1 |
| 666 com/activeandroid/query/From.where (Ljava/la | 0.0% | 2.330 | 1/1 |
| (context switch) | 0.1% | 5.356 | 7/4175 |
| 12 com/activeandroid/util/SQLiteUtils.rawQuery (Ljava/ | 16.1% | 10185.890 | 49+0 |
| 13 com/activeandroid/util/SQLiteUtils.processCursor (Lj | 16.0% | 10154.177 | 47+0 |
| 14 com/activeandroid/query/From.execute ()Ljava/util/ | 16.0% | 10127.068 | 25+0 |
| 15 com/activeandroid/Model.loadFromCursor (Landroic | 12.5% | 7927.554 | 432+0 |
| 16 com/activeandroid/Cache.getIdentifier (Ljava/lang/C | 3.0% | 1870.251 | 731+0 |
| 17 android/database/AbstractCursor.moveToPosition (I | 1.5% | 934.335 | 106+0 |

To estimate the programming effort, we use IntelliJ IDEA [11] and Metrics 1.3.6 [15] to analysis our benchmark code. We use IntelliJ IDEA to calculate ULOC metric, and use Metrics 1.3.6 to calculate McCabe cyclomatic complexity.

# Chapter 7

# Study Results

In this chapter, we report and analyze our experimental results.

## 7.1 Programming Effort Analysis

First, we compute the programming effort metrics (uncommented lines of code, and the McCabe Cyclomatic complexity) for the Android ORM and DaCapo benchmarks, respectively.

### 7.1.1 Uncommented Lines of Code Analysis

Table 7.1 shows the ULOC of Android ORM benchmark set with different frameworks. Table 7.2 shows the ULOC of DaCapo benchmark set with different frameworks. For both benchmark sets, the size of benchmark with Sugar ORM, greenDAO or ActiveAndroid is smaller compared to other two frameworks. Persistence frameworks with active record pattern—provide object interfaces without using database manipulating auxiliary classes—have smaller program sizes, such as Sugar ORM and ActiveAndroid. greenDAO has a relatively small program size because its automatic generator tool helps create the entity classes and also the DAO classes. Java Realm requires that the programmer write more ULOC than with the other frameworks to achieve the same functionality.

Table 7.1: ULOC for Android ORM Benchmark

| Android ORM | LOC |
|---|---|
| ActiveAndroid | 253 |
| greenDAO | 241 |
| OrmLite | 326 |
| Sugar ORM | 226 |
| Android SQLite | 306 |
| Java Realm | 313 |

Table 7.2: ULOC for DaCapo Benchmark

| DaCapo | LOC |
|---|---|
| ActiveAndroid | 2923 |
| greenDAO | 2200 |
| OrmLite | 3310 |
| Sugar ORM | 2911 |
| Android SQLite | 3068 |
| Java Realm | 3071 |

### 7.1.2 McCabe Cyclomatic Complexity Analysis

The McCabe Cyclomatic Complexity measures the number of flows through a certain program method. This metric increases by one when a code branch occurs, for example, conditional statements and loops, conditional expression, conditional logic operators, and exception catch branch.

When calculating the total McCabe Cyclomatic Complexity for our benchmark, we only consider the dominant classes. In our Android ORM Benchmark, we analyze only the main classes we had to write—`ActiveAndroidAORM.java`, `GreenDaoAORM.java`, `OrmliteAORM.java`, `SugarOrmAORM.java`, `SqliteAORM.java`, `RealmAORM.java`— of each persistence framework implementation which contains the concrete persist operations. In DaCapo Benchmark, we analyze major classes—`ActiveAndroidSimpleInsert.java`, `Active-`

`AndroidStandard.java`, `GreenDaoSimpleInsert.java`, `GreenDaoStandard.java`, `OrmliteSimpleInsert.java`, `OrmliteStandard.java`, `SugarOrmSimpleInsert.java`, `SugarOrmStandard.java`, `SqliteSimpleInsert.java`, `SqliteStandard.java`, `RealmSimpleInsert.java`, `RealmStandard.java`—of each persistence framework implementation, which contains the initialization and bank transaction procedures.

When considering the flow complexity, we observe from Table 7.3, Table 7.4 that for applications which involve a small set of database tables and simple operations (such as single or one-to-many object persistence operation) like Android ORM benchmark, ActiveAndroid and Sugar ORM are appropriate, while for large applications which have a complex database construction, frequent select and update operations like DaCapo benchmark, OrmLite and ActiveAndroid are better than others. Java Realm shows a higher McCabe cyclomatic complexity than in other frameworks because of the complex structure of its update procedure (Table 5.1) and the explicit database instance management pattern.

Table 7.3: The McCabe Cyclomatic Complexity for Android ORM Benchmark

| Android ORM | McCabe Cyclomatic Complexity |
|---|---|
| ActiveAndroid | 13 |
| greenDAO | 21 |
| OrmLite | 19 |
| Sugar ORM | 14 |
| Android SQLite | 23 |
| Java Realm | 26 |

Table 7.4: The McCabe Cyclomatic Complexity for the DaCapo Benchmark

| DaCapo | McCabe Cyclomatic Complexity |
|---|---|
| ActiveAndroid | 94 |
| greenDAO | 129 |
| OrmLite | 93 |
| Sugar ORM | 106 |
| Android SQLite | 101 |
| Java Realm | 153 |

### 7.1.3 Overall Programming Effort Analysis

Table 7.5 and Table 7.6 sum up the ranking of each persistence framework with regard to two programming effort metrics (ULOC and McCabe Cyclomatic Complexity) in the Android ORM and DaCapo benchmarks, respectively. We can rank six persistence frameworks in terms of their overall programming effort. In Android ORM benchmark, we have Sugar ORM < ActiveAndroid < greenDAO < OrmLite ≤ Android SQLite < Java Realm, while in DaCapo benchmark, we have ActiveAndroid < greenDAO ≤ Sugar ORM < OrmLite ≤ Android SQLite < Java Realm.

Table 7.5: Comparison of Programming Effort Metrics in the Android ORM Benchmark

| **Compared Item** | **ActiveAndroid** | **greenDAO** | **OrmLite** | **Sugar ORM** | **Android SQLite** | **Java Realm** |
|---|---|---|---|---|---|---|
| **ULOC Ranking** | 3 | 2 | 6 | 1 | 4 | 5 |
| **McCabe Ranking** | 1 | 4 | 3 | 2 | 5 | 6 |
| **Summed up Ranking** | 4 | 6 | 9 | 3 | 9 | 11 |

Table 7.6: Comparison of Programming Effort Metrics in the DaCapo Benchmark

| **Compared Item** | **ActiveAndroid** | **greenDAO** | **OrmLite** | **Sugar ORM** | **Android SQLite** | **Java Realm** |
|---|---|---|---|---|---|---|
| **ULOC Ranking** | 3 | 1 | 6 | 2 | 4 | 5 |
| **McCabe Ranking** | 2 | 5 | 1 | 4 | 3 | 6 |
| **Summed up Ranking** | 5 | 6 | 7 | 6 | 7 | 11 |

## 7.2 Experiments with the Android ORM Benchmark

In this group of experiments, we study how the types of persistence operation (initialization, insert, update, select and delete) and the variations on the number of transactions impact energy consumption and performance with different frameworks when using Android ORM benchmark. The experimental results for each type of database invocation—"initialization", "insert" , "select", "update" and "delete"—are presented in Figure7.1 - Figure7.15 [1].

The results show that overall, the energy consumption is proportional to execution time in each group of experiment. However, the persistence frameworks differ in terms of their respective energy consumption, performance, read, and write measurements. Next, we analyze the results in details by each operation.

### 7.2.1 Initialization Analysis

In the initialization stage, what persistence always does is deleting the old database if necessary, initializing the runtime context, and mostly creating database tables. Figure 7.1, Figure 7.2, and Figure 7.3 compare the energy consumption, execution time and the number of underlying database read/write operations, respectively. ActiveAndroid takes more energy ($80.25\mu Ah$) and relatively longer time ($1086ms$) to complete the initialization since this stage need to obtain various application information from the Android running context, accordingly do quite a number of class loading via the Java reflecting mechanism, and also opens the database connection in the very beginning. In addition, it also involves the largest number of database write operations, including executing extra update statements like changing foreign keys setting and user version. OrmLite costs considerable energy and time to finish its initialization. Its read/write operations are the most frequent because the table creation is inefficient: it queries the rows changed in the database each time when it executes the table creating statement. Sugar ORM is an exception in our study, as it adopts a deferred database table creation strategy rather than creating tables in the first stage. Hence its energy consumption for this stage is the smallest.

[1] To adapt to the size of the figures in the experimental results, we use "SQLite" to refer to "Android SQLite" persistence framework, and use "Realm" to refer to "Java Realm" persistence framework

Figure 7.1: Comparison of initialization energy consumption among different persistence frameworks in Android ORM benchmark. The $x$ axis indicates six Android persistence frameworks and the $y$ axis is the energy consumption of benchmark with individual framework.

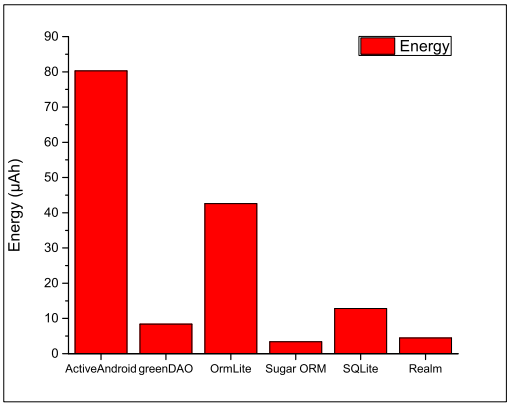



Figure 7.2: Comparison of initialization execution time among different persistence frameworks in the Android ORM Benchmark. The $x$ axis indicates six Android persistence frameworks and the $y$ axis is the execution time of benchmark with individual framework.

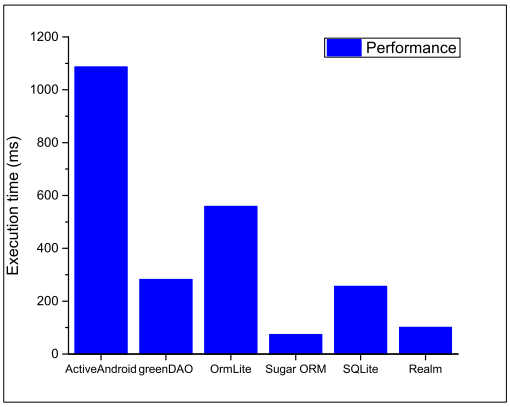
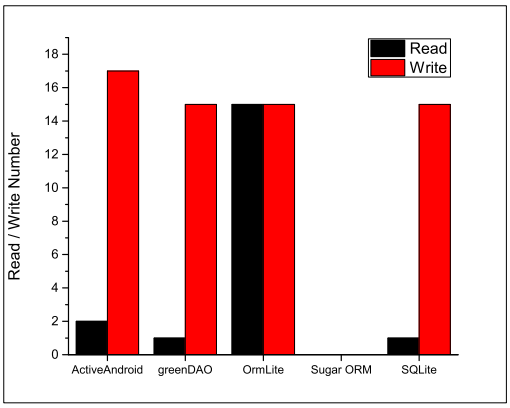


Figure 7.3: Comparison of initialization read/write operations among SQLite-based Android persistence frameworks in the Android ORM Benchmark. The $x$ axis indicates five SQLite-based Android persistence frameworks, and the $y$ axis is the operation number of benchmark with individual framework: specially, black bar shows the read operation number and red bar shows the write operation number. the read and write number for Sugar ORM are 0.




