ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Proyecto Fin de Carrera

rSQLite – A relational SQLite Manager for Android

Para acceder al Titulo de

INGENIERO TÉCNICO DE TELECOMUNICACIÓN

Autor: Pedro Enrique Fernández Gutiérrez

Octubre - 2014

E.T.S. DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACION

INGENIERÍA TÉCNICA DE TELECOMUNICACIÓN

CALIFICACIÓN DEL PROYECTO FIN DE CARRERA

Realizado por: Pedro Enrique Fernández Gutiérrez
Director del PFC: Esteban Stafford Fernández
Título: "rSQLite – Un editor SQLite relacional para Android "
Title: " rSQLite – A relational SQLite Manager for Android"

Presentado a examen el día: 31 Octubre de 2014

para acceder al Título de

INGENIERO TÉCNICO DE TELECOMUNICACIÓN, ESPECIALIDAD EN SISTEMAS ELECTRÓNICOS

Composición del Tribunal:
Presidente (Apellidos, Nombre): Martínez Fernández, Carmen
Secretario (Apellidos, Nombre): Stafford Fernández, Esteban
Vocal (Apellidos, Nombre): Junquera Quintana, Francisco Javier

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente

Fdo.: El Secretario

Fdo.: El Vocal

Fdo.: El Director del PFC
(sólo si es distinto del Secretario)

Vº Bº del Subdirector

Proyecto Fin de Carrera Nº
(a asignar por Secretaría)

Abstract

As the use of mobile devices in the workplace grows exponentially, the current SQLite database managers available were too simple and failed to present the user with an intuitive way to browse and edit data. Most applications simply listed the content of the tables, ignoring the relations between them. This missing capability is what motivated this project. To create an app capable of managing a SQLite database focusing on the relations between tables.

After selecting Android as the target operating system for its huge market share and how easily new developers can start to create projects, the application was developed using Java and Eclipse. Following the study of the way the SQLite language creates and saves the relations between tables, the final code is able to understand these relations and present an intuitive user interface. In order to experience the full capabilities of the application, the database design must follow a short and simple list of conventions, but the relational functionalities can also be disabled, in which case the application works as a simple editor for any SQLite file.

The result is an innovative, intuitive and fast SQLite Manager that allows the user to navigate through the database tables using the relations between them as well as making all the other basic activities such as creating new entries or editing existing ones. In conclusion, users who actively work with SQLite databases will now have a better and easier to use tool that will improve their experience with mobile devices in the workplace.

Contents

Chapter 1

Introduction

The increased popularity of the smartphones in the last 7 years has made a significant change in the mobile applications market. Before the introduction of mobile operating systems, the applications were developed by the same company that made the phone, and they were usually not compatible between different models. But the new operating systems that allowed third-party software into the smartphones created a new market that has grown parallel to the smartphone market for the last few years. The market of mobile applications is expected to be worth $143 Bn in 2016.[1]

This market has numerous categories such as games, communication, health, sports or weather but the app developed in this project will be categorized as a work productivity app. With the constant improvement of mobile devices, developers realized that some applications which previously belonged only to the PC could also be used in smartphones or tablets. Examples could be email clients, word processors and spreadsheets or presentation apps. These applications allow the user to work remotely and to even sync the files to internet servers, so their work is always up to date and secure. This category includes the database managers, which allow the user to manage remote or local databases of different technologies with a mobile device.

A definition of the word "database" is simply "A collection of data arranged for ease and speed of search and retrieval" [2]. With the growth of the processing power of computers, the amount of information stored in databases has grown larger and different technologies have evolvedr. But the main target has not changed: to store as much information as possible and to make it accessible as fast as possible with the least possible memory use. To make the system more efficient, the information is stored in different tables with rows and columns. Each table is a group of rows of the same type of data and can be related to other rows in different tables.

Usually, databases are seen as huge collections of information like for example, the database of a bank or the database of an important website, but there are also smaller databases like the ones in some mobile applications or even personal ones. The first kind of database almost always rely on a database server which stores the information, receives the queries and sends the responses. This is the most usual type of database and MySQL or PostgreSQL are some examples of this kind of technology. On the other hand, sometimes the size of the database is so small that it does not require a dedicated database server and all the information is stored in a single file. SQLite is an example of this small size, serverless database. It is used in most mobile applications and can also be used as a simple personal database.

As stated before, work productivity mobile applications have experienced a great growth in the last years and this project will be focused on SQLite Managers for Android. These apps can be used by developers trying to browse or edit databases of applications in the same device, or by users that work with personal SQLite databases. In this case, Android was selected as the operating system to work with, because of the market share advantage [3] that has over iOS and because of how easy it is for new developers to start creating new apps.

After testing the most popular Android SQLite Managers, it was obvious that all of them were

too simple and at the same time difficult to use, and none of them used the relations of the tables to improve the user experience. Therefore the goal of this project is to develop a new app that will allow the user to navigate through the database tables using the relations between them as well as making all the other basic activities such as creating new entries or editing existing ones.

This document will detail the process followed for the development of the app. In the chapter 2 the different technologies used in this project will be explained. Starting the evolution of the mobile devices in section 2.1, followed by the Java and Eclipse IDE technologies in sections 2.2 and 2.3, an explanation on the Android architecture in section 2.4 and in section 2.5 there is a description of the SQL language and the SQLite technology. Chapter 3 explains the primary and secondary objectives of the application, the activities flowchart and the requirements of the used databases. The proper development of each activity in the application is closely explained in chapter 4. Finally, chapter 5 explains the different testing environments and chapter 6 presents the results of the project along with some possible future improvements.

Chapter 2

Technology overview

2.1 Smartphones

Just as it happened with PC's, laptops, and almost every other computer device, the cell phone technology evolved from the huge brick phones of the 80's to tiny but durable and cheaper cell phones in the early 2000's. But it has been in the last 6 or 7 years, when the evolution of the cell phones has been exponential. Before 2007, the smartphone was just some futuristic phone, which was capable of instant messaging, sending and receiving email and maybe some basic internet browsing, as well as making phone calls. The launch of the iPhone in 2007 resulted in customers starting to talk about things like apps, specifications or display sizes, just as if they were talking about ordinary computers.

The first iPhone, launched in 2007, had a 415 MHz processor, 128 MB of RAM, a 480x320 pixel display and a 2MP camera [4]. Nowadays, only 6 years after, there are devices with quad-core processors running at more than 2GHz, and with up to 3 GB of RAM, FullHD displays and 13 MP cameras. The power of these phones can be equivalent to low-end personal computers with the advantages of both the size, as they will fit in the users pocket, and the cost, since smartphones are usually cheaper than personal computers.

But not only the hardware has improved. With the creation of the smartphone, new mobile operating systems were necessary. Like with any other new technology, at the beginning there were multiple choices such as Palm OS, Web OS, Windows Phone, Symbian... etc, but nowadays there are only four mayor mobile operating systems: Blackberry OS, Windows RT, iOS and Android.

One definition of an operating system, could be: "It is the software designed to control the hardware, in order to allow users and application programs to make use of it" [2]. The old cell phones had proprietary software that allowed the user to make use of the utilities that came pre-installed on the phone. Nowadays, the new mobile operating systems have created a solid base for developers to create all kinds of software applications, also know as apps, which are usually available through application distribution platforms. These are typically operated by the owner of the operating system, such as the Apple App Store, for iOS apps, or the Play Store, for Android apps. The presence of these new smartphones in society has grown in such way, that the term "app" was listed in 2010 as "Word of the Year" by the American Dialect Society [5].

Of the four current biggest mobile operating systems previously listed, only two of them, iOS and Android represent more than 95% of the market share, and as of the third quarter of 2013, Android has grown up to 85% of the total market share [3]. This is clearly one of the biggest appeals for developers who have a project idea but do not know which platform to choose. Another great advantage over iOS is that, at an entry level, the development kit is easier to download and install, and writing and testing apps in the user's phone is very straightforward.

Unlike iOS, which is only used by Apple in a small number of devices, Android is used by lots

of smartphone companies that build very different devices, aiming at very different market groups. This requires Android to be a very flexible OS. But this requirement also affects the applications, therefore, makes it very difficult to have all the devices updated to the latest OS version. Each big Android smartphone maker has its own user interface running over the stock OS, which delays the OS updates from Google. And there might also be makers who decide not to update an old device, so customers buy the newer devices with the latest version of the OS. That, from the developers point of view, means than they must decide between using the latest API and leaving some users out, or make an app without the latest OS features, in order to get it to a broader audience.

2.2 Java

Java is an object-oriented programming language introduced by Sun Microsystems. James Gosling and his team developed Java in 1995. Most of the syntax in Java comes from the C and C++ programming languages but is simpler and easier to use when compared to them. Once compiled into byte-code, java programs can be run on any device capable of running the Java Virtual Machine (JVM) and they do not need to be recompiled. The JVM is a virtual machine which executes Java Byte-code files. Nowadays, Java runs on about 850 million computers and 3 billion mobile devices worldwide. In 2006, Sun Microsystems made Java free and open source. [12]

There are many features that make Java a powerful and popular programming language, but the most important are:

- The presence of a compiler and interpreter
- Platform independence
- Object oriented
- Secure
- Multi thread
- Easily distributed
- Portable
- High performance

The java code must be first complied into byte-code and then it can be interpreted by a JVM, which means that the byte-code can be run on any device which has the JVM installed. This platform independence allows Java to be portable, and the two step process also allows for error reduction and better security.

Java is also an object-oriented programming language and uses classes and objects, providing features such as code reusability and maintenance. The compiler and the interpreter help reduce code and runtime errors, which makes Java reliable, and it also has a garbage collector and memory allocation mechanism.

