Zigurd MEDNIEKS
Series Editor

ANDROID™
DEEP
DIVE

Adam STROUD

ANDROID™ DATABASE BEST PRACTICES

Android™ Database Best Practices

About the Android Deep Dive Series

Zigurd Mednieks, Series Editor

The Android Deep Dive Series is for intermediate and expert developers who use Android Studio and Java, but do not have comprehensive knowledge of Android system-level programming or deep knowledge of Android APIs. Readers of this series want to bolster their knowledge of fundamentally important topics.

Each book in the series stands alone and provides expertise, idioms, frameworks, and engineering approaches. They provide in-depth information, correct patterns and idioms, and ways of avoiding bugs and other problems. The books also take advantage of new Android releases, and avoid deprecated parts of the APIs.

About the Series Editor

Zigurd Mednieks is a consultant to leading OEMs, enterprises, and entrepreneurial ventures creating Android-based systems and software. Previously he was chief architect at D2 Technologies, a voice-over-IP (VoIP) technology provider, and a founder of OpenMobile, an Android-compatibility technology company. At D2 he led engineering and product definition work for products that blended communication and social media in purpose-built embedded systems and on the Android platform. He is lead author of *Programming Android* and *Enterprise Android*.

Android™ Database Best Practices

Adam Stroud

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2016941977

ISBN-13: 978-0-13-443799-6
ISBN-10: 0-13-443799-3

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, July 2016

Publisher
Mark L. Taub

Executive Editor
Laura Lewin

Development Editor
Michael Thurston

Managing Editor
Sandra Schroeder

Full-Service Production Manager
Julie B. Nahil

Project Editor
codeMantra

Copy Editor
Barbara Wood

Indexer
Cheryl Lenser

Proofreader
codeMantra

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
codeMantra

❖

To my wife, Sabrina, and my daughters, Elizabeth and Abigail.
You support, inspire, and motivate me in everything you do.

❖

This page intentionally left blank

Contents in Brief

This page intentionally left blank

Contents

This page intentionally left blank

Preface

The explosion in the number of mobile devices in all parts of the word has led to an increase in both the number and complexity of mobile apps. What was once considered a platform for only simplistic applications now contains countless apps with considerable functionality. Because a mobile device is capable of receiving large amounts of data from multiple data sources, there is an increasing need to store and recall that data efficiently.

In traditional software systems, large sets of data are frequently stored in a database that can be optimized to both store the data as well as recall the data on demand. Android provides this same functionality and includes a database system, SQLite. SQLite provides enough power to support today's modern apps and also can perform well in the resource-constrained environment of most mobile devices. This book provides details on how to use the embedded Android database system. Additionally, the book contains advice inspired by problems encountered when writing "real-world" Android apps.

Who Should Read This Book

This book is written for developers who have at least some experience with writing Android apps. Specifically, an understanding of basic Android components (activities, fragments, intents, and the application manifest) is assumed, and familiarity with the Android threading model is helpful.

At least some knowledge of relational database systems is also helpful but is not necessarily a prerequisite for understanding the topics in this book.

How This Book Is Organized

This book begins with a discussion of the theory behind relational databases as well as some history of the relational model and how it came into existence. Next, the discussion moves to the Structured Query Language (SQL) and how to use SQL to build a database as well as manipulate and read a database. The discussion of SQL provides some details on Android specifics but generally discusses non-Android-specific SQL.

From there, the book moves on to provide information on SQLite and how it relates to Android. The book also covers the Android APIs that can be used to interact with a database as well as some best practices for database use.

With the basics of database, SQL, and SQLite covered, the book then moves into solving some of the problems app developers often face while using a database in Android. Topics such as threading, accessing remote data, and displaying data to the user are covered. Additionally, the book presents an example database access layer based on a content provider.

Following is an overview of each of the chapters:

- Chapter 1, "Relational Databases," provides an introduction to the relational database model as well as some information on why the relational model is more popular than older database models.
- Chapter 2, "An Introduction to SQL," provides details on SQL as it relates to databases in general. This chapter discusses the SQL language features for creating database structure as well as the features used to manipulate data in a database.
- Chapter 3, "An Introduction to SQLite," contains details of the SQLite database system, including how SQLite differs from other database systems.
- Chapter 4, "SQLite in Android," discusses the Android-specific SQLite details such as where a database resides for an app. It also discusses accessing a database from outside an app, which can be important for debugging.
- Chapter 5, "Working with Databases in Android," presents the Android API for working with databases and explains how to get data from an app to a database and back again.
- Chapter 6, "Content Providers," discusses the details around using a content provider as a data access mechanism in Android as well as some thoughts on when to use one.
- Chapter 7, "Databases and the UI," explains how to get data from the local database and display it to the user, taking into account some of the threading concerns that exist on Android.
- Chapter 8, "Sharing Data with Intents," discusses ways, other than using content providers, that data can be shared between apps, specifically by using intents.
- Chapter 9, "Communicating with Web APIs," discusses some of the methods and tools used to achieve two-way communication between an app and a remote Web API.
- Chapter 10, "Data Binding," discusses the data binding API and how it can be used to display data in the UI. In addition to providing an overview of the API, this chapter provides an example of how to view data from a database.

Example Code

This book includes a lot of source code examples, including an example app that is discussed in later chapters of the book. Readers are encouraged to download the example source code and manipulate it to gain a deeper understanding of the information presented in the text.

The example app is a Gradle-based Android project that should build and run. It was built with the latest libraries and build tools that were available at the time of this writing.

The source code for the example can be found on GitHub at https://github.com/android-database-best-practices/device-database. It is made available under the Apache 2 open-source license and can be used according to that license.

Conventions Used in This Book

The following typographical conventions are used in this book:

- `Constant width` is used for program listings, as well as within paragraphs to refer to program elements such as variable and function names, databases, data types, environment variables, statements, and keywords.
- **`Constant width bold`** is used to highlight sections of code.

> **Note**
>
> A Note signifies a tip, suggestion, or general note.

This page intentionally left blank

Acknowledgments

I have often believed that software development is a team sport. Well, I am now convinced that authoring is also a team sport. I would not have made it through this experience without the support, guidance, and at times patience of the team. I would like to thank executive editor Laura Lewin and editorial assistant Olivia Basegio for their countless hours and limitless e-mails to help keep the project on schedule.

I would also like to thank my development editor, Michael Thurston, and technical editors, Maija Mednieks, Zigurd Mednieks, and David Whittaker, for helping me transform my unfinished, random, and meandering thoughts into something directed and cohesive. The support of the team is what truly made this a rewarding experience, and it would not have been possible without all of you.

Last, I would like to thank my beautiful wife and wonderful daughters. Your patience and support have meant more than I can express.

This page intentionally left blank

About the Author

Adam Stroud is an Android developer who has been developing apps for Android since 2010. He has been an early employee at multiple start-ups, including Runkeeper, Mustbin, and Chef Nightly, and has led the Android development from the ground up. He has a strong passion for Android and open source and seems to be attracted to all things Android.

In addition to writing code, he has written other books on Android development and enjoys giving talks on a wide range of topics, including Android gaining root access on Android devices. He loves being a part of the Android community and getting together with other Android enthusiasts to "geek out."

Adam is currently the technical cofounder and lead Android developer at a new start-up where he oversees the development of the Android app.

This page intentionally left blank

1
Relational Databases

The relational database model is one of the more popular models for databases today. Android comes with a built-in database called SQLite that is designed around the relational database model. This chapter covers some of the basic concepts of a relational database. It starts with a brief history of databases, then moves to a discussion of the relational model. Finally, it covers the evolution of database languages. This chapter is meant for the reader who is largely unfamiliar with the concept of a relational database. Readers who feel comfortable with the concepts of a relational database can safely move on to chapters that discuss the unique features of the SQLite database system that comes bundled with Android.

History of Databases

Like other aspects of the world of computing, modern databases evolved over time. While we tend to talk about NoSQL and relational databases nowadays, it is sometimes important to know "how we got here" to understand why things work the way they do. This section of the chapter presents a little history of how the database evolved into what it is today.

> **Note**
>
> This section of the chapter presents information that may be of interest to some but seem superfluous to others. Feel free to move on to the next section to get into the details of how databases work on Android.

The problem of storing, managing, and recalling data is not a new one. Even decades before computers, people were storing, managing, and recalling data. It is easy to think of a paper-based system where important data was manually written, then organized and stored in a filing cabinet until it would need to be recalled. I need only to look in the corner of my basement to be reminded of the days when this was a common paradigm for data storage.

The paper-based approach has obvious limitations, the main one being its ability to scale as the amount of data grows. As the amount of data increases, so does the amount of time it takes to both manage the data store and recall data from the data store.

A paper-based approach also implies a highly manual process for data storage and retrieval, making it slow and error prone as well taking up a lot of space.