### 7.2.2 Insert Analysis

Figure 7.4, Figure 7.5, and Figure 7.6 compare the energy consumption, execution time and the number of underlying database read/write operations respectively. We observe that ActiveAndroid has the longest insert operation runtime, while Sugar ORM is second longest, with the remaining frameworks showing similar performance levels. The runtime trace reveals that interactions with the cache triggered by insert in ActiveAndroid are expensive, costing 62% of the overall execution time. Sugar ORM performs the highest number of database operations, a measurement that explains its performance. In addition, it involves considerable number of traversal queries and map usage to get mapping information, insert data, and do checks. One notable problem with Sugar ORM is that it contains log trace logic for each save which cannot be forbidden, and these features affect its energy consumption and performance. By contrast, greenDAO's performance is the best, due to its simple but efficient batch insert encapsulation, as shown in Table 5.1.

Figure 7.4: Comparison of insert energy consumption among different persistence frameworks in the Android ORM Benchmark. Six Android persistence frameworks are represented by different colors, the $x$ axis indicates number of transaction, and the $y$ axis is the energy consumption of benchmark with individual framework.

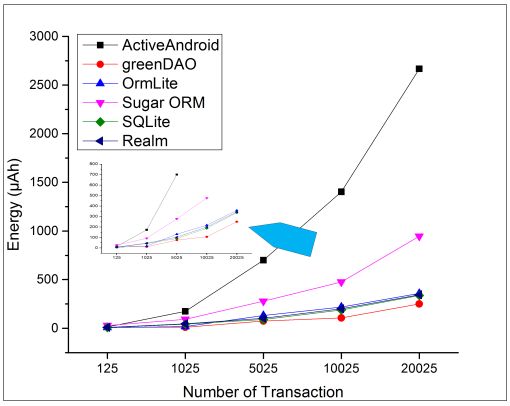



Figure 7.5: Comparison of insert execution time among different persistence frameworks in the Android ORM Benchmark. Six Android persistence frameworks are represented by different colors, the $x$ axis indicates number of transaction, and the $y$ axis is the execution time of benchmark with individual framework.

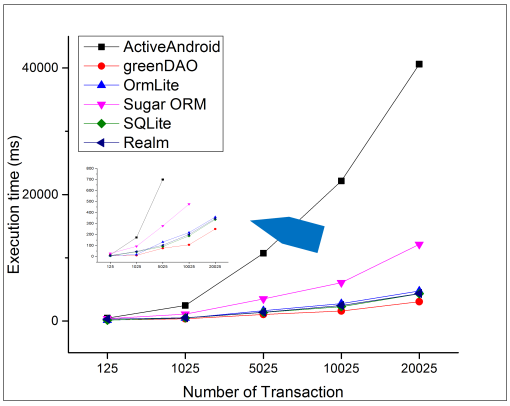
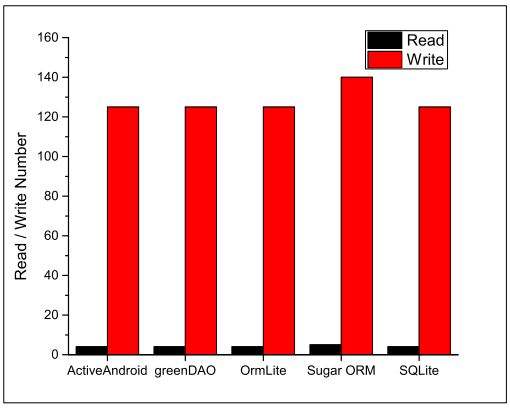


Figure 7.6: Comparison of insert read/write operations among SQLite-based Android persistence frameworks in the Android ORM Benchmark. The $x$ axis indicates five SQLite-based Android persistence frameworks, and the $y$ axis is the operation number of benchmark with individual framework when number of transaction = 125: specially, black bar shows the read operation number and red bar shows the write operation number.




### 7.2.3 Update Analysis

From Figure 7.7 and Figure 7.8 we can observe that, the cost of Java Realm update is several orders of magnitude larger than other frameworks especially when the number of transactions grows. As shown in Table 5.1, one reason is that Java Realm lacks support for batch update. Another cause is that its update procedure invokes the underlying Realm library method—`TableView.size()`—performed on a memory-hosted list of entities and costing 98.3% of the overall execution time. The cost of Sugar ORM is still high because it has the highest number of read and write operations. Sugar ORM needs to find the target object before updating it. This finding procedure involves an expensive recursive design for the `SugarRecord.find()` method, costing 96% of the overall execution time. OrmLite and Java SQLite updates have comparable efficiency in our experiment since both of them provide direct update operations while other four frameworks adopt query-and-save modes which are not very economic.

Figure 7.7: Comparison of update energy consumption among different persistence frameworks in the Android ORM Benchmark. Six Android persistence frameworks are represented by different colors, the $x$ axis indicates number of transaction, and the $y$ axis is the energy consumption of benchmark with individual framework.

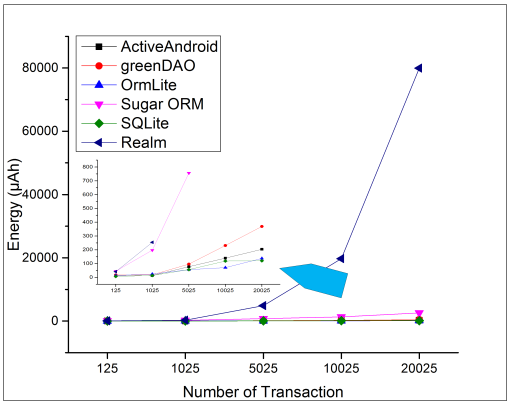



Figure 7.8: Comparison of update execution time among different persistence frameworks in the Android ORM Benchmark. Six Android persistence frameworks are represented by different colors, the $x$ axis indicates number of transactions, and the $y$ axis is the execution time of benchmark with individual framework.

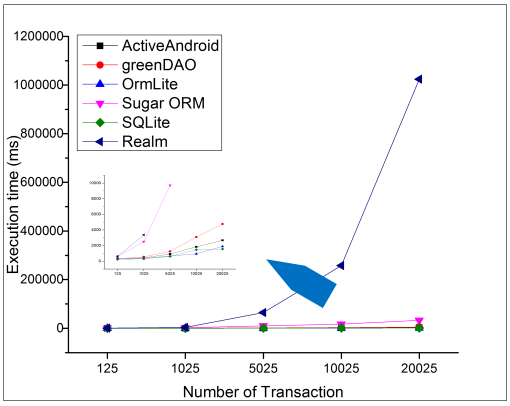
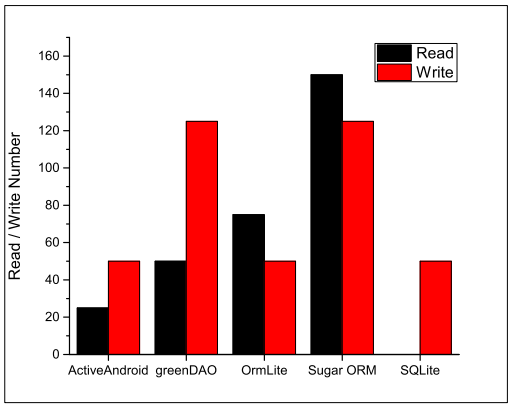


Figure 7.9: Comparison of update read/write operations among different persistence frameworks in the Android ORM Benchmark. The $x$ axis indicates five SQLite-based Android persistence frameworks, and the $y$ axis is the operation number of benchmark with individual framework when number of transactions = 125: specially, black bar shows the read operation number and red bar shows the write operation number.




### 7.2.4 Select and Delete Analysis

For select and delete operations, we observe from Figure 7.10, Figure 7.11, Figure 7.12 and Figure 7.13, Figure 7.14, Figure 7.15 that Sugar ORM: 1) has the worst performance in terms of execution time and energy consumption; 2) has the highest number of database operations, as it executes an extra query for each atomic operation. Sugar ORM's select and delete inefficiency is related to the extra underlying operations, but as mentioned above, most of the execution time is spent in the recursive $find$ method. OrmLite, greenDAO and Android SQLite show comparable performance levels with these two operations.

Figure 7.10: Comparison of select energy consumption among different persistence frameworks in the Android ORM Benchmark. Six Android persistence frameworks are represented by different colors, the $x$ axis indicates number of transactions, and the $y$ axis is the energy consumption of benchmark with individual framework.

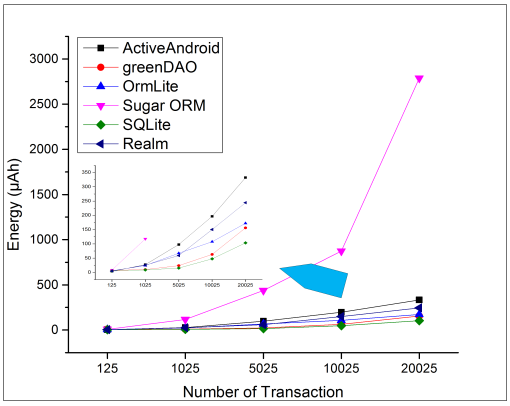



Figure 7.11: Comparison of select execution time among different persistence frameworks in the Android ORM Benchmark. Six Android persistence frameworks are represented by different colors, the $x$ axis indicates number of transactions, and the $y$ axis is the execution time of benchmark with individual framework.

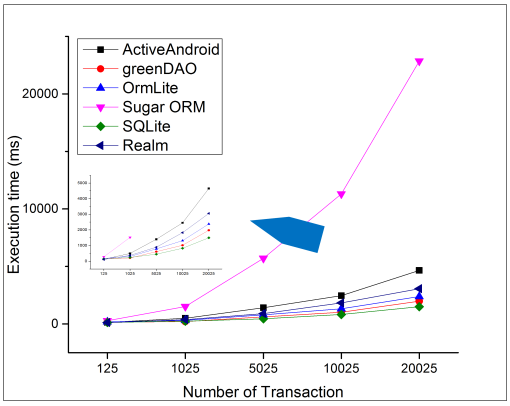


Figure 7.12: Comparison of select read/write operations among SQLite-based Android persistence frameworks in the Android ORM Benchmark. The $x$ axis indicates five SQLite-based Android persistence frameworks, and the $y$ axis is the operation number of benchmark with individual framework when number of transactions = 125: specially, black bar shows the read operation number and red bar shows the write operation number.

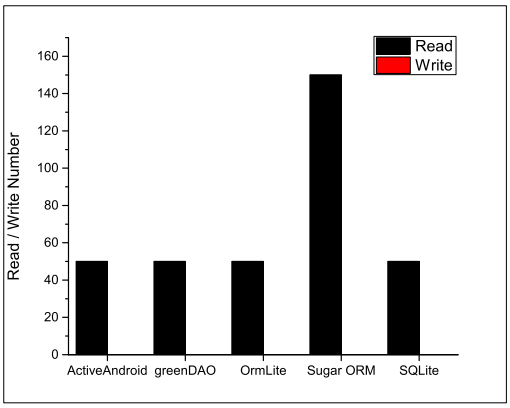



Figure 7.13: Comparison of delete energy consumption among different persistence frameworks in the Android ORM Benchmark. Six Android persistence frameworks are represented by different colors, the $x$ axis indicates number of transactions, and the $y$ axis is the energy consumption of benchmark with individual framework.