2.3 The Eclipse IDE

Eclipse is an integrated development environment (IDE). It contains a base workspace and an extensible plug-in system to customize the environment. Written mostly in Java, it can be used to develop Java applications, and by using different plug-ins, it can also be used to develop applications in other programming languages, such as Ada, C, C++, Fortran, Haskell, JavaScript, Perl, PHP, Python or Scala, among others.

Eclipse began as one of the projects of IBM Canada. Object Technology Internatonal (OTI), which had previously marketed a family of integrated development products named VisualAge, developed the new product as a Java-based replacement. In November 2001, Eclipse began to be developed as open-source software. The number of stewards responsible for the development work increased to over 80 by the end of 2003, and in January 2004, the Eclipse Foundation was created. [13]

Developers interested in creating Android applications, need to install the Android Development Tools, a plug-in for the Eclipse IDE that is designed to give developers a powerful, integrated environment to build Android applications. ADT extends the capabilities of Eclipse to let developers quickly set up new Android projects, create application UIs, add packages based on the Android Framework API, debug their applications using the Android SDK tools, and even export signed or unsigned apk files in order to distribute their applications.

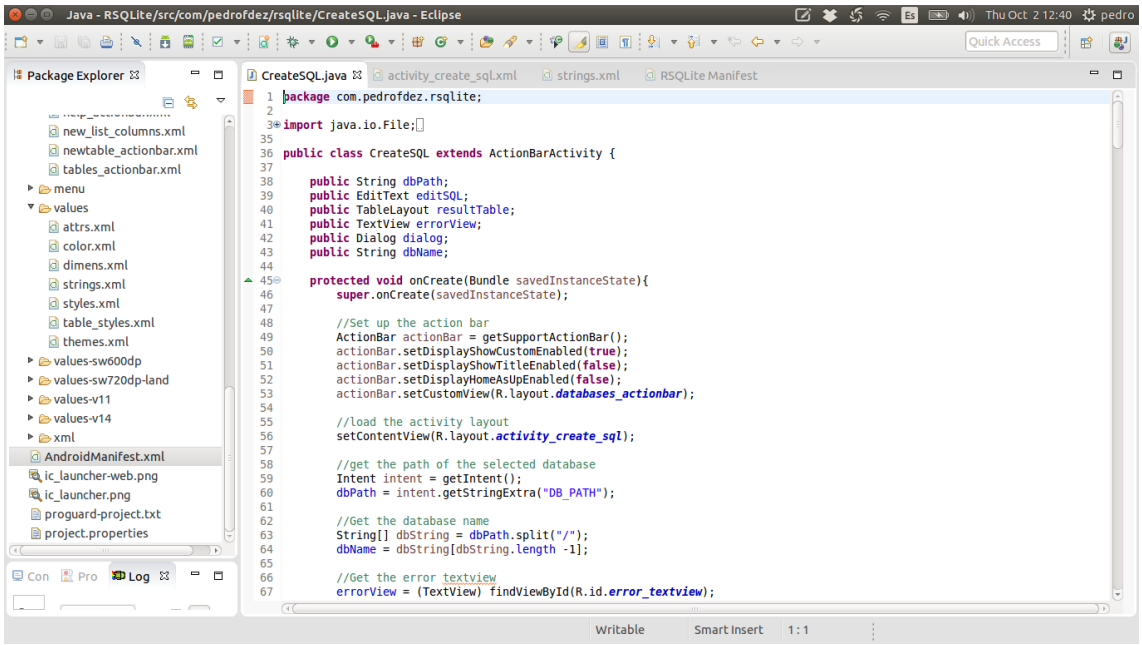

Figure 2.1: Example of the Eclipse IDE running in Ubuntu

2.4 Android

Android Inc. was founded in Palo Alto, California in October 2003 by Andy Rubin, Rich Miner, Nick Sears and Chris White. Rubin's idea was to develop "smarter mobile devices that are more aware of its owner's location and preferences". Android Inc. operated secretly at first, despite the past accomplishments of the founders. They tried to develop an advanced operating system for digital cameras. Soon they realized that the market for such products was not large enough, and they diverted their efforts to produce a smartphone operating system to rival Symbian and Windows mobile.

Google acquired Android on August 17, 2005 but key employees like Rubin, Miner and White stayed after the acquisition. At Google, the team lead by Rubin continued to develop a mobile device platform powered by the Linux kernel. Google presented the platform to handset makers and carriers promising a flexible and upgradeable system. They had lined up a series of hardware components and software partners and signaled to carriers that it was open to various degrees of cooperation on their part.

On November 5, 2007 the Open Handset Alliance was formed. They were a group of technology companies including Google, device makers such as HTC, Sony and Samsung, wireless carriers like Sprint and T-Mobile and chipset Makers such as Qualcomm and Texas Instruments. Their goal

was to develop open standards for mobile devices, and Android was unveiled that day as its first product, a mobile device operating system based on the Linux kernel version 2.6.25. The first commercially available smartphone running Android, the HTC Dream, was released on October 22, 2008. [6]

Android has now been in the market for more than 6 years, and during all this time, it has experienced a huge rate of changes unlike any other development cycle ever. Google's rapid-iteration, web-style update cycle was applied to the mobile operating system, and the result has been a fast and continuous improvement. The last few years, Android has been on a six month development cycle, but for the first years of its commercial existence, Google was updating its newest version every two and a half months. By comparison, its rivals move at snails pace. Microsoft's desktop OS is updated every three to five years, and Apple updates OS X and iOS once a year. And even those yearly updates are not of equal magnitude, for example, iOS had only one mayor design revision in seven years. On Android, however, users are lucky if something looks the same now as it did last year.

From a developer's point of view, there are numerous programming languages available such as C, C++, PHP or Java and any of them can be used, but Android uses the latter as the native programming language. Most of the applications for Android are created using Java and then compiled using the javac compiler. Those compiled files are converted into Dalvik Executable (.dex) files and then run in the Dalvik Virtual Machine, which is a virtual machine to test and run the applications. The applications developed using Java use the Android Software Development Kit (SDK) whereas the ones developed using C/C++ use Android Native Development Kit (NDK).

2.4.1 Android architecture

In order to create an app, someone could simply download the required development tools and start writing code, but a good Android development knowledge requires and understanding of the overall architecture of the operating system. Android is designed in the form of a software stack. The Android Stack (Figure 2.2), as Google calls it, has layers, which are made of several programs or libraries. Each layer of the stack, and the elements within each layer, are tightly integrated and carefully tuned to provide the optimal application development for mobile devices, as well as a reliable execution environment.

Kernel Layer

At the bottom of the stack is the Linux kernel. It provides a level of abstraction between the device's hardware and the upper layers of the Android Stack. Is based on Linux 2.6 and provides preemptive multi tasking, low-level core system services such as memory, process and power management, a network stack and device drivers for some parts of the hardware like for example the display or the audio.

Android Runtime Dalvik Virtual Machine

As mentioned before, the Linux kernel provides a multi tasking execution environment, but the Android applications do not run directly as processes on the Linux kernel, but on their own instances of the Dalvik Virtual Machine (VM)

Running applications on virtual machines has a number of advantages. Firstly, applications are essentially sandboxed and they can not interfere (either intentionally or otherwise) with other applications or with the operating system, nor can they directly access the device's hardware. Secondly, the level of abstraction means that applications are never tied to any speficic hardware, which is one of the most important features of Android. The Dalvik VM was developed by Google and is more efficient than the standard Java VM in terms of memory usage and is specifically

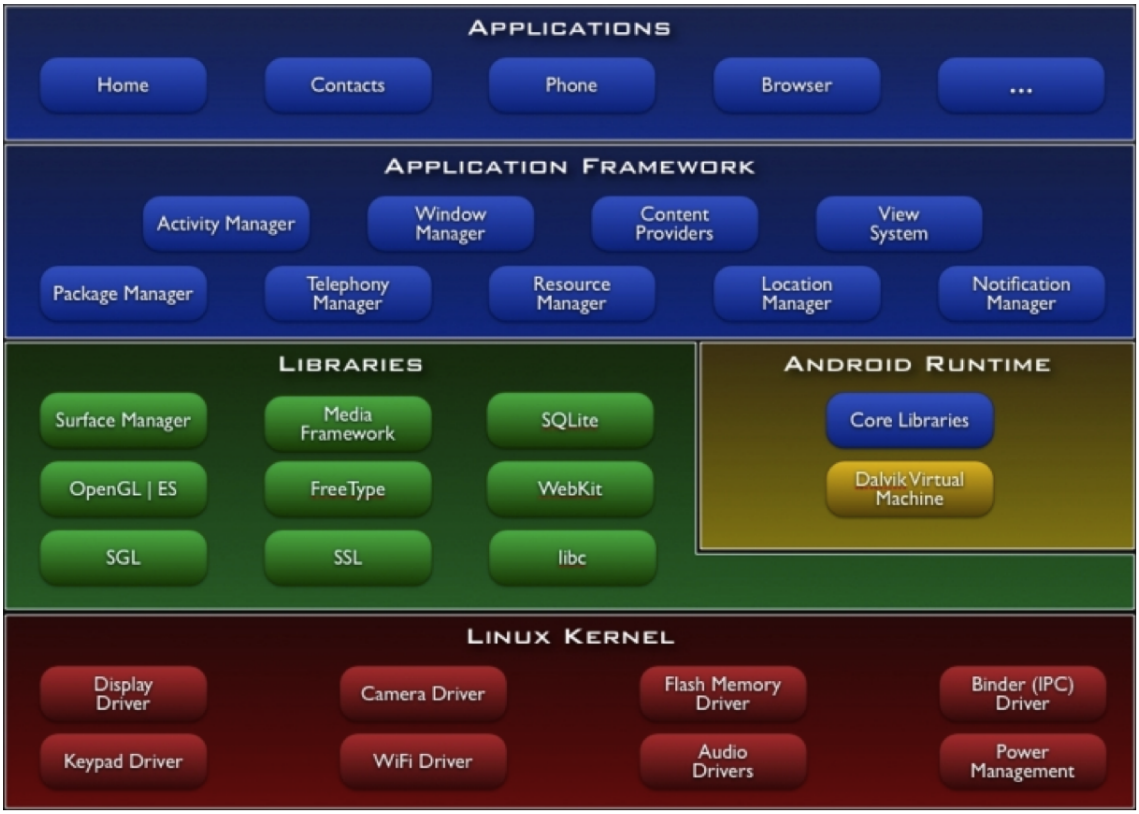

Figure 2.2: Android Stack [7]

designed to allow multiple instances to run efficiently within the resource constraints of a mobile device.