Early attempts to offload some of this process onto machines followed a very similar approach. The difference was that instead of using hard copies of the data written on paper, data was stored and organized electronically. In a typical electronic-file-based system, a single file would contain multiple entries of data that was somehow related to other data in the file.

While this approach did offer benefits over older approaches, it still had many problems. Typically, these file stores were not centralized. This led to large amounts of redundant data, which made processing slow and took large amounts of storage space. Additionally, problems with incompatible file formats were also frequent because there was rarely a common system in charge of controlling the data. In addition, there were often difficulties in changing the structure of the data as the usage of the data evolved over time.

Databases were an attempt to address the problems of decentralized file stores. Database technology is relatively new when compared to other technological fields, or even other areas of computer science. This is primarily because the computer itself had to evolve to a point where databases provided enough utility to justify their expense. It wasn't until the early to mid-1960s that computers became cheap enough to be owned by private entities as well as possess enough power and storage capacity to allow the concept of a database to be useful.

The first databases used models that are different from the relational model discussed in this chapter. In the early days, the two main models in widespread use were the network model and the hierarchical model.

Hierarchical Model

In the hierarchical model data is organized into a tree structure. The model maintains a one-to-many relationship between child and parent records with each child node having no more than one parent. However, each parent node may have multiple children. An initial implementation of the hierarchical model was developed jointly by IBM and Rockwell in the 1960s for the Apollo space program. This implementation was named the IBM Information Management System (IMS). In addition to providing a database, IMS could be used to generate reports. The combination of these two features made IMS one of the major software applications of its time and helped establish IBM as a major player in the computer world. IMS is still a widely used hierarchical database system on mainframes.

Network Model

The network model was another popular early database model. Unlike the hierarchical model, the network model formed a graph structure that removed the limitation of the one-to-many parent/child node relationship. This structure allowed the model to represent more complex data structures and relations. In addition, the network model was standardized by the Conference on Data Systems Language (CODASYL) in the late 1960s.

The Introduction of the Relational Model

The relational database model was introduced by Edgar Codd in 1970 in his paper "A Relational Model of Data for Large Shared Data Banks." The paper outlined some of the problems of the models of the time as well as introduced a new model for efficiently storing data. Codd went into details about how a relational model solved some of the shortcomings of the current models and discussed some areas where a relational model needed to be enhanced.

This was viewed as the introduction to relational databases and caused the idea to be improved and evolve into the relational database systems that we use today. While very few, if any, modern database systems strictly follow the guidelines that Codd outlined in his paper, they do implement most of his ideas and realize many of the benefits.

The Relational Model

The relational model makes use of the mathematical concept of a relation to add structure to data that is stored in a database. The model has a foundation based in set theory and first-order predicate logic. The cornerstone of the relational model is the relation.

Relation

In the relational model, conceptual data (the modeling of real-world data and its relationships) is mapped into relations. A relation can be thought of as a table with rows and columns. The columns of a relation represent its **attributes**, and the rows represent an entry in the table or a **tuple**. In addition to having attributes and tuples, the relational model mandates that the relation have a formal name.

Let's consider an example of a relation that can be used to track Android OS versions. In the relation, we want to model a subset of data from the Android dashboard (https://developer.android.com/about/dashboards/index.html). We will name this relation `os`.

The relation depicted in Table 1.1 has three attributes—`version`, `codename`, and `api`—representing the properties of the relation. In addition, the relation has four tuples tracking Android OS versions 5.1, 5.0, 4.4, and 4.3. Each tuple can be thought of as an entry in the relation that has properties defined by the relation attributes.

Table 1.1 The `os` Relation

version	codename	api
5.1	Lollipop	22
5.0	Lollipop	21
4.4	KitKat	19
4.3	Jelly Bean	18

Attribute

The attributes of a relation provide the data points for each tuple. In order to add structure to a relation, each attribute is assigned a **domain** that defines what data values can be represented by the attribute. The domain can place restrictions on the type of data that can be represented by an attribute as well as the range of values that an attribute can have. In the previous example, the `api` attribute is limited to the domain of integers and is said to be of type `integer`. Additionally, the domain of the `api` attribute can be further reduced to the set of positive integers (an upper bound can also be defined if the need arises).