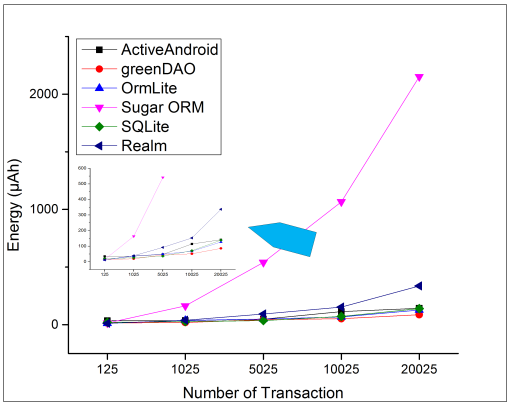


Figure 7.14: Comparison of delete execution time among different persistence frameworks in the Android ORM Benchmark. Six Android persistence frameworks are represented by different colors, the $x$ axis indicates number of transaction, and the $y$ axis is the execution time of benchmark with individual framework.

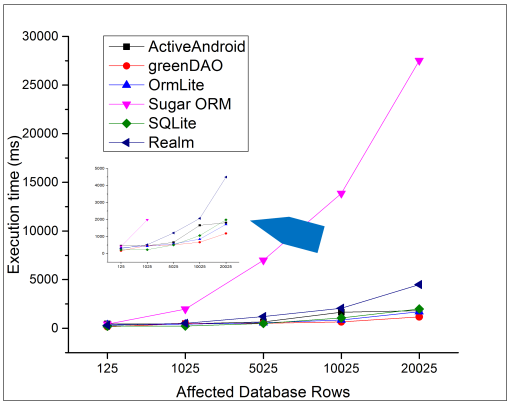
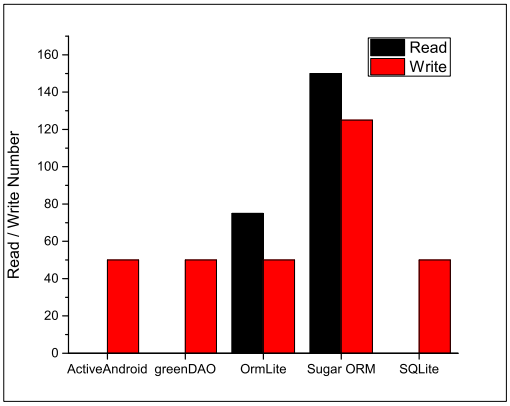



Figure 7.15: Comparison of delete read/write operations among SQLite-based Android persistence frameworks in the Android ORM Benchmark. The $x$ axis indicates five SQLite-based Android persistence frameworks, and the $y$ axis is the operation number of benchmark with individual framework when number of transactions = 125: specially, black bar shows the read operation number and red bar shows the write operation number.



### 7.2.5 Overall Analysis

Table 7.7 sums up the ranking of each persistence framework with regard to different database operation invocations (i.e., Initialization, Insert, Update, Select, and Delete).

Table 7.7: Overall Performance Comparison of Persistence Frameworks in the Android ORM experiment

| **Compared Item** | **ActiveAndroid** | **greenDAO** | **OrmLite** | **Sugar ORM** | **Android SQLite** | **Java Realm** |
|---|---|---|---|---|---|---|
| **Initialization Ranking** | 6 | 4 | 5 | **1** | 3 | 2 |
| **Insert Ranking** | 6 | 1 | 4 | 5 | 2 | 3 |
| **Update Ranking** | 3 | 4 | 2 | 5 | 1 | 6 |
| **Select Ranking** | 5 | 2 | 3 | 6 | 1 | 4 |
| **Delete Ranking** | 3 | 1 | 2 | 6 | 3 | 5 |
| **Summed up Ranking** | 23 | 12 | 16 | 23 | 10 | 20 |

From Table 7.7, Table 7.5, and our above analysis, we can draw the following conclusions:

1 By adding up the ranking of different operations, we can rank these frameworks in terms of their overall performance: Android SQLite >greenDAO >OrmLite >Java Realm >Sugar ORM $\geq$ ActiveAndroid , where ">" means "faster than".

2 Considering the programming effort of implementing all database operations using different frameworks, sugar ORM, ActiveAndroid and greenDAO require less programming effort.

3 When considering the balance of programming effort and performance, greenDAO can be generally recommended for developing database-oriented mobile application with standard database operation/schema complexity.

4 Sugar ORM would not be an optimal choice when the dominating operations in a mobile application are select or delete, while Java Realm would not be optimal when the dominating operation is update.

## 7.3 Experiments with the DaCapo benchmark

In this group of experiments, we use the DaCapo benchmark to study how the energy consumption and performance of each framework changes in relation to the number of executed bank transactions. The benchmark comes with a total of 41,971 records, so in our experiments we differ the number of bank transactions. We observe that in the DaCapo experiments, the energy consumption is proportional to the execution time for each run. However, like the Android ORM benchmark, the overall performance differs with different persistence frameworks.

Figure 7.16 and Figure 7.17 show the energy/performance and read/write operations of the DaCapo Initialization, respectively. The dominant database operation in this phase is insert, and Figure 7.16 shows the performance levels consistent with those seen in the Android ORM benchmark for the same operation: Sugar ORM and ActiveAndroid have the longest runtime. greenDAO performs better than Android SQLite, possibly due to greenDAO supporting batch insert (Table 5.1).

Figure 7.16: Comparison of initialization energy consumption / execution time among different persistence frameworks in DaCapo Benchmark when database record = 41,971.The $x$ axis indicates six Android persistence frameworks, black bar shows the energy consumption and red bar shows the execution time.

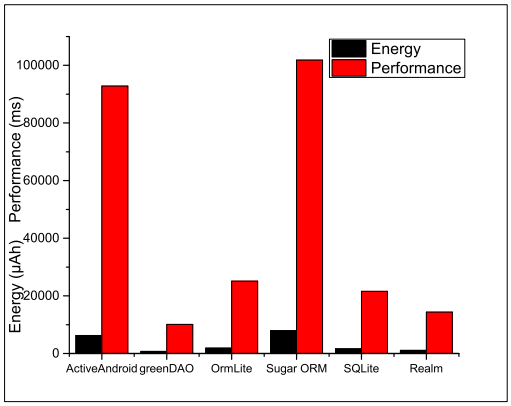
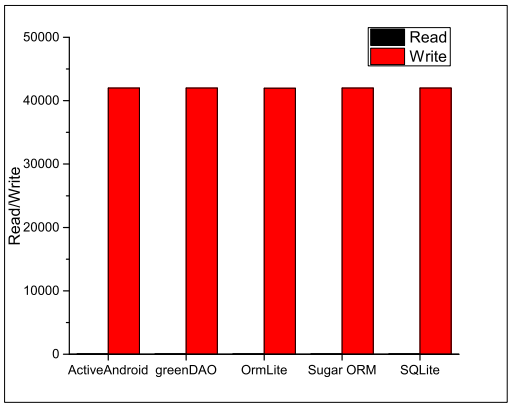



Figure 7.17: Comparison of initialization read/write operations among SQLite-based Android persistence frameworks in DaCapo Benchmark when database record = 41,971.The $x$ axis indicates six Android persistence frameworks, and the $y$ axis is the operation number of benchmark with individual framework: specially, black bar shows the read operation number and red bar shows the write operation number.




In our measurements, we vary the number of bank transactions over the following numbers: 40, 120, 200, 280, 360, 440, 520, 600, 800, 1000, and 1500. The total

number of transactions is the sum of basic bank transactions, as listed in Table 7.8. Each transaction comprises a complex set of database operations. The key transactions in each run are "New Order", "Payment by Name", and "Payment by ID", which mainly execute the "query" and "update" operations. In our experiments, "New Order" itself takes 42.5% of the entire number of transactions. Figure 7.18, Figure 7.19 show the energy consumption and execution time for each transaction number. Figure 7.20, Figure 7.21 show the read/write operation number, and Table 7.8 shows the average time consumption for each transaction.

Figure 7.18: Comparison of transaction energy consumption among different persistence frameworks in DaCapo Benchmark when database record = 41,971.The $x$ axis indicates six Android persistence frameworks, and the $y$ axis is the energy consumption of benchmark with individual framework.

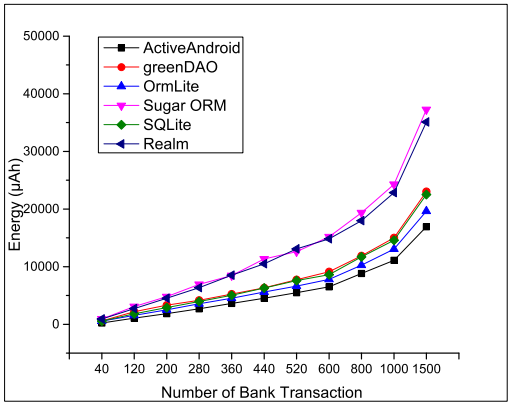


Figure 7.19: Comparison of transaction execution time among different persistence frameworks in DaCapo Benchmark when database record = 41,971.The $x$ axis indicates six Android persistence frameworks, and the $y$ axis is the execution time of benchmark with individual framework.

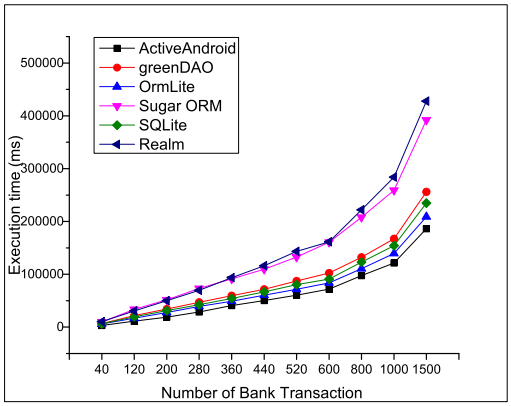



Figure 7.20: Comparison of transaction read operations among SQLite-based Android persistence frameworks in DaCapo Benchmark when database record = 41,971. The $x$ axis indicates six Android persistence frameworks, and the $y$ axis is the operation number of benchmark with individual framework.

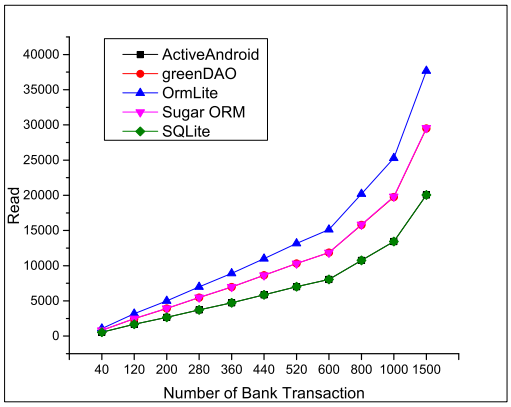
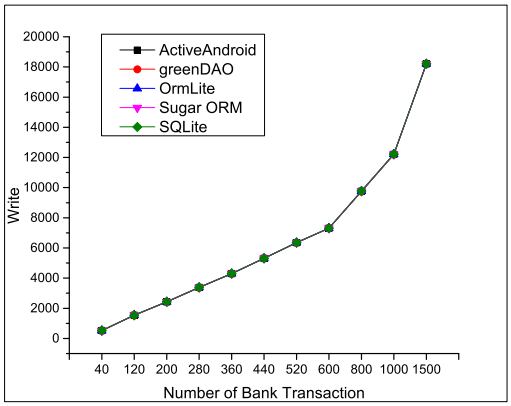


Figure 7.21: Comparison of transaction write operations among SQLite-based Android persistence frameworks in DaCapo Benchmark when database record = 41,971.The $x$ axis indicates six Android persistence frameworks, and the $y$ axis is the operation number of benchmark with individual framework.