In order to execute an application in the Dalvik VM, the application's code must be transformed from the standard Java class files to the Dalvik executable format, which has a 50% smaller memory footprint than the standard Java bytecode.

Android Runtime - Libraries

There are three main categories of the Android Core Libraries

- Dalvik VM libraries: They are the libraries used for interacting directly with an instance of the Dalvik VM.
- Java Interoperability Libraries: These libraries provide support for tasks such as string handling, network and file manipulations. They are an open source implementation of the Standard Java Core Libraries, but are adapted to be used by applications running in a Dalvik VM.
- Android Libraries: These libraries are specific to Android development, including the application framework libraries in addition to those that facilitate user interface building, graphics drawing and database access.

Open source libraries:

- Surface manager: Composing windows on the screen
- SGL: 2D Graphics

- Open Gl — ES: 3D library
- Media Framework: Supports playback and recording of various audio, video and picture formats
- Free Type: Font rendering
- WebKit: Browser engine
- libc: System C libraries
- SQLite
- Open SSL

Application Framework

Is a collection of services that form the environment in which Android applications run and are managed. This framework implements the concept that Android applications are built from reusable, interchangeable and replaceable elements. They are even able to publish their capabilities along with any other data so they can be found and reused by other applications.

The Android Framework includes the following key services:

- Activity Manger: Controls all aspects of the application lifecycle and activity stack.
- Content Providers: Allows applications to publish and share data with other applications.
- Resource Manager: Provides access to non-code embedded resources such as strings, color setting and user interface layouts.
- Notifications Manger: Allows applications to display alerts and notifications to the user.
- View System: An extensible set of views used to create application user interfaces.
- Package Manager: The system by which applications are able to find out information about other applications currently installed on the device.
- Telephony Manger: Provides information to the application about the telephony services available on the device such as status and subscriber information.
- Location Manger: Provides access to the location services allowing an application to receive updates about location changes.

Applications

The applications are at the top of the Android Software Stack. These applications can be native apps provided by the particular Android implementation, like for example the phone app, or the calendar app, or they can also be third party applications installed by the user after purchasing the device.

2.4.2 Android Application Components

They are the basic elements required to develop an Android application. Through these components, the systems can interact with the application. There are many types of components which are objects defined in the Android SDK and provide different methods that developers need to call and extend these classes.

The main Android Application Components are:

- Activities
- Services
- Content Providers
- Broadcast receivers
- Intents

Activities

An activity is an individual user interface screen in an Android application, where visual elements called views are shown so the user can interact with them. The views can be added programmatically to the activity itself, or hardcoded previously in the xml layout file.

An Android application usually has more than one activity and each one operates independently, but can be linked to one another. For an activity to be created, it must be previously defined in the application's manifest file. Finally, each activity class of the application must extend the Activity class defined in the Android SDK.

Services

A service is an Android application element that runs in the background and has no visual user interface. Services are used to perform the processing sections of the application in the background. A service can be launched by different application components such as activities or other services and it will continue to run in the background even after the user switches to a different application. For this reason, services are very likely to be destroyed by the Android system to free resources.

All Android services must be implemented as a subclass of the Service class defined in the Android SDK.

Content Providers

The content providers in Android deliver a flexible way to make data available across applications. In case a developer where creating any type of data in an app and were storing it at any storage location, database, file system or online storage. Other applications would be able to query, access or even modify the created data, as long as the content provider allowed it. In a similar way, a developer can access data created by other applications, also by using content providers.

All custom content providers are implemented as a subclass of the ContentProvider class which is defined by the Android SDK.

Broadcast receivers

Broadcast receivers are used to receive messages broadcasted by the Android system or by other Android applications. There are many broadcasts initiated by the Android system itself and other applications can receive them by using a Broadcast Receiver.

While programming, they can be used to receive messages and act differently depending of what it is. Applications can also initialize broadcasts.

Intents

Intents are not exactly one of the Android Application Components, but the activating mechanism that starts other components. They are the core message system in Android and compose the required message to activate components. For example, if a developer wants to start a new activity from a current one, an intent specifying with the information of the new activity must be created. The same process must be followed to start a different application, from the current one. In summary, by firing an intent, the developer is telling the Android system to make something happen.

2.4.3 Android Application Lifecycle

The Android operating system run in a vastly wide range of devices and in one way or another, all of those devices have different limited resources, it might be memory, processing power, screen size... Therefore the Android system is allowed to manage the available resources by automatically terminating running processes or recycling Android components.

Apart from resource management, Android also recreates some activities in case any configuration changes occur. The Configuration object contains the current device configuration, if this configuration changes, the activities are restarted, as they may need different resources for this new configuration. This process should not be noticeable for the user.

To make this possible, the Android platform has a series of lifecycle events which are called when a process of a component is terminated, or the configuration changes. With these events, the developer is responsible to restore the activity instance state. The instance state of an activity is the non-persistent data that needs to be passed between activities during a configuration change to restore the user preferences.

Application

Every time any of the application objects are started, an application object is created. It is created in a new process with an unique ID under an unique user. Even if it is not specified in the AndroidManifiest xml file, the Android system creates a default object. This object provides the following lifecycle methods:

- onCreate() - called when the first component of the application starts.
- onLowMemory() - called when the Android system requests that the application needs to clean up memory.
- onTerminate() - only for testing, never called in a production state.
- onConfigurationChanged() - called whenever the configuration changes.

The application object starts before any other component and keeps running at least as long as any other component is still running.

If the Android system needs to terminate processes, it follows the following priority system:

Process status	Description	Priority
Foreground	An application in which the user is interacting with an activity, or which has a service which is bound to such an activity. Also if a service is executing one of its lifecycle methods.	1
Visible	User is not interacting with the activity, but the activity is still visible or the application has a service which is used by an inactive but visible activity.	2
Service	Application with a running service which does not qualify for 1 or 2.	3
Background	Application with only stopped activities and without a service or executing receiver. Android keeps then in a least recent used (LRU) list and if required, terminates the least used one.	4
Empty	Application without any active components.	5

Activity Lifecycle

The Android system is also allowed to recycle components in order to free up resources. This section explains how activities are tagged across the application's lifecycle. An activity can be in any of these states:

- Running: It is visible and interacts with the user.
- Paused: It is still visible but partially obscured. The instance is running but it may be killed by the system.
- Stopped: It is not visible. The instance is running but it may be killed by the system.
- Killed: It has been terminated by the system or by a finish() call in its method.

As mentioned before, the user should not notice the changes of state in an activity. To accomplish this, the developer should store and restore the activity's state at the right moment. Also, any unnecessary actions if the activity is not visible should be avoided.

- onCreate(): Called when the activity is created. Used to initialize the activity, for example, create the user interface.
- onResume(): Called if the activity get visible again and the user starts interacting with the activity again. Used to initialize fields, register listeners, bind to services...
- onPause(): called once another activity gets into the foreground. Always called before the activity is not visible anymore. Used to release resources or save application data. For example unregister listeners, intent receivers, unbind from services or remove system service listeners.
- onStop(): Called once the activity is no longer visible. Time or CPU intensive shut-down operations such as writing information to a database should be in this method.

2.4.4 Android SDK and development environment

The Android Software Development Kid (Android SDK) is a collection of software development elements used to develop applications for the Android platform. The SDK is available to download for free form the Android Developers website and it has the following features:

- Required libraries
- Support libraries
- Debugger
- Emulator
- Documentation for the Android APIs
- Sample source code
- Tutorials

With each release of a new Android version, the corresponding SDK is also made public.

For new developers, the Android ADT Bundle is available also in the Android Developers website. This bundle includes all the necessary components and programs to begin developing apps:

- Eclipse IDE + ADT Plugin
- Android SDK Tools
- Android Platform Tools
- A version of the Android platform
- A version of the Android system image for the emulator

The ADT plugin extends the capabilities of the Eclipse IDE to allow developers set up new Android Projects, build user interfaces and add packages based on the Android Framework API. Developers can also debug applications using the Android SDK tools and export .apk files in order to distribute the application.

For more experienced developers, any of the stand-alone SDK tools are also available. However, Eclipse and the ADT plugin will be eventually replaced by Android Studio, a new Android development environment based on IntelliJ IDEA and currently in a beta stage (v 0.8.0)

2.5 SQLite Databases

2.5.1 Structured Query Language (SQL)

SQL is a special-purpose programming language designed to manage data stored in a relational database management system (RDBMS). It consists of a data definition language, and a data manipulation language. The range of SQL operations include data insert, update and delete, schema creation and modification and data access control.

It was one of the first commercial languages to adapt Edgar F. Cood's relational model, described in his 1970 paper A Relational Model of Data for Large Shared Data Banks. Despite not entirely adhering to the relational model, it became the most widely used database language and eventually became standard. It was first accepted by the American National Standards Institute in 1986 and by the International Organization for Standarization (ISO) in 1987. Despite these standards, the code is not completely portable among different database systems and there are many different systems with their own extensions, such as MySQL, PostgreSQL, MSSQL or SQLite. [8]

These are some of the most important definitions regarding SQL elements:

- A **table** is a set of values that is organized using a model of vertical columns and horizontal rows. The columns are identified by their names.
- The **schema** of a database systems is its structure, described in a formal language. It defines the tables, fields, relationships, views... etc.
- A table **row** represents a single, implicitly structured data item. It may be also called a tuple or a record.
- The **columns** provide the structure by which the rows are composed.
- A **field** is a single item that exists at the intersection between one row and one column.
- A **primary key**, uniquely identifies each row in the table.
- A **foreign key** is a referential constraint between two tables. It identifies a column or a set of columns in one table that refers to a column or set of columns in another table.
- A **trigger** is a procedural code that is automatically executed in response to certain events on a particular table in a database.
- A **view** is a specific representation of data from one or more tables. It can arrange data in some specific order, highlight it, or hide it.
- A **transaction** is an automatic unit of database operations against the data in one or more databases.
- An SQL **result set** is a set of rows from a database, returned by the SELECT statement.
- And **index** is a data structure that improves the speed of data retrieval operations on a database table.

2.5.2 SQLite

SQLite is both an embedded SQL database engine, and an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. The code is in the public domain an is therefore free to use for any purpose, commercial or private. It supports most of the query language features found in the SQL92 (SQL2) standard and provides a simple and easy to use API.

Unlike most other SQL databases, SQLite doest not have a separate server process, it reads and writes directly to ordinary disk files. A complete SQL database with multiple tables, triggers and views is contained on a single disk file. The database file format is also cross-platform. All of these features make SQLite a popular choice as an Application File Format.

The SQLite library is very compact. With all its features enabled, the library file size can be less than 500 KiB, depending on the target platform and compiler optimization settings. If optional features are omitted, the size of the library can be reduced down to bellow 300 KiB. It can also be made to run in minimal stack space (4KiB) and with very little heap (100KiB) making SQLite a popular database engine choice on memory constrained devices such as cellphones, PDAs and portable music players. There is a tradeoff between memory usage and speed. Obviously, SQLite will run faster the more memory it has. Nevertheless, performance is unexpectedly good even in low-memory situations. [9]

The following are some of the most important commands for SQLite, which are very similar to SQL. Depending on their operational nature, these commands can be classified into groups.

DDL - Data Definition Language	
Command	Description
CREATE	Creates a new table, a view of a table, or other object in the database
ALTER	Modifies an existing database object, such as a table
DROP	Deletes an entire table, a view of a table, or other object in the database
DML - Data Manipulation Language	
Command	Description
INSERT	Creates a record
UPDATE	Modifies records
DELETE	Deletes records
DQL - Data Query Language	
Command	Description
SELECT	Retrieves certain records from one or more tables

2.5.3 Example of a SQLite Database

This is a simple example of a SQLite Database for the management of an office and its IT infrastructure.

worker	
id	Integer PK
name	Text
age	Integer
iddepartment	Integer FK
idworkstation	Integer FK

app	
id	Integer PK
name	Text
reqos	Text
minram	Integer

department	
id	Integer PK
name	Text
description	text

workstation	
id	Integer PK
os	Text
ram	Integer

apptoworkstation	
id	Integer PK
idapp	Integer FK
idworkstation	Integer FK

This example contains the three basic sql table relations: one to one, one to many, and many to many.

- **One to one**: This means that one of the rows in a table is related to one of the rows in another table. Each record of one table can be linked to one single record in the other thable and vice versa. In this example each worker has its own workstation. A worker can not have more than one, and a workstation can only have one worker. This relation is not very used, because the columns of the second table could be inserted in the first one, however there are some situations in which these relations can improve performance.
- **One to many / many to one**: With this relation, a row from one table can be related to many rows in a different table. This effectively saves storage, as the related record does not have to be stored multiple times in the relating table. In the example, each worker belong to one department, but many workers can belong to a department. Without this relationship, the info from each worker's department should be saved in each worker's table, creating a lot of duplicated data. And in case of any change in the department data, with this relationship the change must be done only once.
- **Many to many relationships**: This means that one, or more rows in a table, can be

related to as many rows from another table as the user wants. This requires a mapping table in order to keep a list of the single relations. In the office example, the "apptoworkstation" is the table which keeps a record of the apps installed on each workstation. An app can be installed on more than one workstation, and one workstation can have more than one app installed.

The following are the sqlite commands to create each table from the office database:

```
CREATE TABLE department(
       _id integer primary key,
       name text,
       description text);

CREATE TABLE workstation(
       _id integer primary key,
       os text,
       ram integer) ;

CREATE TABLE worker(
       _id integer primary key,
       name text,
       id_workstation integer,
       id_department integer,
       foreign key(id_workstation) references workstation(_id),
       foreign key(id_department) references department(_id)) ;

CREATE TABLE app(
       _id primary key,
       name text,
       req_os text,
       min_ram integer) ;

CREATE TABLE app_to_workstation(
       _id integer primary key,
       id_app integer,
       id_workstation integer,
       foreign key(id_app) references app(_id) on delete cascade,
       foreign key(id_workstation) references workstation(_id) on delete cascade);
```

2.5.4 Other SQL implementations

There are many other SQL implementations such as MySQL, PostgreSQL or MSSQL. The biggest difference between them and SQLite is that they need both server, which stores the databases, and a client, which sends the SQL commands and retrieve the data. These other implementations have better options for performance tunning and unlike SQLite, can manage users and permissions, but they are far more complex to set up.

Therefore, the most complex SQL implementations are designed for large scale production environments.

2.5.5 Current SQLite Database browsers for Android

There are several apps related to SQLite Databases available to download in the Google Play store. There are some that focus more on the root SQLite files that some of the apps in the device can create, others give the user the ability to connect to a remote SQLite database, and other have more of a browser/editor focus. In the latter group there are between 6 to 8 apps, and the best

two have been installed and tested, in order to get a picture of the current possibilities the users have, and the potential room for improvements.

aSQLiteManager [10]

Name	aSQLiteManager
Developer	Andersens Android Applications
Number of reviews	1.238
Score (0 to 5)	4,5
Installs	100.000 - 500.000
Current version	4.4.1
Updated	January 19, 2014

This is the best rated free SQLite manger in the Android Play Store right now. It has all the required functionalities desired from a database manager. It can open a database from the file system, create a new one, or select one recently used, although they are not directly shown. However, it can not open SQLite files from third party applications like Dropbox or Google Drive.

The user is directly sent to the resident file browser. Once the database is selected, the user sees a list of tables, there is also the option to browse views and indexes, as well as creating a custom SQL query. The user does have some help creating the query, but is not very intuitive, having to access it by the options button.

Once a table is selected, the user is not sent directly to the table content, but to the table info, labeled as field. One of the biggest problems with the application, is that to browse the data of the application, the user has to use two buttons (PgUp and PgDn) which does not make the user experience very smooth.

A good feature in the preferences page, is that the user can change the color set up, font size, and page size.

SQLite Manager [11]

Name	SQLite Manager
Developer	John Li
Number of reviews	1.374
Score (0 to 5)	3,8
Installs	100.000 - 500.000
Current version	1.3.3
Updated	June 5, 2014

Unlike aSQLiteManager, this application can also load a file from Dropbox, although this is the only option for a third party application and it needs a separate button. In the first screen, the user can also open the resident file browser, and can directly see the latest used databases.

Once a database is selected, the user is shown the list of the tables and two buttons, one to run a custom SQL command, and another to see the table information. There is a redundant button in the options menu to open the custom SQL command activity.

When a table is selected, the user is directly shown its content. Like the last application, it only loads the first 50 rows, then the user has to navigate through pages using two buttons.

However, these two applications only allow the user to browse or edit single tables, without any concern to the relations between tables. There is not an application available in the Play Store which can do that and that is exactly the main goal of this project: to create an intuitive application for a user to easily navigate between the tables of a database, using the relations which

link those tables together.

Building an application able to understand these relations is more complex than working only with single tables, but the final improvement of the user experience should be even greater.

Chapter 3

Application Structure and Design

3.1 Requirements

As it was briefly commented on section 1, the main objective of this application is to fill a void which all of the top SQLite Managers on the Play Store have: the relations between tables.

But first of all, the application must have the basic functionalities of any SQLite Manager. The user should be able to browse and edit content in a database with a user interface as simple as possible. Then to distinguish itself from the competitor apps, the application should also give the user the ability to navigate through the different tables of the database by using the relations between tables.

These are the primary objectives of the application:

- Open a SQLite database file
- List all the database tables
- Show the content of a table
- Show the content of a table row and the extended related content
- Create, edit and delete table entries

There are also some secondary objectives:

- Save a list of the recently used databases
- Open a database file directly from Dropbox
- Order the content of a table
- Run custom SQL commands on the database.

3.2 Activities Flowchart

The above figure 3.1 shows the various ways the user can move throughout the application, starting at the top left from the *Select Database* activity (Figure 4.1).