From Table 7.8, we observe that Java Realm and Sugar ORM have the longest execution time when executing the transactions whose major database operation is update (e.g., "New order", "New order rollback", "Payment by name", and "Payment by ID"). This conclusion is consistent with that derived from the Android ORM update experiments discussed above. Android SQLite takes rather long to execute, as it involves database aggregation (e.g., *sum*, and the table queried had 30,060 records) and arithmetic operations (e.g. $field - 1$) in the select clause. Meanwhile, as ActiveAndroid only uses raw SQL manipulation interface for complex update operations (Table 5.1), its performance is the fastest, albeit at the cost of additional programming effort.

Table 7.8 also shows that greenDAO, ActiveAndroid, and Android SQLite incur higher execution costs for the "Stock level" transaction. One possible explanation is that this transaction contains a multiple entities conjunctive query action, and only these three frameworks provide the SQL "JOIN" interface (Table. 5.1). Supporting this interface is known to be computationally expensive. However, the SQL "JOIN" interface can help save the programming effort.

Table 7.8: Performance Analysis for individual DaCapo bank transactions, this table shows the execution time (ms) for each persistence framework in each transaction type

| **Transaction Type** | **ActiveAndroid** | **greenDAO** | **OrmLite** | **Sugar ORM** | **Android SQLite** | **Java Realm** |
|---|---|---|---|---|---|---|
| Stock level | 136 | 189 | 86 | 91 | 98 | 41 |
| Order status by name | 72 | 101 | 95 | 100 | 112 | 36 |
| Order status by ID | 106 | 91 | 94 | 113 | 108 | 33 |
| Payment by name | 50 | 55 | 50 | 124 | 59 | 86 |
| Payment by ID | 25 | 32 | 40 | 119 | 44 | 60 |
| Delivery schedule | 1 | 1 | 1 | 1 | 1 | 1 |
| New order | 177 | 209 | 189 | 402 | 272 | 496 |
| New order rollback | 186 | 271 | 176 | 299 | 248 | 427 |
| Payment by name exception | 12 | 12 | 11 | 11 | 13 | 31 |
| Order status by name exception | 13 | 14 | 11 | 14 | 13 | 35 |

From Figure 7.18 - Figure 7.21, Table 7.2, and Table 7.6 we can conclude that:

1 ActiveAndroid offers the overall best performance for all DaCapo transactions. It shows the best performance for the most common transactions, and also the least programming effort. Besides, its execution invokes the smallest number of database operations, due to its caching mechanism.

2 Sugar ORM and Java Realm have the longest execution time, in line with the Android ORM benchmark's results above.

3 greenDAO's performance and programming effort is in the middle, note that it

has the smallest lines of code—taking 24.5% fewer uncommented lines of code to implement than the other frameworks.

# Chapter 8

# Recommendation Model

## 8.1 Recommendation Model

In this section, we further propose a recommendation model for ascertaining the overall utility of persistence frameworks, in terms of the **P**erformance, **E**nergy consumption, and **P**rogramming **E**ffort (PEP). We apply this model to the two benchmarks studied in the thesis.

### 8.1.1 Design of PEP Model

As our experiments show, the utility of persistence frameworks is closely related to application features (e.g., data schema complexity, operations involved, data records manipulated, database operations executed). Therefore, we use $p$ to denote an application with a set of given features.

Let $\mathcal{O} = \{o = \textbf{ActiveAndroid}, \textbf{greenDAO}, \textbf{OrmLite}, \textbf{Sugar ORM}, \textbf{Android SQLite}, \textbf{Java Realm}\}$ denote all the considered ORM/OO frameworks. Let $E_o(p), \forall o \in \mathcal{O}$ denote the energy consumption of six different implementation of $p$ using various ORM/OO frameworks $o$, while let $T_o(p), \forall o \in \mathcal{O}$ denote the time consumption of six different implementation of $p$ using various ORM/OO frameworks $o$. We use the Euclidean distance of a two dimensional vector to calculate the overall performance, which can be denoted as $P_o p = \sqrt{T_o(p)^2 + E_o(p)^2}, \forall o \in \mathcal{O}$.

The programming effort is represented by the ULOC and McCabe Cyclomatic Com-

plexity, and here we use $L_o(p), \forall o \in \mathcal{O}$ to denote the programming size of six different implementations of the project $p$ using different ORM/OO frameworks, and use $M_o(p), \forall o \in \mathcal{O}$ to denote the McCabe Cyclomatic Complexity of six different implementations of the project $p$ using different ORM/OO frameworks. We use the Euclidean distance of a two dimensional vector to calculate the overall programming effort index $C_o(p)$:

$$C_o(p) = \sqrt{\frac{L_o(p)}{\min(L_o(p), \forall o \in \mathcal{O})}^2 + \frac{M_o(p)}{\min(M_o(p), \forall o \in \mathcal{O})}^2}, \forall o \in \mathcal{O} \tag{8.1}$$

We consider both the framework performance and programming effort, to compute the utility index $I_o(p)$:

$$I_o(p) = \frac{\min(P_o(p), \forall o \in \mathcal{O})}{P_o(p)} / C_o(p) \tag{8.2}$$

The equation's first part, $\frac{\min(P_o(p), \forall o \in \mathcal{O})}{P_o(p)}$, compares the performance provided by an ORM framework $o$ based implementation, and the implementation that has the best performance. The equation's second part, $C_o(p)$, compares the overall programming effort between an implementation $o(p)$ and the implementation that requires the minimal programming effort.

For example, for project $p$, Android SQLite might provide the best overall performance, while the greenDAO based implementation consumes twice the energy or takes twice the execution time. Therefore, the overall performance index of $P_{greenDAO}(p)$ is 0.5. On the other hand, the greenDAO based implementation might have the least overall programming effort. Therefore, the implementation complexity index of $L_{greenDAO}p$ is 1. Thus, the overall utility index is $0.5/1 = 0.5$.

When the utility index $I_o(p)$ of an ORM/OO $o$ based implementation is close to 1, the implementation is likely to offer good performance, while requiring low programming effort.

Application developers apply dissimilar standards to judge the trade-offs between performance and programming effort. Some developers focus solely on performance, while others may prefer the shortest time-to-market. We introduce $\tau$ to indicate these preferences.

$$I_o(p) = \frac{\min(P_o(p), \forall o \in \mathcal{O})}{P_o(p)} / (C_o(p))^\tau \tag{8.3}$$

,where $\tau \geq 0$.

When $\tau > 1$, the larger $\tau$ is, the more weight is assigned to the programming effort target. Otherwise, when $0 \leq \tau < 1$, the lower $\tau$ is, the more weight is assigned to the performance target. Specially, when $\tau = 0$ only performance target is considered, when $\tau = 1$ performance and programming effort are equally considered.

### 8.1.2 Benchmark Evaluation

Next we apply the PEP model to the Android ORM and DaCapo benchmarks. We consider typical low and high transaction volume respectively in each benchmark. Specially, for the Android ORM benchmark, we evaluate two sets of input, 1,025 transactions and 20,025 transactions. For the DaCapo benchmark, we evaluate two sets of input, 40 transactions and 1,500 transactions. For each set of input, we set $\tau$ to 0.5, 1, 1.5 respectively, to show whether the programmer is willing to spend additional programming effort to improve performance. Specifically, when $\tau = 0.5$, the programmer's main concern is performance, when $\tau = 1$, the balance of performance and programming effort is important, when $\tau = 1.5$, the programmer pays more attention to programming effort.

Table 8.1 shows the calculated values of utility index —$I_o(p)$ for all cases. We are interested in the largest $I_o(p)$ which is most close to 1 in each case because it indicates the best energy consumption, performance and programming effort tradeoffs.

From this table, we have the following findings which are consistent with the guidelines we concluded from previous two experiments:

For the DaCapo benchmark, ActiveAndroid represents the best performance and programming effort trade-off for all cases (both low-scale and high-scale transactions are executed).

For the Android ORM benchmark, the choice varies:

1 Android SQLite is the first choice if performance is preferred.

2 Android SQLite is the first choice if the balance of performance and programming effort is needed.

3 greenDAO taking the lead if the major aim is to minimize the programming effort.

Table 8.1: Recommendation Result For DaCapo and Android ORM Benchmark

| **Benchmark** | **Parameters & Index** | **Active Android** | **greenDAO** | **OrmLite** | **Sugar ORM** | **Android SQLite** | **Java Realm** |
|---|---|---|---|---|---|---|---|
| DaCapo | transactions=40, $\tau$=0.5 | **0.9246** | 0.4457 | 0.4824 | 0.2980 | 0.4645 | 0.2494 |
| DaCapo | transactions=40, $\tau$=1 | **0.8549** | 0.4080 | 0.4311 | 0.2686 | 0.4171 | 0.2022 |
| DaCapo | transactions=40, $\tau$=1.5 | **0.7905** | 0.3735 | 0.3852 | 0.2420 | 0.3745 | 0.1640 |
| DaCapo | transactions=1500, $\tau$=0.5 | **0.9246** | 0.6654 | 0.7967 | 0.4278 | 0.7117 | 0.3533 |
| DaCapo | transactions=1500, $\tau$=1 | **0.8549** | 0.6091 | 0.7120 | 0.3855 | 0.6391 | 0.2865 |
| DaCapo | transactions=1500, $\tau$=1.5 | **0.7905** | 0.5575 | 0.6362 | 0.3474 | 0.5738 | 0.2323 |
| Android ORM | transactions=1025, $\tau$=0.5 | 0.2967 | 0.7053 | 0.5894 | 0.2066 | **0.8002** | 0.2374 |
| Android ORM | transactions=1025, $\tau$=1 | 0.2882 | 0.6091 | 0.4891 | 0.2027 | **0.6404** | 0.1825 |
| Android ORM | transactions=1025, $\tau$=1.5 | 0.2800 | **0.5260** | 0.4059 | 0.1989 | 0.5124 | 0.1403 |
| Android ORM | transactions=20025, $\tau$=0.5 | 0.1836 | 0.7375 | 0.7063 | 0.0984 | **0.8002** | 0.0071 |
| Android ORM | transactions=20025, $\tau$=1 | 0.1784 | 0.6369 | 0.5861 | 0.0966 | **0.6404** | 0.0055 |
| Android ORM | transactions=20025, $\tau$=1.5 | 0.1733 | **0.5500** | 0.4864 | 0.0948 | 0.5124 | 0.0042 |

# Chapter 9

# Threats to Validity

Next, we discuss the threats to the validity of our experimental results. Although in designing our experimental evaluation, we tried to perform as an objective assessment as possible, our design choices could have certainly affected the validity and applicability of our conclusions.

## 9.1 External Threats

The key external threat to validity is our choice of the hardware devices, Android version, and profiling equipment. Specifically, we conduct our experiments with an LG mobile phone, with 1.2GHz quad-core Qualcomm Snapdragon 400 processor, running Android 4.4.2 KitKat, profiled with the Monsoon Power Monitor. Even though these experimental parameters are representative of the Android computing ecosystem, changing any of these parameters could have affected the outcome of our experiments.