At this first activity, the user should be able to select a database file in three different ways: open it from the device's internal memory, choose a recently used file, or open the example database

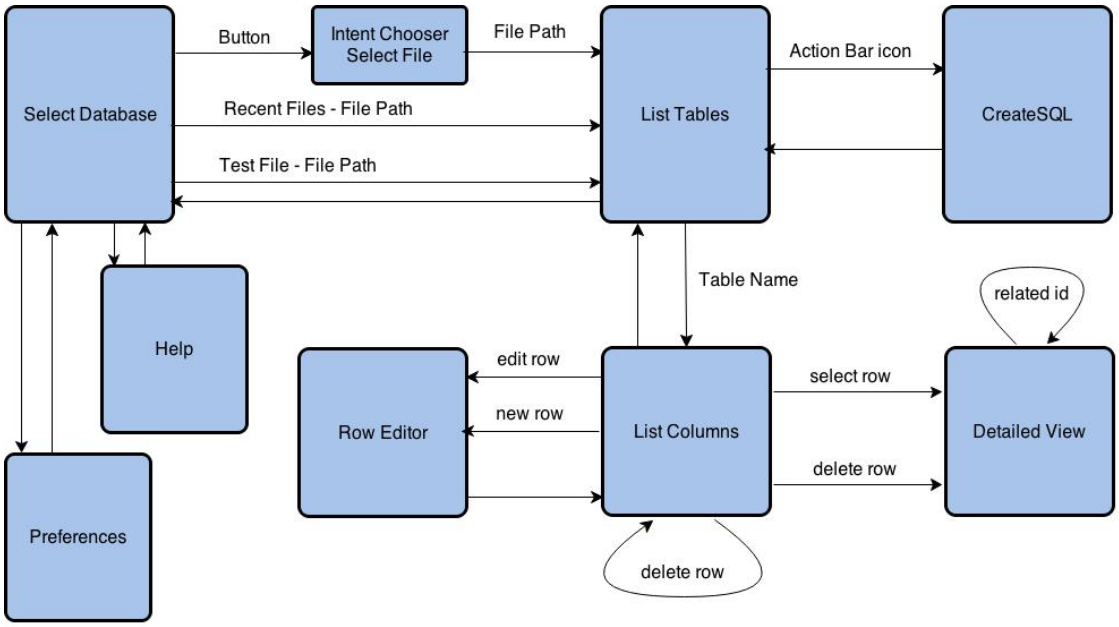

Figure 3.1: Application's activities flowchart

file. As can be seen in figure 3.1, the activities *Prefenreces* and *Help* activities (Figures 4.7 and 4.8) can also be accessed from *Select Database*.

Once a SQLite file is selected, the application will start the *List Tables* activity (Figure 4.2) and show a list with all the tables in the database, all of them clickable for the user to see their content. At this point, the user should also be able to see the selected database information, and to enter the *Create SQL* activity (Figure 4.6, where any SQL command can be written and run in the database.

If the user clicks on one of the table names, the activity *List Columns* starts (Figure 4.3). This activity shows the content of a table, just as any other database manager application would do. The user will be able to delete a row, edit it, or create a new one. To edit or create a row, the same *Row Editor* (Figure 4.5) activity will be used with the only difference that in case of editing a row, the current data will be shown for the user to change it.

The first real difference with all the other SQLite Managers comes in the *List Columns* activity (Figure 4.4). If the user has decided so, the application will check for a column_visibility table which will indicate if the data of a column should be visible on this activity. The application will run this check if the user has not unchecked this option on the *Preferences* activity.

The second and most important new feature also comes in the *Detailed View* activity. By default the user will be able to click on any of the rows of a table, to open the *Detailed View* activity, which as its name suggests, opens a new table with all the information of a row. The first part of this information includes all the data of the row and in case some of the columns are foreign keys, the extended information of the related table. The second part includes all the rows of different tables, in which the selected entry is a foreign key

From this activity, the user will be able to open other *Detailed View* activities by clicking in the content of the table. The delete row or edit row features will also be accessible.

3.3 Database Structure Requirements

In order to experience all the capabilities of this application, the structure of the SQLite database must follow a few simple naming conventions. The application uses these guidelines to understand

the relations between tables, necessary for some of the capabilities. However if these conventions were not met, the user will also be able to execute simple tasks such as browse table entries, create new ones or delete them.

The naming conventions are:

- Primary Key Fields: The first field of each table should be a Primary Key, and it should be named *_id*.
- Foreign Key Fields: Foreign key fields should be named *id_* + the exact same name of the parent table. For example if a table *Order* has a foreign key linked to the table *Customer*, the column should be named *id_Customer*.
- Table *column_visibility*: If a table has too many columns, it can be difficult to move through them to find something. With the *column_visibility* table the user has the option to hide the less necessary columns in the content table, although they will still visible on the detailed view. The *column_visibility* table consists of three columns: *table_name*, *column_name* and *visibility*. The columns will not be visible if visibility is 0.

Chapter 4

Application Development

This chapter explains the development process for each activity in the application. The activities were listed and roughly explained in section 3.2, but this chapter will now go more in depth to describe the development process. First of all section 4.1 describes the different methods that can be used to select and open a database file. Section 4.2 does not describe an activity, but the two important java classes that will create the different table views used in most activities. The rest of the sections illustrate the development process for each of the remaining activities.

4.1 Selecting and opening the SQLite file

The main goal of the first activity of the application, called *Select Database* is to select a SQLite file. For this activity the user interface components will always be the same, so all of them will be written in a xml file which will be loaded at the beginning of the activity code. It will be composed by a relative layout with four children: a button, two text views and a scroll view which will contain a table layout. The button, one clickable text view and the table rows will enable three different ways for the user to select a SQLite file.

To give the three options for the user to select a database, when the activity is created, it will automatically set up both buttons, and load the table with the latest used databases. Each option has an OnClick method, but all of them finish with the creation of an intent which will have a database file path as an extra.

The "Open SQLite File" button

This will be the main way to select the database files. When the button (Figure 4.1, section a) is pressed a menu will appear with different methods for the user to select a file. In case the user does not have a file browser installed, there will be an option to open the application's own file browser, by clicking "Select a file". This browser is an external library downloaded and added to this project.

The "Recent databases" list

In order to make a faster and easier user experience, when the user selects a database, its path is saved in an internal file. This file will contain the paths of the last five used files. Every time the user selects a new file, the code checks if the path is already in the cache file. If not, it will be added. There will be a table listing the last 5 used files (Figure 4.1, section c) and when the user selects one of them, the path is retrieved from the cache file and sent to the next activity.

The "Example Database" button

In case the user does not have any SQLite file available or wants only to test the application, there is an example database file available. When this option is selected (Figure 4.1, section d) , a SQLite file is retrieved from the assets folder of the application and copied into the device's memory. Once the file has been copied, the path is sent to the next activity.

Using the options button in the action bar (Figure 4.1, section b) the user can also access the preferences and help activities, which are explained on section 4.8.

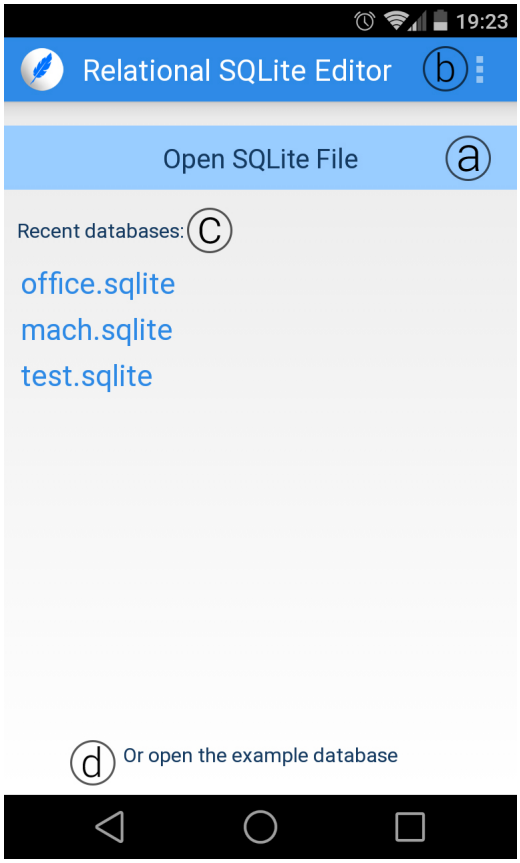

Figure 4.1: Select Database activity

4.2 Building the database tables - Table Helper and Row Helper

Most of the application's work consists on making SQL queries to the database and building tables with the results. To avoid confusions between the database tables and the tables that appear on the applications user interface, the latter will be from now on known as table views. These views will be seen in most of the activities. There is one table view to show the database's tables, one to show the content of a table, one to show all the content in a row, one to show other elements with the same id, one to list table entries in order to select one, and finally, one to create or edit an entry. It was perfectly possible to write the code to make every one of these table views, but instead, the decision was made to find some similarities between them and to create a tool that would allow the developer to save code, and time.

In this case, there are three table views that constitute most of the application and there will be a method to create them with a simple command in the activity code. The tables are:

- The *Content table* view: It will show the rows that a SQL query returns.
- The *Detailed table* view: It will show all the fields of a table row, including related elements linked to different tables by foreign keys.
- The *Edit table* view: It will show a table for the user to edit a table entry or to create a new one.

The Android element used to build table views is the TableLayout, it is a class in the Android API and provides with two ways to build a table view: programmatically, by creating it from scratch in an Activity, or by inflating a xml layout file which has the table view elements predefined in it. The second option is very convenient when the table structure never changes but in this case it is impossible to know how many rows a table view will need before starting the application, so they will have to be built programmatically. A new helper class named TableHelper will be written, extending the TableLayout capabilities. It will consist on one method for each kind of table view, and each method will return a view with all its content and format.