## 9.2 Internal Threats

The key internal threat to validity are our design choices for structuring the database and the persistence application functionality. Specifically, while our Android ORM benchmark set explores the object features of Android persistence frameworks, the original DaCapo [39] H2 benchmark manipulates relational database structures di-

rectly, without stress-testing the object-oriented persistence frameworks around it. To retarget DaCapo to focus on persistence frameworks rather than the JVM alone, we adapted the benchmark to make use of transparent persistence as a means of accessing its database-related functionality. Nevertheless, the relatively large scale of data volume, with the select and update operations bank transactions dominating the execution, this benchmark is representative of a large class of database application systems, but not all of them.

# Chapter 10

# Future Work

This chapter discusses potential future work directions of this thesis, which include expanding the studied platforms and benchmarks, understanding the concurrency behavior of Android persistence frameworks, designing energy efficient persistence frameworks, and expanding the study of the Android middleware layer.

## 10.1 Expanding Platform and Case Study

One possible direction is extending the work in this thesis to other platforms such as iOS devices and PC. Persisting data is a core concern both for mobile and PC devices. iOS developers also face various choices when it comes to third-party persistence libraries, such as MagicalRecord [14], SugarRecord [28], etc. Java developers also face the similar selection dilemma, such as Hibernate [10], Apache OpenJPA [4], MyBatis [18], etc. Possible future work in this domain can focus on analyzing the performance and programming efforts of the persistence frameworks involved, applying our recommendation model to help select the best framework for a given set of design constraints, and optimizing the persistence functionality of an application.

Another aspect to improve our work is to consider additional case studies. In this thesis, our study is based on the basic persistence features such as relationship support, raw query interface support, batch operations, aggregation support, etc. However, our benchmark selection doesn't cover the whole feature set. In the future, we will investigate more database-involved real world applications in order to compare other advanced feature differences among the persistence frameworks, such as

complex query support (e.g., different table JOIN strategy; IN, LIKE, wild-cards operator support), caching , data encryption and remote persistence. We expect performance and programming effort discrepancies to remain, whenever different persistence frameworks provide the same feature.

## 10.2 Understanding the Concurrency Behavior of Android Persistence Framework

Concurrency support is another important feature for persistence frameworks. Multiple-threaded applications often behave quite differently from single-threaded ones. One question about Android parallel processing is whether single-threaded Android applications would utilize the multiple cores in a mobile device. According to one study with ARM architecture processors [67], Android applications benefit from multiple core processors even without multiple-threaded design. The application code will be sub-divided into multiple threads by the scheduler and be distributed to different cores, as long as the application's code can be divided into exclusive parts. We expect that the energy consumption will not always be proportional to execution time in multiple-threaded environments.

## 10.3 Designing Energy Efficient Persistence Frameworks

One research goal of this thesis is to give out a set of guidelines for persistence framework designers. As the next step, one may want to employ these findings to design an energy aware persistence framework. From the current study, we discovered that some abstraction patterns incur notably less energy, or shorter execution time. By comparing and analyzing different Android persistence frameworks, such as database access interface pattern, database feature encapsulation, excessive validity checks, inefficient cache maintenance and access, inappropriate context switch, extra underlying database access, etc., we will further explore the relationship between the abstraction patterns and overall performance to facilitate the development of energy-efficient persistence frameworks.

## 10.4 Expanding the Study of Android Middleware

This thesis studies persistence frameworks, thus making the first step in understanding the tradeoffs between the energy consumption, performance, and programming effort of the persistence middleware layer. However, other popular middleware systems are frequently used as well, and as such can have a remarkable impact on the execution efficiency of mobile devices. For instance, networking frameworks (such as such as AFNetworking [2], JSONModel [12]) for requesting and responding data, and providing functionality like serialization and validation over the network, image downloading and caching frameworks like SDWebImage [5], and game building engines such as libGDX [13], Godot [7] and Monkey X Pro [16]. One can study these other middleware systems using an approach similar to ours to understand the trade-offs endemic to these systems and thus help mobile programmers develop energy and performance efficient applications, while expending the expected amounts of programming effort.

# Chapter 11

# Conclusions

In this thesis, we present a systematic study of popular Android ORM/OO persistence frameworks. We first compare and contrast the frameworks to present an overview of their features and capabilities. Then we present our experimental design of two sets of benchmarks, variables, and input parameters, used to explore the performance, energy consumption and programming effort of these frameworks in different application scenarios. We describe the benchmark results, and also use our analysis of the framework features and capabilities to explain the results. Finally, we propose a recommendation model to help guide mobile developers in their decision making process when choosing a persistence framework for a given application. To the best of our knowledge, this research is the first step to better understand the trade-offs between the performance, energy efficiency, and programming effort of Android persistence frameworks.

# Bibliography


[1] Activeandroid by pardom. http://www.activeandroid.com/.

[2] Afnetworking 2.0. http://nshipster.com/afnetworking-2/.

[3] Android sqlite. https://developer.android.com/reference/android/database/sqlite/package-summary.html.

[4] Apache openjpa. http://openjpa.apache.org/.

[5] Asynchronous image downloader. https://github.com/rs/SDWebImage.

[6] Definition of ampere hour. http://whatis.techtarget.com/definition/ampere-hour-Ah-or-amp-hour.

[7] Godot engine - free and open source 2d and 3d game engine. http://www.godotengine.org/.

[8] greendao: the superfast android orm for sqlite. http://greenrobot.org/greendao/.

[9] Haptic energy consumption - texas instruments. http://www.ti.com/lit/an/sloa194/sloa194.pdf.

[10] Hibernate orm. http://hibernate.org/orm/.

[11] Intellij idea plugins. https://plugins.jetbrains.com/plugin/?idea&id=4509.

[12] Jsonmodel for ios and osx: Magical data modelling framework. http://www.jsonmodel.com/.

[13] libgdx: Desktop/android/blackberry/ios/html5 java game development framework. https://libgdx.badlogicgames.com/.

[14] Magicalrecord: Core data api. https://github.com/magicalpanda/MagicalRecord.

[15] Metrics 1.3.6. http://metrics.sourceforge.net/.

[16] Monkey x: X-platform programming language. http://www.monkey-x.com/.

[17] Monsoon power monitor. https://www.msoon.com/LabEquipment/PowerMonitor/.

[18] Mybatis. http://www.mybatis.org/mybatis-3/.

[19] Obfuscation (software). https://en.wikipedia.org/wiki/Obfuscation_(software).

[20] Orm mismatch. http://www.tutorialspoint.com/hibernate/orm_overview.htm.

[21] Ormlite - lightweight object relational mapping (orm) java package. http://ormlite.com/.

[22] Realm database engine. https://realm.io/.

[23] Realm java. https://realm.io/docs/java/latest/.

[24] Smartphone os market share, 2015 q2. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[25] Sqlite database engine. https://www.sqlite.org/.

[26] Statista: Mobile apps available in leading stores (july 2015). http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[27] Sugar orm - insanely easy way to work with android databases. http://satyan.github.io/sugar/.

[28] Sugarrecord: A data management library. https://github.com/pepibumur/SugarRecord.

[29] A survey of object-relational mapping (orm) libraries for android and ios. https://dzone.com/articles/a-survey-of-object-relational-mapping-orm-librarie.

[30] Traceview. https://developer.android.com/studio/profile/traceview-walkthru.html.

[31] Understanding energy. https://www.kompulsa.com/energy-index/general-energy-data-and-power-consumption/.

[32] Xposed framework. http://repo.xposed.info/module/de.robv.android.xposed.installer.

[33] Alephnaught.com. Javaone conference trip report: Enterprise javabeans technology: Developing and deploying business applications as components. retrieved 2012-06-17.

[34] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[35] Victor R Basili, Lionel C Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, 22(10):751–761, 1996.

[36] Christian Bauer and Gavin King. *Java Persistance with Hibernate*. Dreamtech Press, 2006.

[37] Frank Bellosa, Andreas Weissel, Martin Waitz, and Simon Kellner. Event-driven energy accounting for dynamic thermal management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP03)*, volume 22, 2003.

[38] S Bhatti, Z Abro, and F Rufabro. Performance evaluation of java based object relational mapping tool. *Mehran University Research Journal of Engineering and Technology*, 32(2):159–166, 2013.

[39] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[40] David John Brandow, John Murray Childs, Robert Donald Close, JY Eric Giguere, and Geno Coschi. Java-based data access object, August 30 2005. US Patent 6,938,041.

[41] Roderic Geoffrey Galton Cattell, Douglas K Barry, Dirk Bartels, Mark Berler, Jeff Eastman, Sophie Gamerman, David Jordan, Adam Springer, Henry Strickland, and Drew Wade. *The object database standard: ODMG 2.0*, volume 131. Morgan Kaufmann Publishers San Mateo, 1997.

[42] Shyam R Chidamber, David P Darcy, and Chris F Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. *Software Engineering, IEEE Transactions on*, 24(8):629–639, 1998.

[43] Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. Energy types. In *ACM SIGPLAN Notices*, volume 47, pages 831–850. ACM, 2012.

[44] Chris J Date. *Relational database: selected writings*, volume 1. Addison-Wesley, 1986.

[45] John Deacon. Model-view-controller (mvc) architecture. *Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf*, 2009.

[46] Linda DeMichiel and Michael Keith. Java persistence api. *JSR*, 220, 2006.

[47] Mian Dong and Lin Zhong. Sesame: Self-constructive system energy modeling for battery-powered mobile systems. *arXiv preprint arXiv:1012.2831*, 2010.

[48] Fernando Brito e Abreu and Rogério Carapuça. Candidate metrics for object-oriented software within a taxonomy framework. *Journal of Systems and Software*, 26(1):87–96, 1994.

[49] Geoffrey K Gill and Chris F Kemerer. Cyclomatic complexity density and software maintenance productivity. *Software Engineering, IEEE Transactions on*, 17(12):1284–1288, 1991.

[50] Maurice H Halstead. Toward a theoretical basis for estimating programming effort. In *Proceedings of the 1975 annual conference*, pages 222–224. ACM, 1975.

[51] Shuai Hao, Ding Li, William GJ Halfond, and Ramesh Govindan. Estimating mobile application energy consumption using program analysis. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 92–101. IEEE, 2013.

[52] Rachel Harrison, Steve J Counsell, and Reuben V Nithi. An evaluation of the mood set of object-oriented software metrics. *Software Engineering, IEEE Transactions on*, 24(6):491–496, 1998.

[53] Abram Hindle. Green mining: A methodology of relating software change to power consumption. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 78–87. IEEE Press, 2012.

[54] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. A classification of object-relational impedance mismatch. In *Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA'09. First International Conference on*, pages 36–43. IEEE, 2009.

[55] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 93. IEEE Computer Society, 2003.

[56] S Jha. Poorly written apps can sap 30 to 40% of a phones juice., june 2011. In *CEO, Motorola Mobility, Bank of America Merrill Lynch 2011 Technology Conference*.

[57] Mick Jordan. A comparative study of persistence mechanisms for the java platform. 2004.

[58] Mick J Jordan and Malcolm P Atkinson. Orthogonal persistence for javaa mid-term report. *Morrison et al.[161]*, pages 335–352, 1999.

[59] Russ Joseph and Margaret Martonosi. Run-time power estimation in high performance microprocessors. In *Proceedings of the 2001 international symposium on Low power electronics and design*, pages 135–140. ACM, 2001.

[60] Lv Junyan, Xu Shiguo, and Li Yijie. Application research of embedded database sqlite. In *Information Technology and Applications, 2009. IFITA'09. International Forum on*, volume 2, pages 539–543. IEEE, 2009.

[61] Joseph P Kearney, Robert L Sedlmeyer, William B Thompson, Michael A Gray, and Michael A Adler. Software complexity measurement. *Communications of the ACM*, 29(11):1044–1050, 1986.

[62] Young-Woo Kwon and Eli Tilevich. Reducing the energy consumption of mobile applications behind the scenes. In *2013 IEEE International Conference on Software Maintenance*, pages 170–179. IEEE, 2013.

[63] Doug Lea. Overview of the util. concurrent package.

[64] Ding Li, Shuai Hao, Jiaping Gui, and William GJ Halfond. An empirical study of the energy consumption of Android applications. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 121–130. IEEE, 2014.

[65] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *Journal of systems and software*, 23(2):111–122, 1993.

[66] Mark Lorenz and Jeff Kidd. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., 1994.

[67] Frank Maker and Y-h Chan. A survey on android vs. linux. *University of California*, pages 1–10, 2009.

[68] Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[69] Hztzefa Mehta, Robert Michael Owens, and Mary Jane Irwin. Instruction level power profiling. In *Acoustics, Speech, and Signal Processing, 1996. ICASSP-96. Conference Proceedings., 1996 IEEE International Conference on*, volume 6, pages 3326–3329. IEEE, 1996.

[70] Frederic P Miller, Agnes F Vandome, and John McBrewster. Active record pattern. 2010.

[71] Shruti Mukherjee and Ishani Mondal. Future practicability of android application development with new android libraries and frameworks. *International Journal of Computer Science and Information Technologies*, 5(4):5575–5579, 2014.

[72] Glenford J Myers. An extension to the cyclomatic measure of program complexity. *ACM Sigplan Notices*, 12(10):61–64, 1977.

[73] Chris Newman. *SQLite (Developer's Library)*. Sams, 2004.

[74] Enrique I Oviedo. Control flow, data flow and program complexity. In *Software engineering metrics I*, pages 52–65. McGraw-Hill, Inc., 1993.

[75] Dipti Phutela and Mindfire Solutions. Hibernate vs jdbc. *Mindfire Solutions http://www. mindfiresolutions. com/mindfire/Java_ Hibernate_JDBC. pdf.*

[76] Gustavo Pinto, Fernando Castor, and Yu David Liu. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 22–31. ACM, 2014.

[77] Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding energy behaviors of thread management constructs. In *ACM SIGPLAN Notices*, volume 49, pages 345–360. ACM, 2014.

[78] Julien Ponge. Fork and join: Java can excel at painless parallel programming too!, 2011.

[79] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William E. Lorensen, et al. *Object-oriented modeling and design*, volume 199. Prentice-hall Englewood Cliffs, NJ, 1991.

[80] Jeffry T Russell and Margarida F Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. In *Computer Design: VLSI in Computers and Processors, 1998. ICCD'98. Proceedings. International Conference on*, pages 328–333. IEEE, 1998.

[81] Cagri Sahin, Furkan Cayci, Irene Lizeth Manotas Gutiérrez, James Clause, Fouad Kiamilev, Lori Pollock, and Kristina Winbladh. Initial explorations on design pattern energy usage. In *Green and Sustainable Software (GREENS), 2012 First International Workshop on*, pages 55–61. IEEE, 2012.

[82] Alex Shye, Benjamin Scholbrock, and Gokhan Memik. Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 168–178. ACM, 2009.

[83] Mahendra Pratap Singh and Manoj Kumar Jain. Evolution of processor architecture in mobile phones. *International Journal of Computer Applications*, 90(4), 2014.

[84] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):437–445, 1994.

[85] Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike Tien-Chien Lee. Instruction level power analysis and optimization of software. In *Technologies for wireless computing*, pages 139–154. Springer, 1996.

[86] Antonio Vetro, Luca Ardito, Giuseppe Procaccianti, and Maurizio Morisio. Definition, implementation and validation of energy code smells: an exploratory study on an embedded system. 2013.

[87] Chao Wang, Wei Duan, Jianzhang Ma, and Chenhui Wang. The research of android system architecture and application programming. In *Computer Science and Network Technology (ICCSNT), 2011 International Conference on*, volume 2, pages 785–790. IEEE, 2011.

[88] Elaine J Weyuker. Evaluating software complexity measures. *Software Engineering, IEEE Transactions on*, 14(9):1357–1365, 1988.

[89] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 105–114. ACM, 2010.

# Appendix A

# Persistence Framework Libraries

This appendix introduces the persistence framework libraries used in our Android benchmark implementations.

1. activeandroid-3.1.0-SNAPSHOT.jar
2. greendao-2.1.0.jar
3. ormlite-android-4.49-SNAPSHOT.jar
4. ormlite-core-4.49-SNAPSHOT.jar
5. ormlite-jdbc-4.49-SNAPSHOT.jar
6. android.jar (SDK level 23)
7. sugar-1.4.jar
8. realm-android-library-0.88.3.jar
9. realm-annotations-0.88.3.jar

From the above list, 1 is ActiveAndroid library, 2 is greenDAO library, 3-5 are OrmLite libraries, 6 provides Android SQLite package, 7 is Sugar ORM library, and 8-9 are Realm Java libraries.

# Appendix B

# Initialization Core Code

This appendix displays the core code snippets of initialization implementation. We omitted irrelevant code, and only show the programming pattern differences when using six ORM/OO Android persistence frameworks to achieve the same functionality. Here we mainly focus on database context initialization patten and database table creation manner.

1 ActiveAndroid Initialization

```
public void initialize() throws Exception {
   // delete obsolete database

   // database context and configuration initialization
   Configuration dbConfiguration = new
       Configuration.Builder(AppContext.getInstance())
      .setDatabaseName(BaseBenchmark.DATABASE_NAME).create();
   ActiveAndroid.initialize(dbConfiguration);

   // automatically create tables from entities
}
```

2 greenDAO Initialization

```
public void initialize() throws Exception {
   if (null == ga) {
      throw new NullPointerException();
```

```
4     }
5
6     // delete obsolete database
7
8     // create database tables
9     SQLiteDatabase db = ga.getDatabase();
10    DaoMaster daoMaster = ga.getDaoMaster();
11    daoMaster.createAllTables(db, true);
12 }
```

3 OrmLite Initialization

```
1  public void initialize() throws Exception {
2     if (null == oa) {
3        throw new NullPointerException();
4     }
5
6     // delete obsolete database
7
8     // create database tables
9     ConnectionSource connectionSource = oa.getDbHelper()
          .getConnectionSource();
10    TableUtils.createTable(connectionSource, Category.class);
11    TableUtils.createTable(connectionSource, CItem.class);
12 }
```

4 Android SQLite Initialization

```
1  public void initialize() throws Exception {
2     if (null == sa) {
3        throw new NullPointerException();
4     }
5
6     // delete obsolete database
7
8     // create database tables
9     SQLiteDatabase db = sa.getWritableDatabase();
10    // CATEGORY
11    String CREATE_TABLE_CATEGORY = "CREATE TABLE CATEGORY
12       (C_ID INTEGER PRIMARY KEY,C_TITLE TEXT NOT NULL,C_PAGES
```

```
13      INTEGER NOT NULL,C_SUBCATS INTEGER NOT NULL,C_FILES
14      INTEGER NOT NULL);";
15   db.execSQL(CREATE_TABLE_CATEGORY);
16
17   // CITEM
18   String CREATE_TABLE_CITEM = "CREATE TABLE CITEM (I_ID
19      INTEGER PRIMARY KEY, I_CAT_ID INTEGER NOT NULL,I_IM_ID
20      INTEGER NOT NULL,I_NAME TEXT NOT NULL,I_PRICE DECIMAL
21      (5,2) NOT NULL,I_DATA TEXT NOT NULL, FOREIGN KEY
22      (I_CAT_ID) REFERENCES CATEGORY(C_ID));";
23   db.execSQL(CREATE_TABLE_CITEM);
24 }
```

5 Realm Java Initialization

```
1  public void initialize() throws Exception {
2     // database context and configuration initialization, and delete
          obsolete database
3     RealmConfiguration config = new RealmConfiguration
4        .Builder(AppContext.getInstance()).deleteRealmIfMigrationNeeded()
5        .build();
6     Realm.deleteRealm(config);
7  }
```

6 Sugar ORM Initialization

```
1  public void initialize() throws Exception {
2     // delete old database
3
4     // database context and configuration initialization
5     SugarContext.init(AppContext.getInstance());
6  }
```

# Appendix C

# Insert Core Code

This appendix displays the core code snippets of insert implementation. We omitted irrelevant code, and only show the programming pattern differences when using six ORM/OO Android persistence frameworks to achieve the same functionality. Here we compare the batch insert, and transaction features provided by each persistence framework.

1 ActiveAndroid Insert

```
private void categoryAndItemTable() throws Exception {
   Category cg = null;

   // use transaction to batch insert entities to database
   ActiveAndroid.beginTransaction();
   for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
      cg = new Category();
      cg.setLCid(c);
      cg.setStrCTitle(random.randomAString26_50());
      cg.setiCPages(random.randomInt(1, 10000));
      cg.setiCSubCats(random.randomInt(1, 5000));
      cg.setiCFiles(random.randomInt(1, 20000));
      cg.save();
   }

   ActiveAndroid.setTransactionSuccessful();
   ActiveAndroid.endTransaction();
```

```
}
```

2 greenDAO Insert

```
private void categoryAndItemTable() throws Exception {
   if (null == ga) {
      throw new NullPointerException();
   }

   // create new entities and add to list
   List<Category> cList = new ArrayList<Category>();
   Category cg = null;

   for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
      cg = new Category();
      cg.setId((long) c);
      cg.setStrCTitle(random.randomAString26_50());
      cg.setICPages(random.randomInt(1, 10000));
      cg.setICSubCats(random.randomInt(1, 5000));
      cg.setICFiles(random.randomInt(1, 20000));
      cList.add(cg);
   }

   // batch insert entities to database
   SQLiteDatabase db = ga.getDatabase();
   DaoSession ds = ga.getDaoSession();
   ds.getCategoryDao().insertInTx(cList);
}
```

3 OrmLite Insert

```
private void categoryAndItemTable() throws Exception {
   if (null == oa) {
      throw new NullPointerException();
   }

   final OrmliteDBHelper dbHelper = oa.getDbHelper();
   final OERandom random = this.random;

   // use transaction to batch insert entities to database
```

```
10    TransactionManager.callInTransaction(dbHelper.getConnectionSource(),
          new Callable<Void>() {
11        public Void call() throws Exception {
12           OrmliteDBHelper dbHelper = oa.getDbHelper();
13           Dao<Category, Long> cgDao = dbHelper.getCategoryDao();
14           Category cg = null;
15
16           for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
17              cg = new Category();
18              cg.setlCid(c);
19              cg.setStrCTitle(random.randomAString26_50());
20              cg.setiCPages(random.randomInt(1, 10000));
21              cg.setiCSubCats(random.randomInt(1, 5000));
22              cg.setiCFiles(random.randomInt(1, 20000));
23              cgDao.create(cg);
24           }
25
26           return null;
27        }
28    });
29 }
```