The original idea was for TableHelper to provide all three types of table views, but after the first series of tests a problem appeared with the content table view. For big table views with more than 100 rows, it took too long for the application to build them and there was a lag of about 3 seconds from the moment the user selected a table, to the moment when the table content appeared on the screen. This kind of lag was unacceptable, therefore a new approach was necessary.

The most popular SQLite applications in the Play Store do not load entire tables either, the content is loaded by pages instead. Each page shows 20 to 40 rows, depending on the application. This approach is definitely faster than loading the entire table but it is not intuitive enough. After some though the decision was made, not to alter the user interface. The user should still see a full screen of results, but instead of loading all the entries at once, the rows will be added in packages of 40 at a time. And only if the user scrolls down to the end of a package, the next one will be added. This solution will allow the app's user interface to remain very intuitive and easy to use but will also help improve the performance. After the improvement, the loading time for tables with either 50 or 5000 rows where the same, with the user experience being also the same. By fine tunning the precise position for the event at the end of a current table view, the rows are added without almost no noticeable lag. However, this change meant that the table can not be created by the TableHelper because the content is loaded by rows, instead of a full table. The content needed to be loaded by a RowHelper method which will return an array of 40 Table Row elements each time is called.

The TableHelper methods used to create the detailed table and the edit table are:

- `createDetailedTable(String dbPath, String tableName, String id)`

  This method creates a table view with the information of one row. It only needs three arguments: `dbPath` is the path of the selected database, `tableName` is the name of the selected table, and `id` is the id of the selected row.

  Firstly, the method opens the database connection and runs a query to get the names of the columns. Then it iterates and runs a query for each column. If the column does not start with id_" it means that it is not a foreign key and it just writes the data in the table view. If the column is a foreign key, it starts a different iteration with a query to the related table in order to show all the related data instead of the foreign key.

- `createEditorView(String dbPath, String tableName, String id)`

  This method returns a table with two elements for each column of the selected table. The first one is a Text View with the name of the column and its type. The second one is an Edit Text element to insert or edit the value of a field. Just as `createdDetailedTable`, this method iterates a cursor with the column's names and also has a different behavior when the column is a foreign key. In this case, in the second row, a button is added besides the Edit Text element, so when the table is used, the user can either insert the key manually or use the button to open and browse the related table.

In this method, the argument `id` divides two different uses: if is 0, the user is creating a new table entry and all edit views will be empty, and if it is different than 0, the user is trying to edit a row, in which case the id of the row will be given and the edit views will have all the current data in them.

Both tables are returned by each method and are completely formated with the application's standard format, sharing sizes, fonts and colors with the rest of the activities.

When the *complete table* is being built, the method will first search for the "column_visibility" table, if the user has selected such option in the Preferences. Before adding table columns to the activity layout, it will check that the column name does not have a 0 visibility, which will mean that it should not be shown in the *complete table*.

To create the entire table view, the RowHelper has two methods, one return a first row with the column names to be shown and the second one return 40 rows of the table's content.

- `getTitleRow(String dbPath, String tableName`

  Firstly the method will check if the column_visibility table even exists. If it does and the user has not uncheked the option that allows the application to hide columns in this view, the method will check every column name in the table and the ones that are not supposed to be hidden, will be added to the title row and given format.

- `getRows(TableRow titleRow, String dbPath, String tableName, String whereExtra, String order, int offset)`

  This is probably the most flexible method in the entire application. First with the title row, it creates a string containing all the column names to add to the query. The query will always have a limit of 40 results and the offset will be a variable, since the method is called each time the user reached the bottom of a table in some activities.

  The strings `whereExtra` and `order` are optional. The first one will only be used to filter the results if necessary and the second one will be used to order the query results when the user clicks the name of a column in a *complete table*.

Since each activity requires different listeners for each table, those will be added in the activities' code, once the helpers return the tables or rows.

4.3 List Tables activity

The List Tables activity is the view where the user has a list of the tables in the selected database. It also provides access to the *CreateSQL* activity that allows the user to write and run custom SQL queries and it also creates the database relations file which includes all the foreign key relations.

But the most important feature of the activity is the main table (Figure 4.2, section c). With the database path from the *Select Database* activity, the application runs the following command: "`SELECT name FROM sqlite_master WHERE type='table';`". This returns a one column cursor with the names of all the tables in the database. Then the code simply goes through the cursor, adding one table row for each table.

Each table row will have an `OnClickListener` that will get the name of the selected table, put it in a new intent along with the database path and it will start the next activity, *ListColumns*.

In the action bar there will be an SQL icon (Figure 4.2, section a) that will simply start the *CreateSQL* activity along with an option to see the database info (Figure 4.2, section b), shown in a small dialog.

Finally, the `createRelationsFile` method writes a txt file with all the 1 to n relations in the database. First of all, the method gets a cursor with all the table names, and then checks the

SQL descriptions of the tables, looking for foreign keys. If a table *Order* has in its description a column named *id_Customer*, the method will detect it and save the relation. This file will later help to show the user database relations. For example, when the user looks up the information on a *Customer* record, the application will know that it is a foreign key in a different table, and it will automatically display all the *Order* entries with the same *id_Customer* that the user selected.

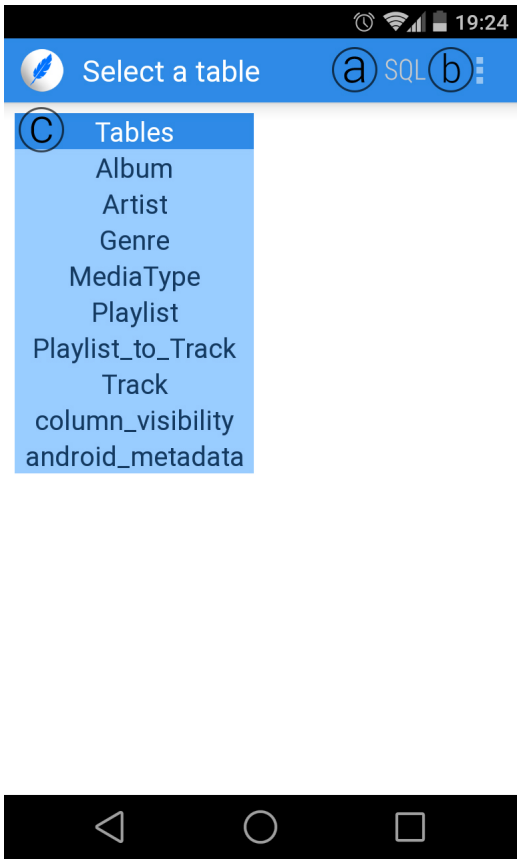

Figure 4.2: List Tables activity

4.4 List Columns activity

On this activity, the user will see the content of the table selected in the last activity. It may look simple but it is key to direct the user to some of the most important activities in the application.

The most important element the user will see is the content table view (Figure 4.3, section b) which was explained in 4.2. As explained before, to improve performance the table is build in packages of 40 rows returned by the RowHelper when the user scrolls to the end of the current table. However, the stock ScrollView class from the Android API does not have a valid listener to perform the required task so it was extended in ScrollViewExt and the setScrollViewListener was created. This method checks how scroll has changed and therefore is possible to know if the current position has reached the bottom of the view, or if it is near.

To be able to scroll both vertically and horizontally, the table layout is created inside an ScrollViewExt element which is inside an HorizontalScrollView. Once the intent extras such as database file path, table name and extra queries are saved the views are created and the RowHelper is used to get the title row and the first 40 rows of the table. As explained in 4.2, the title row method will check the column_visibility table if the user has such option selected. This means that some of the columns may not appear.

Once the table view is built, several OnClickListeners are added:

- `showRelationListener`: It is set in every content row of the table view only if the user has selected such option in the preferences activity. It will open a Detailed View activity for the selected table entry.
- `sortListener`: It is set in each text view of the column names row. It will restart the activity, re-arranging the table view to order it by the selected column. The first click will order it ascendantly, a second one will set the table in descending order and a third one will restore the table to its original order.
- `ContextMenu`: A long click on any content row will trigger a context menu with three possibilities:
  - Edit row: It will open the Row Editor activity with the current row data for the user to edit it.
  - Delete row: After user confirmation, it will delete the selected row.
  - Show detailed view: It will open a Detailed View activity for the selected table row.

The user can also add a new entry to the table by using either the icon on the action bar (Figure 4.3, section a) or the *add new row* button at the end of the table.

Figure 4.3: List Columns activity

4.5 Detailed View activity

This is probably the activity that separates this application from the other SQLite applications available in the Android Play Store. On this screen, the user will see a detailed view of a selected table entry (Figure 4.4, section c & d). In case this entry is also a foreign key in a different table, a second table will be shown with the entries of the parent table with the same foreign key (Figure

4.4, section e). For example, if the user selects a *Customer* with *id*=3 and there is a table *Order* which has a foreign key named *id_Customer*, the detailed view will also show all the *Order* entries with *id_Customer*=3.

The detailed view table is built by the TableHelper as explained on section 4.2. It shows two columns, one for the column names and the other for the content. If one of the column names is a foreign key, it will also show the content of the related table (Figure 4.4, section d). These related entries will have an OnCLickListener to start a new Detailed View activity with the foreign key as the selected table entry. Using the same example as before, if the user is in the Detailed View of an *Order* with *id_Customer*=3, the table will show all the information from the *Customer* with id=3 instead of only showing the number. Also if the user clicks on any of that information a new detailed view of that *Customer* will be created.

When the activity is first started, it saves a few variables from the intent that will later be used to create the tables:

- `dbPath`: A string with the database file path.
- `tableName`: The name of the table from which the detailed view content comes from.
- `id`: It is the id of the row selected in the last activity and which information will show in the detailed table.
- `extraWhere`: This string gets added to the second table's query, in order to filter it.
- `extraOrder`: This string determines the order of the second table.