4 Android SQLite Insert

```
1  private void categoryAndItemTable() throws Exception {
2     if (null == sa) {
3        throw new NullPointerException();
4     }
5
6     // use transaction to batch insert entities to database
7     SQLiteDatabase db = sa.getWritableDatabase();
8     ContentValues values = new ContentValues();
9     sa.beginTransaction(db);
10
11    for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
12       values.clear();
13       values.put(Category.COL_C_ID, c);
14       values.put(Category.COL_C_TITLE, this.random.randomAString26_50());
15       values.put(Category.COL_C_PAGES, this.random.randomInt(1, 10000));
16       values.put(Category.COL_C_SUBCATS, this.random.randomInt(1, 5000));
```

```
17        values.put(Category.COL_C_FILES, this.random.randomInt(1, 20000));
18
19        // Insert record to databse
20        db.insert(Category.TABLE, null, values);
21     }
22
23     sa.setTransactionSuccessful(db);
24     sa.endTransaction(db);
25  }
```

5 Realm Java Insert

```
1  private void categoryAndItemTable() throws Exception {
2     if (null == ra) {
3        throw new NullPointerException();
4     }
5
6     Category cg = null;
7     Realm realm = null;
8     // use transaction to batch insert entities to database
9     try {
10       realm = ra.getRealmInstance();
11
12       // transaction is necessary for modification operation
13       realm.beginTransaction();
14
15       for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
16          cg = realm.createObject(Category.class);
17          cg.setlCid(c);
18          cg.setStrCTitle(random.randomAString26_50());
19          cg.setiCPages(random.randomInt(1, 10000));
20          cg.setiCSubCats(random.randomInt(1, 5000));
21          cg.setiCFiles(random.randomInt(1, 20000));
22       }
23
24       realm.commitTransaction();
25    } finally {
26       if (null != realm) {
27          // explicityly close database connection
28          realm.close();
```

```
29        }
30    }
31 }
```

6 Sugar ORM Insert

```
1  private void categoryAndItemTable() throws Exception {
2     List<Category> cList = new ArrayList<Category>();
3     Category cg = null;
4     for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
5        cg = new Category();
6        cg.setId((long)c);
7        cg.setStrCTitle(random.randomAString26_50());
8        cg.setiCPages(random.randomInt(1, 10000));
9        cg.setiCSubCats(random.randomInt(1, 5000));
10       cg.setiCFiles(random.randomInt(1, 20000));
11       cList.add(cg);
12    }
13
14    // batch insert entities to database
15    Category.saveInTx(cList);
16 }
```

# Appendix D

# Update Core Code

This appendix displays the core code snippets of update implementation. We omitted irrelevant code, and only show the programming pattern differences when using six ORM/OO Android persistence frameworks to achieve the same functionality. Here we presents the update interface, and batch update support of persistence frameworks.

1 ActiveAndroid Update

```
1  public void update() throws Exception {
2     Category cg = null;
3     CItem cItem = null;
4
5     // use transaction to batch update entities
6     for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
7        ActiveAndroid.beginTransaction();
8        // find and update Category entities
9        cg = Category.load(Category.class, c);
10       cg.setStrCTitle(this.random.randomAString26_50());
11       cg.setiCPages(this.random.randomInt(1, 10000));
12       cg.setiCSubCats(this.random.randomInt(1, 5000));
13       cg.setiCFiles(this.random.randomInt(1, 20000));
14       cg.save();
15
16       // update CItem entities
17       new Update(CItem.class).set("I_IM_ID = ?, I_NAME=?, I_PRICE=?,
```

```
            I_DATA=?",
18          this.random.randomInt(1, 10000),
19          this.random.randomAString14_24(),
20          this.random.randomDecimalString(100, 9999, 2),
21          this.random.randomData())
22          .where("Category = ?", c).execute();
23
24          ActiveAndroid.setTransactionSuccessful();
25          ActiveAndroid.endTransaction();
26      }
27  }
```

2 greenDAO Update

```
1   public void update() throws Exception {
2       if (null == ga) {
3           throw new NullPointerException();
4       }
5
6       // obtain entity DAO, and use query builder to build SQL-like entity
            query statements
7
8       SQLiteDatabase db = ga.getDatabase();
9       DaoSession ds = ga.getDaoSession();
10      CategoryDao cDao = ds.getCategoryDao();
11      CItemDao iDao = ds.getCItemDao();
12      QueryBuilder cQb = null;
13      QueryBuilder iQb = null;
14
15      List<Category> cList = null;
16      List<CItem> iList = null;
17
18      String strCTitle = null;
19      int iCPages = 0;
20      int iCSubCats = 0;
21      int iCFiles = 0;
22      int iImId = 0;
23      String strIName = null;
24      float fIPrice = 0;
25      String strIData = null;
```

```
26
27    for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
28       // find Category entities and update values
29       cQb = cDao.queryBuilder();
30       cQb.where(CategoryDao.Properties.Id.eq((long)c));
31       cList = cQb.list();
32
33       strCTitle = this.random.randomAString26_50();
34       iCPages = this.random.randomInt(1, 10000);
35       iCSubCats = this.random.randomInt(1, 5000);
36       iCFiles = this.random.randomInt(1, 20000);
37
38       if (null != cList) {
39          for (Category category : cList) {
40             category.setStrCTitle(strCTitle);
41             category.setICPages(iCPages);
42             category.setICSubCats(iCSubCats);
43             category.setICFiles(iCFiles);
44          }
45       }
46
47       // batch update Category entities
48       cDao.updateInTx(cList);
49
50       // find Category entities and update values
51       iQb = iDao.queryBuilder();
52       iQb.where(CItemDao.Properties.CId.eq((long)c));
53
54       iList = iQb.list();
55       iImId = this.random.randomInt(1, 10000);
56       strIName = this.random.randomAString14_24();
57       fIPrice = Float.parseFloat(this.random.randomDecimalString(100,
             9999, 2));
58       strIData = this.random.randomData();
59
60       if (null != iList) {
61          for (CItem cItem : iList) {
62             cItem.setIImId(iImId);
63             cItem.setIName(strIName);
64             cItem.setIPrice(fIPrice);
65             cItem.setIData(strIData);
```

```
66          }
67       }
68
69       // batch update CItem entities
70       iDao.updateInTx(iList);
71    }
72 }
```

3 OrmLite Update

```
1  public void update() throws Exception {
2     if (null == oa) {
3        throw new NullPointerException();
4     }
5
6     final OrmliteDBHelper dbHelper = oa.getDbHelper();
7     final OERandom random = this.random;
8     // obtain entity DAO
9     final Dao<Category, Long> cgDao = dbHelper.getCategoryDao();
10    final Dao<CItem, Long> cItemDao = dbHelper.getCItemDao();
11
12    for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
13       // use transaction to batch update entities
14       final int cValue = c;
15       TransactionManager.callInTransaction(dbHelper.getConnectionSource(),
             new Callable<Void>() {
16          public Void call() throws Exception {
17             // find and update Category entities
18             Category category = cgDao.queryForId((long)cValue);
19             category.setStrCTitle(random.randomAString26_50());
20             category.setiCPages(random.randomInt(1, 10000));
21             category.setiCSubCats(random.randomInt(1, 5000));
22             category.setiCFiles(random.randomInt(1, 20000));
23             cgDao.update(category);
24
25             // use update builder to update CItem entities
26             UpdateBuilder<CItem, Long> ciUb = cItemDao.updateBuilder();
27             Where<CItem, Long> ciWhere = ciUb.where();
28             ciWhere.eq(CItem.COL_I_CAT_ID, cValue);
29             ciUb.updateColumnValue(CItem.COL_I_IM_ID,
```

```
                random.randomInt(1, 10000));
30          ciUb.updateColumnValue(CItem.COL_I_NAME,
                random.randomAString14_24());
31          ciUb.updateColumnValue(CItem.COL_I_PRICE,
                Float.parseFloat(random.randomDecimalString(100, 9999,
                2)));
32          ciUb.updateColumnValue(CItem.COL_I_DATA,
                random.randomData());
33          ciUb.update();
34
35          return null;
36        }
37      });
38    }
39  }
```

4 Android SQLite Update

```
1  public void update() throws Exception {
2    if (null == sa) {
3      throw new NullPointerException();
4    }
5
6    SQLiteDatabase db = sa.getWritableDatabase();
7    ContentValues values = new ContentValues();
8    String iWhereClause = CItem.COL_I_CAT_ID + " = ?";;
9    String cWhereClause = Category.COL_C_ID + " = ?";
10   String[] whereArgs = null;
11
12   for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
13     // use transaction to batch update entities
14     sa.beginTransaction(db);
15
16     whereArgs = new String[]{ String.valueOf(c) };
17     // update Category entities
18     values.clear();
19     values.put(Category.COL_C_TITLE, this.random.randomAString26_50());
20     values.put(Category.COL_C_PAGES,
           String.valueOf(this.random.randomInt(1, 10000)));
21     values.put(Category.COL_C_SUBCATS,
```

```
            String.valueOf(this.random.randomInt(1, 5000)));
        values.put(Category.COL_C_FILES,
            String.valueOf(this.random.randomInt(1, 20000)));
        db.update(Category.TABLE, values, cWhereClause, whereArgs);

        // update CItem entities
        values.clear();
        values.put(CItem.COL_I_IM_ID, this.random.randomInt(1, 10000));
        values.put(CItem.COL_I_NAME, this.random.randomAString14_24());
        values.put(CItem.COL_I_PRICE, this.random.randomDecimalString(100,
            9999, 2));
        values.put(CItem.COL_I_DATA, this.random.randomData());
        db.update(CItem.TABLE, values, iWhereClause, whereArgs);

        sa.setTransactionSuccessful(db);
        sa.endTransaction(db);
    }
}
```

5 Realm Java Update

```
public void update() throws Exception {
    if (null == ra) {
        throw new NullPointerException();
    }

    Realm realm = null;
    try {
        realm = ra.getRealmInstance();

        Category cg = null;
        CItem cItem = null;
        RealmResults<Category> cs = null;
        RealmResults<CItem> is = null;

        String StrCTitle = null;
        int iCPages = 0;
        int iCSubCats = 0;
        int iCFiles =0;
        int iIImId = 0;
```

```
20      String StrIName = null;
21      float fIPrice = 0;
22      String strIData = null;
23
24      for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
25         // find Category and CItem entities
26         cs = realm.where(Category.class)
27         .equalTo("lCid", c).findAll();
28
29         is = realm.where(CItem.class)
30         .equalTo("category.lCid", c).findAll();
31
32         // transaction is necessary for modification, use transaction
               to batch update entities
33         realm.beginTransaction();
34
35         // update Category entities
36         StrCTitle = this.random.randomAString26_50();
37         iCPages = this.random.randomInt(1, 10000);
38         iCSubCats = this.random.randomInt(1, 5000);
39         iCFiles = this.random.randomInt(1, 20000);
40
41         for (int k=0; k<cs.size(); k++) {
42            cg = cs.get(k);
43            cg.setStrCTitle(StrCTitle);
44            cg.setiCPages(iCPages);
45            cg.setiCSubCats(iCSubCats);
46            cg.setiCFiles(iCFiles);
47         }
48
49         // update CItem entities
50         iIImId = this.random.randomInt(1, 10000);
51         StrIName = this.random.randomAString14_24();
52         fIPrice = Float.parseFloat(this.random.randomDecimalString(100,
               9999, 2));
53         strIData = this.random.randomData();
54
55         for (int k=0; k<is.size(); k++) {
56            cItem = is.get(k);
57            cItem.setiIImId(iIImId);
58            cItem.setStrIName(StrIName);
```

```
59              cItem.setfIPrice(fIPrice);
60              cItem.setiData(strIData);
61           }
62
63           realm.commitTransaction();
64        }
65     } finally {
66        if (null != realm) {
67           // explicityly close database connection
68           realm.close();
69        }
70     }
71  }
```