The first element to be created is the detailed view table created by the TableHelper. Then it comes the second table, which is a complete table showing a number of rows from a different table. As explained before, here are two situations that will require the second table: If the selected entry is a foreign key in a different table or if the detailed view has been created by the user clicking on the related data of a different detailed view table. Either way, the Row Helper will need both an extraTable and an extraWhere to work. Both strings might have been read from the activity intent if the current Detailed View comes from a different one, but in case it comes from a List Columns activity, the possible relations will be searched. The method `getRelations(String dbPath, String tableName` will check the relations file made in the List Tables activity and explained in section 4.3 and will return the name of the table where the selected row id might be a foreign key. If after the two checkups the strings remain empty, the second table will not appear. But if they are not empty, the complete table view will be created just as in the List Columns activity. The table view will also be created by packages of 40 rows and they will be sortable by clicking on the column names and will also have an *Add new row* button at the end (Figure 4.4, section f). Nevertheless, in this case, the possible actions will be limited to a simple onClickListener that will start a new Detailed View activity, if the user has selected so in the preferences activity.

Finally, on the Action Bar at the top of the screen, the user will be presented with two more options: edit row (Figure 4.4, section a) and delete row (Figure 4.4, section b). By clicking on edit row a Row Editor activity with the content of the current row will be started and by clicking on deleting the row the user will be asked for confirmation to delete the currently selected table entry.

Figure 4.4: Detailed View activity

4.6 Row Editor Activity

This activity has two different but similar functions: it is used to either create a new row or to edit a existing one. Either way, the activity's user interface remains the same. It consists on a list of text views and edit views. As explained at the section 4.2, each table row has a text view with the name of the column and its type and an edit view to write or edit the content of the column.

The activity receives three variables on its intent: `dbPath` is the path of the database file, `tableName` is the name is the table which will be edited and `id` represents the id of the row to be edited. If the `id` is 0, it means that the activity is being used to produce a new table entry.

The first version of this activity was simply composed of text views and edit views, but after testing it, it was clear that this simple layout was not very useful in some cases. The first problem came with the columns which represented foreign keys. The user was asked to enter only an id and in some cases that meant going back to a different table to look for the correct one. Since one of the goals of the application was to create an intuitive interface, the decision was made to add a button next to every foreign key field that would detect the related table and open its content in a dialog so the user can easily pick one, without having to memorize its id (Figure 4.5, section b). In order to recycle as much code as possible this content table view is also produced by the Row Helper and is built in packages of 40 rows, as explained in section 4.2. This method is also much slower than simply typing the id in, so both options were kept and selecting one row in the dialog only writes the correct id in the edit view. This functionality will only work if the naming conventions are met. For example, if a table *Order* has a column named *id_Customer*, under the id_Customer text view, the user will see both an edit view and a button with the text "Select Customer". If the button is clicked, a dialog will show the Customer table and when a row is selected, the edit view will be automatically updated with the selected id.

Another issue that appeared in the first versions was that the user had to manually write the

row id (Figure 4.5, section a). Having the application to automatically set the id was possible, but this might take too much control off the user, so an intermediate approach was taken. The user would have the chance to decide if the application should pre write the row id or not. In case the user checks this option in the preferences menu, the app will automatically check the highest id of the table and pre write the next one in the id's edit view.

Finally, the user can save the changes using the button at the end of the scroll view (Figure 4.5, section c).

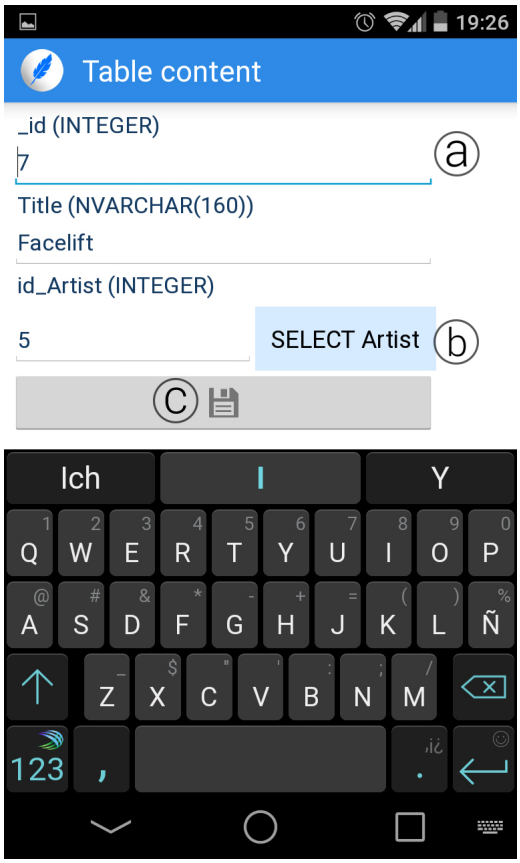

Figure 4.5: Row Editor activity

4.7 Create SQL activity

So far the application has opened an SQLite database file, show its tables, show the content of a table, a detailed view of the content of a single row, and created or edited rows. After creating all of those activities and testing them, it was clear that even though the application is very smart and intuitive, there was a lack of freedom for the user to manually run SQL commands without restrictions. Therefore the *CreateSQL* activity was added. The user can access by clicking an icon at the List Table's action bar. This location was chosen because at this point, a database file has been selected, but nothing more.

The user interface in this activity consists of three elements:

- An edit view to write the SQL command (Figure 4.6, section b).
- Two rows of buttons. The first one with the most common SQL syntax elements (Figure 4.6, section c), and the second one with the names of the tables (Figure 4.6, section d).
- A table layout to show the results if the command returns something (Figure 4.6, section e).

Although in this activity, the user's knowledge about SQL syntax needs to be significantly higher than in the other activities, it is still relatively easy to make a mistake when typing on a mobile device and that is the reason why the two rows of buttons were added. The first one is hard coded into the user interface xml layout and contains the most common SQL syntax elements such as `SELECT`, `FROM`, `WHERE` or `UPDATE`. The second one consists of one button for each one of the table names. There is a query to `sqlite_master` at the beginning of the activity to retrieve them. Both rows of buttons are placed into an Horizontal Scroll View, so they do not take up much space but are still easy to use.

In case the user runs a `SELECT` command, if the result cursor is empty it will be announced in a pop up message, instead of leaving the results space empty. If the command actually returns something, it will be shown in a table below the buttons. The table would be scrollable both vertical and horizontally. For commands that do not return anything, such as `DELETE` or `CREATE`, a confirmation pop up message will appear. If the user makes a syntax mistake, the database connection will return an error message and it will be automatically shown in the space below the button rows.

After testing, it became apparent that if a user needs to run a complex SQL command frequently, it would become a problem, to write it every single time. Something similar happened with the Select Database activity and the solution was to create a cache file with the recently opened databases so the same approach was taken in this activity.

For this capability, three methods were written:

- `getSQLhistory()`: It checks the cache file of the database on which the commands are being executed and returns an array with the strings.
- `saveSQL(String command)`: First, it checks how many commands are saved in the file. If the number is lower than 10, it simply adds a new one. If it is bigger, the oldest one is removed and the new one is saved.
- `isNotSaved(String command)`: Checks the current file and returns true if the command has not been already saved. Return false if it has.

Only after a command is successfully executed by the database engine, the activity will check if it has been already saved. If it has not, it will save it. This way, both repetitions and wrong commands are not saved in the cache file.

An option was added to the action bar to see the latest commands used (Figure 4.6, section a). This inflates a dialog with the commands and by clicking on any of them, they automatically replace whatever is in the edit view.

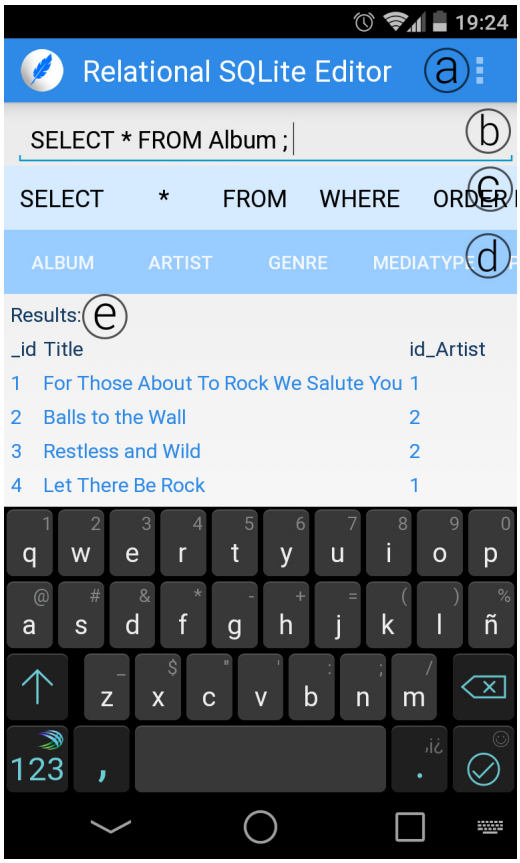

Figure 4.6: Select SQL activity

4.8 Preferences and Help activities

One of the most important goals for this project was to make the application as easy to use as possible and for it to understand the relations between tables, the database needs to follow some simple naming conventions. These conventions have already been explained in this documentation at section 3.3, but they should also be displayed at the application itself. However, if a database doesn't comply with these conventions and the user wants to just browse its content and disable some of the advanced features of the application, there should be an easy way to do so as well.

For these two features, two simple activities were created: *Help* activity (Figure 4.7) and *MyPref* activity (Figure 4.8). Both can be accessed via the action bar at the *Select Database* activity.