6 Sugar ORM Update

```
1  public void update() throws Exception {
2     Category cg = null;
3     List<CItem> iList = null;
4     String strCTitle = null;
5     int iCPages = 0;
6     int iCSubCats = 0;
7     int iCFiles = 0;
8     int iIImId = 0;
9     String StrIName = null;
10    float fIPrice = 0;
11    String strIData = null;
12
13    for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
14       // find and update Category entities
15       cg = Category.findById(Category.class, c);
16
17       strCTitle = this.random.randomAString26_50();
18       iCPages = this.random.randomInt(1, 10000);
19       iCSubCats = this.random.randomInt(1, 5000);
20       iCFiles = this.random.randomInt(1, 20000);
21
22       cg.setStrCTitle(strCTitle);
23       cg.setiCPages(iCPages);
24       cg.setiCSubCats(iCSubCats);
```

```
25        cg.setiCFiles(iCFiles);
26        Category.save(cg);
27
28        // find and update the values of CItem entities
29        iList = CItem.find(CItem.class, "category = ? ",
              String.valueOf(c));
30
31        iIImId = this.random.randomInt(1, 10000);
32        StrIName = this.random.randomAString14_24();
33        fIPrice = Float.parseFloat(this.random.randomDecimalString(100,
              9999, 2));
34        strIData = this.random.randomData();
35
36        for (CItem cItem : iList) {
37           cItem.setiIImId(iIImId);
38           cItem.setStrIName(StrIName);
39           cItem.setfIPrice(fIPrice);
40           cItem.setiData(strIData);
41        }
42        // batch update CItem entities
43        CItem.saveInTx(iList);
44     }
45  }
```

# Appendix E

# Select Core Code

This appendix displays the core code snippets of select implementation. We omitted irrelevant code, and only show the programming pattern differences when using six ORM/OO Android persistence frameworks to achieve the same query functionality.

1 ActiveAndroid Select

```
public void select() throws Exception {
   Category category = null;
   List<CItem> ciList = null;

   for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
      // query Category entities
      Category cg = Category.load(Category.class, c);

      // query CItem entities
      ciList = new Select().from(CItem.class).where("Category = ?",
          c).execute();
      if (null != ciList) {
         for (CItem cItem: ciList) {
         }
      }
   }
}
```

2 greenDAO Select

```
public void select() throws Exception {
   if (null == ga) {
      throw new NullPointerException();
   }

   // obtain entity DAO
   SQLiteDatabase db = ga.getDatabase();
   DaoSession ds = ga.getDaoSession();
   CategoryDao cDao = ds.getCategoryDao();
   CItemDao iDao = null;
   // use query builder to build query statement
   QueryBuilder cQb = null;
   QueryBuilder iQb = null;
   List<Category> cList = null;
   List<CItem> iList = null;
   Category category = null;
   int size = 0;

   for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
      // query Category entities
      cQb = cDao.queryBuilder();
      cQb.where(CategoryDao.Properties.Id.eq((long)c));
      cList = cQb.limit(1).list();

      if (null != cList) {
         for (Category cg : cList) {
         }
      }

      // query CItem entities
      iDao = ds.getCItemDao();
      iQb = iDao.queryBuilder();
      iQb.where(CItemDao.Properties.CId.eq((long)c));
      iList = iQb.list();
      if (null != iList) {
         for (CItem cItem : iList) {
         }
      }
   }
}
```

3 OrmLite Select

```
public void select() throws Exception {
   if (null == oa) {
      throw new NullPointerException();
   }

   OrmliteDBHelper dbHelper = oa.getDbHelper();
   // obtain entity DAO
   Dao<Category, Long> cDao = dbHelper.getCategoryDao();
   Dao<CItem, Long> cItemDao = dbHelper.getCItemDao();
   Category category = null;
   List<CItem> ciList = null;

   for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
      // query Category entities
      category = cDao.queryForId((long)c);
      if (null != category) {
      }

      // query CItem entities
      ciList = cItemDao.queryForEq(CItem.COL_I_CAT_ID, c);
      if (null != ciList) {
         for (CItem cItem : ciList) {
         }
      }
   }
}
```

4 Android SQLite Select

```
public void select() throws Exception {
   if (null == sa) {
      throw new NullPointerException();
   }

   SQLiteDatabase db = sa.getReadableDatabase();
   String[] cColumns = new String[]{ Category.COL_C_TITLE,
```

```
        Category.COL_C_PAGES, Category.COL_C_SUBCATS,
        Category.COL_C_FILES };
    String cWhereClause = Category.COL_C_ID + " = ?";

    String[] iColumns = new String[]{ CItem.COL_I_ID, CItem.COL_I_IM_ID,
        CItem.COL_I_NAME, CItem.COL_I_PRICE, CItem.COL_I_DATA };
    String iWhereClause = CItem.COL_I_CAT_ID + " = ?";

    String[] whereArgs = null;
    Cursor cursor = null;
    Category category = new Category();
    CItem cItem = new CItem();

    for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
       // query Category entities
       whereArgs = new String[]{String.valueOf(c)};
       cursor = db.query(Category.TABLE, cColumns, cWhereClause,
           whereArgs, null, null, null);
       if (cursor.moveToFirst()) {
       }
       cursor.close();

       // query CItem entities
       cursor = db.query(CItem.TABLE, iColumns, iWhereClause, whereArgs,
           null, null, null);
       if (cursor.moveToFirst()) {
          do {
          } while (cursor.moveToNext());
       }

       cursor.close();
    }
}
```

5 Realm Java Select

```
public void select() throws Exception {
    if (null == ra) {
       throw new NullPointerException();
    }
```

```
5
6     Realm realm = null;
7     try {
8        realm = ra.getRealmInstance();
9        Category category = null;
10       RealmResults<CItem> is = null;
11
12       for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
13          // query Category entities
14          category = realm.where(Category.class).equalTo("lCid",
                c).findFirst();
15          if (null != category) {
16          }
17
18          // query CItem entities
19          is = realm.where(CItem.class)
20          .equalTo("category.lCid", c).findAll();
21          if (null != is) {
22             for (CItem cItem : is) {
23             }
24          }
25       }
26    }  finally {
27       if (null != realm) {
28          // explicityly close database connection
29          realm.close();
30       }
31    }
32 }
```

6 Sugar ORM Select

```
1  public void select() throws Exception {
2     List<CItem> iList = null;
3     Category category = null;
4
5     for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
6        // query Category entities
7        category = Category.findById(Category.class, c);
8        if (null != category) {
```

```
9         }
10
11        // query CItem entities
12        iList = CItem.find(CItem.class, "category = ? ",
              String.valueOf(c));
13        if (null != iList) {
14           for (CItem cItem : iList) {
15           }
16        }
17     }
18  }
```

# Appendix F

# Delete Core Code

This appendix displays the core code snippets of select implementation. We omitted irrelevant code, and only show the programming pattern differences when using six ORM/OO Android persistence frameworks to achieve the same functionality. Here we compare the delete mode, and batch delete support of each persistence framework.

1 ActiveAndroid Delete

```
public void delete() throws Exception {
   for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
      // delete CItem entities
      new Delete().from(CItem.class).where("Category = ?", c).execute();

      // delete Category entities
      Category.delete(Category.class, c);
   }
}
```

2 greenDAO Delete

```
public void delete() throws Exception {
   if (null == ga) {
      throw new NullPointerException();
   }

   // obtain entity DAO
```

```
7     SQLiteDatabase db = ga.getDatabase();
8     DaoSession ds = ga.getDaoSession();
9
10    for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
11       // use query buidler build query statement, find CItem entities,
             then delete CItem entities
12       ds.getCItemDao().queryBuilder().where(CItemDao.Properties.CId
13          .eq((long)c)).buildDelete()
14          .executeDeleteWithoutDetachingEntities();
15
16       // use query buidler build query statement, find Category
             entities, then delete Category entities
17       ds.getCategoryDao().queryBuilder().where(CategoryDao.Properties.Id
18          .eq(c)).buildDelete().executeDeleteWithoutDetachingEntities();
19    }
20 }
```

3 OrmLite Delete

```
1  public void delete() throws Exception {
2     if (null == oa) {
3        throw new NullPointerException();
4     }
5
6     // obtain entity DAO
7     OrmliteDBHelper dbHelper = oa.getDbHelper();
8     Dao<CItem, Long> cItemDao = dbHelper.getCItemDao();
9     Dao<Category, Long> cDao = dbHelper.getCategoryDao();
10
11    DeleteBuilder<CItem, Long> idb = null;
12    Where<CItem, Long> iWhere = null;
13    Category category = null;
14
15    for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
16       // use delete builder build delete statemnt, delete CItem entities
17       idb = cItemDao.deleteBuilder();
18       iWhere = idb.where();
19       iWhere.eq(CItem.COL_I_CAT_ID, c);
20       idb.delete();
21
```

```
22      // find Category entitiy and delete it
23      category = cDao.queryForId((long)c);
24      if (null != category) {
25         cDao.delete(category);
26      }
27   }
28}
```

4 Android SQLite Delete

```
1  public void delete() throws Exception {
2     if (null == sa) {
3        throw new NullPointerException();
4     }
5
6     SQLiteDatabase db = sa.getWritableDatabase();
7
8     for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
9        // delete CItem entities
10       db.delete(CItem.TABLE, CItem.COL_I_CAT_ID + "=?", new
             String[]{String.valueOf(c)});
11
12       // delete Category entities
13       db.delete(Category.TABLE, Category.COL_C_ID + "=?", new
             String[]{String.valueOf(c)});
14    }
15 }
```

5 Realm Java Delete

```
1  public void delete() throws Exception {
2     Realm realm = null;
3
4     try {
5        realm = ra.getRealmInstance();
6
7        RealmResults<CItem> is = null;
8        RealmResults<Category> cs = null;
9        for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
10          // transaction is necessary for modification operation
```

```
        realm.beginTransaction();

        // find and delete CItem entities
        is = realm.where(CItem.class)
        .equalTo("category.lCid", c).findAll();
        is.clear();

        // find and delete Category entities
        cs = realm.where(Category.class).equalTo("lCid", c).findAll();
        cs.clear();

        realm.commitTransaction();
      }
   } finally {
      if (null != realm) {
        // explicityly close database connection
        realm.close();
      }
   }
}
```

6 Sugar ORM Delete

```
public void delete() throws Exception {
   Category category = null;
   List<CItem> iList = new ArrayList<CItem>();

   for (int c = 1; c <= AppContext.getInstance().getScale(); c++) {
      // find CItem entities
      iList = CItem.find(CItem.class, "category = ? ",
          String.valueOf(c));
      // batch delete CItem entities
      CItem.deleteInTx(iList);

      // find and delete Category entities
      category = Category.findById(Category.class, c);
      Category.delete(category);
   }
}
```