Help activity is simply a text view inside a scroll view, in case the display is too small. The entire text is saved in a html coded string, which is then loaded into the activity.

For the preferences activity, Android has a special activity class named Preference Activity. This class allows the developer to save the application's preferences in a xml file. This file is read by almost all other activities by a SharedPreferences element, which easily loads all the preference values. The Preference Activity automatically creates a layout for the user to edit the preferences. In this case all three options are boolean values, represented as check boxes:

- `prefColVis`(boolean, true by default): If this is disabled, the application will not check for the existence of a column_visibility table to limit the number of columns displayed in the List Columns activity.
- `prefAutoId`(boolean, true by default): If this is disabled, the application will not automatically suggest an id for a new row.
- `prefDetView`(boolean, true by default): If this is disabled, the user will not be able to access

a detailed view of a row by simply clicking on it. However, it will still be accessible through the context menu which appears when a row is long pressed.

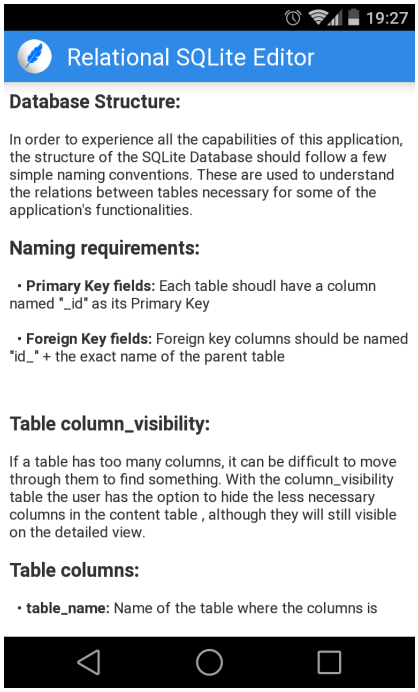

Figure 4.7: Help activity

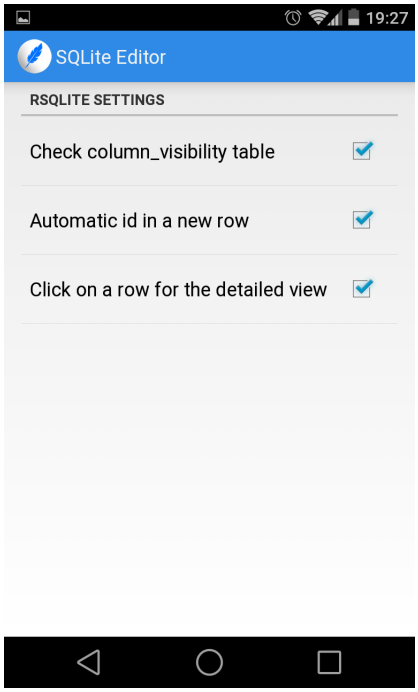

Figure 4.8: Preferences activity

Chapter 5

Testing and results

Parallel to the development process, the different capabilities of the application were continuously tested. However, several points were set so the functionalities had to be thoroughly tested. Successfully passing these, allowed further developement with a robust and reliable basis to continue the work. This did not mean that once a point was passed it was not tested again, at each point, all the previous capabilities were equally and thoroughly tested.

These points were:

- Successfully getting the path of a selected file.
- Opening a database connection and displaying its tables.
- Showing the content of a table.
- Showing the detailed and related content of a row.
- Successfully creating, editing or deleting a row.
- Final testing.

The device used for the development process has been a 2013 LG Nexus 4 running Android 4.4 and Android L. It has a 4.7 inches display, 2 GB of RAM and a Quad-core Qualcomm APQ8064 Snapdragon chipset running at 1.5 GHz. Although these specifications are no longer top market specifications, they are still above average for an Android device. The application was also tested on an older Samsung Galaxy Mini, running Android 2.3.3 and with a lower than average specifications. Although the slower device could suffice the need to test the app in a slow processor situation, to thoroughly test the user interface more devices with different screen sizes where needed.

In order to test applications in as much devices as possible, the Android SDK includes the Android Virtual Device Manager. This allows developers without many devices to test the application to create virtual devices to simulate behavior of the application on different environments. For this project, the application was tested on a virtual device running Android 2.3.3, with a 3,4 (240x432) screen and 256 MB of RAM (Figure 5.1). And to test the user interface, the screen was altered with different sizes from 3 to 6 inches.

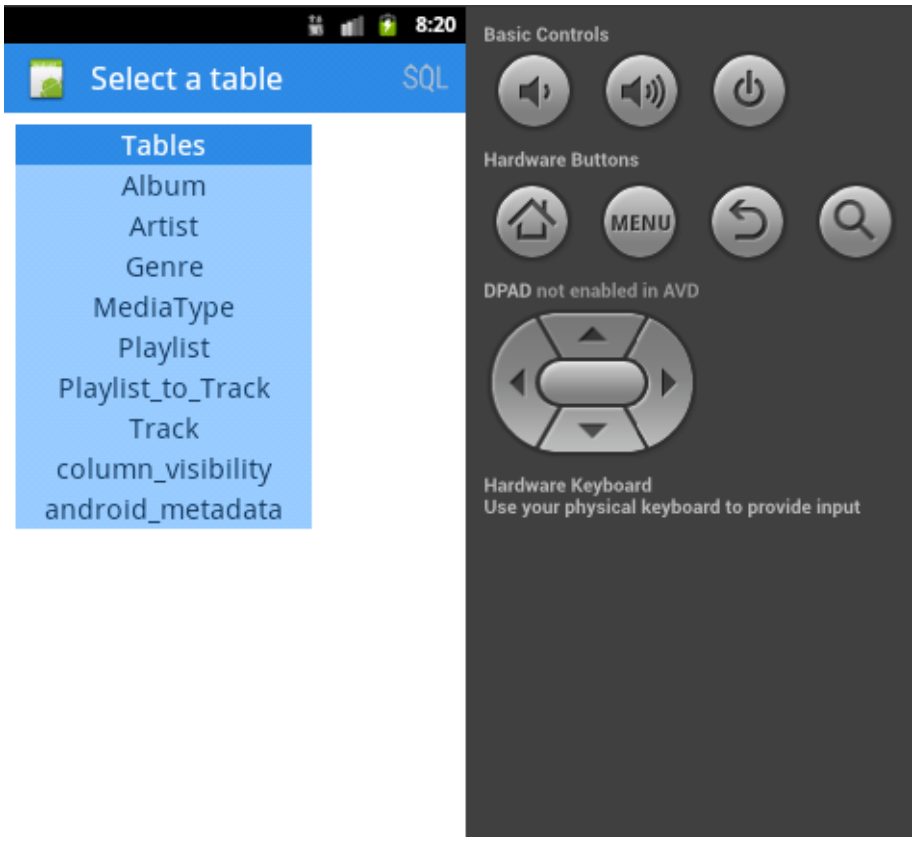

Figure 5.1: The app running in a low end virtual device

Chapter 6

Conclusions and Future work

After the completion of the development and testing processes, it can be stated that the primary goals of the project have been accomplished. The application has all the basic capabilities expected from any SQLite manager: it can open a database and list all the tables, open a table to show its content and add more rows or edit the existing ones. The app also provides an intuitive navigation process which uses the relations between the tables, and a detailed view screen that shows the user the complete information of a single database entry. The ability to hide columns using the column_visibility table is also a unique feature and although the CreateSQL activity requires to have more SQL experience, it also has a intuitive user interface.

The result is an innovative, user friendly and fast application that fills the void left from the old SQLite managers for Android. During the development process, some secondary objectives such as the ability to open files directly from Dropbox or running custom SQL commands have also been achieved.

However there is always room for improvement and this is only the first version of the app. There are some functionalities that were not critical for this project and could be added in the future to improve the application. These are three areas where further improvements could be made:

- Creating databases: The option could be added for the user to create tables or entire databases by a user interface editor, command or importing a CSV.
- Root databases: Most Android applications use SQLite Databases, but a root access is needed in order to manage them.
- More flexible naming conventions: Other possibilities could be added so the user could choose from a number of naming conventions when designing a database.

Bibliography

[1] Developer Economics: App market forecasts 2013 - 2016
http://www.visionmobile.com/blog/2013/07/_developer-_economics-_app-_market--_forecasts-_2013-_2016/

[2] *The American Heritage®Dictionary of the English Language*. Houghton Mifflin Company, Fourth Edition, 2000.

[3] IDC: Smartphone OS Market Share 2014, 2013, 2012 and 2011
http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[4] iPhone (1st generation)
http://en.wikipedia.org/wiki/IPhone_(1st_generation)

[5] "App" voted 2010 word of the year by the American Dialect Society
http://www.americandialect.org/app-voted-2010-word-of-the-year-by-the--american-dialect-society-updated

[6] Wikipedia, Android (operating system)
http://en.wikipedia.org/wiki/Android_(operating_system)

[7] Android Architecture - The key concepts of Android OS
http://www.android-app-market.com/android-architecture.html

[8] SQL
http://en.wikipedia.org/wiki/SQL

[9] About SQLite
http://www.sqlite.org/about.html

[10] aSQLiteManager - Android Apps on Google Play
https://play.google.com/store/apps/details?id=dk.andsen.asqlitemanager

[11] SQLite Manager - Android Apps on Google Play
https://play.google.com/store/apps/details?id=com.xuecs.sqlitemanager

[12] What Is The Java Computer Programming Language?
http://java.about.com/od/gettingstarted/a/whatisjava.htm

[13] Eclipse IDE - Tutorial
http://www.vogella.com/tutorials/Eclipse/article.htm