The concept of a domain for a relation is important to the relational model as it allows the relation to establish constraints on attribute data. This becomes useful in maintaining data integrity and ensuring that the attributes of a relation are not misused. In the relation depicted in Table 1.1, a string `api` value could make certain operations difficult or allow operations to produce unpredictable results. Imagine adding a tuple to the `os` relation that contains a nonnumeric value for the `api` attribute, then asking the database to return all `os` versions with an `api` value that is greater than 19. The results would be unintuitive and possibly misleading.

The number of attributes in a relation is referred to as its **degree**. The relation in Table 1.1 has a degree of three because it has three attributes. A relation with a degree of one is called a **unary** relation. Similarly, a relation with a degree of two is **binary**, and a relation with a degree of three is called **ternary**. A relation with a degree higher than three is referred to as an **n-ary** relation.

Tuples

Tuples are represented by rows in the tabular representation of a relation. They represent the data of the relation containing values for the relation's attributes.

The number of tuples in a relation is called its **cardinality**. The relation in Table 1.1 has a cardinality of four since it contains four tuples.

An important point regarding a relation's cardinality and its degree is the level of volatility. A relation's degree helps define its structure and will change infrequently. A change in the degree is a change in the relation itself.

In contrast, a relation's cardinality will change with high frequency. Every time a tuple is added or removed from a relation, the relation's cardinality changes. In a large-scale database, the cardinality could change several times per second, but the degree may not change for days at a time, or indeed ever.

Intension/Extension

A relation's attributes and the attributes' domains and any other constraints on attribute values define a relation's **intension**. A relation's tuples define its **extension**. Since intension and extension are related to cardinality and degree respectively, it is easy to see that a relation's intension will also remain fairly static whereas it extension is dynamic, changing as tuples are added, deleted, and modified. A relation's degree is a property of its intension, and its cardinality is a property of its extension.

Schema

The structure of a relation is defined by its relational **schema**. A schema is a list of attributes along with the specification of the domain for those attributes. While the tabular form of a relation (Table 1.1) allows us to deduce the schema of a relation, a schema can also be specified in text. Here is the text representation of the schema from Table 1.1:

```
os(version, codename, api)
```

Notice the name of the relation along with the list of the attributes. In addition, the primary key is sometimes indicated with bold column names. Primary keys are discussed later in the chapter.

Properties of a Relation

Each relation in the relational model must follow a set of rules. These rules allow the relation to effectively represent real-world data models as well as address some of the limitations of older database systems. Relations that adhere to the following set of rules conform to a property known as the **first normal form**:

- **Unique name**: Each relation must have a name that uniquely identifies it. This allows the relation to be identified in the system.
- **Uniquely named attributes**: In addition to a uniquely named relation, each attribute in a relation must have a unique name. Much like the relation name, the attribute's unique name allows it to be identified.
- **Single-valued attributes**: Each attribute in a relation can have at most one value associated with it per tuple. In the example in Table 1.1, each `api` level attribute has only a single integer value. Including a tuple that has multiple values (19 and 20) is considered bad form.
- **Domain-limited attribute values**: As discussed previously, the value of each attribute for a tuple must conform to the attribute's domain. The domain for an attribute defines the attribute's "legal" values.
- **Unique tuples**: There should be no duplicate tuples in the relation. While there may be parts of a tuple that have common values for a subset of the relation's attributes, no two tuples should be identical.
- **Insignificant attribute ordering**: The order of the attributes in a relation has no effect on the representation of the relation of the tuples defined in the relation. This is because each attribute has a unique name that is used to refer to that attribute.

  For example, in Table 1.1, if the column ordering of the `codename` and `api` attributes were switched, the relation would remain the same. This is because the attributes are referred to by their unique names rather than their column ordering.
- **Insignificant tuple ordering**: The order of the tuples in a relation has no effect on the relation. While tuples can be added and removed, their ordering has no significance for the relation.

Relationships

Most conceptual data models require a relational model that contains multiple relations. Fortunately, the relational model allows relationships between multiple relations to be defined to support this. In order to define relationships between two relations, keys must be defined for them. A **key** is a set of attributes that uniquely identify a tuple in a relation. A key is frequently used to relate one relation to another and allows for complex data models to be represented as a relational model.

- **Superkey**: A superkey is a set of attributes that uniquely identify a tuple in a relation. There are no limits placed on the number of attributes used to form a superkey. This means that the set of all attributes should define a superkey that is used for all tuples.
- **Candidate key**: A candidate key is the smallest set of attributes that uniquely identify a tuple in a relation. A candidate key is like a superkey with a constraint placed on the maximum number of attributes. No subset of attributes from a candidate key should uniquely identify a tuple. There may be multiple candidate keys in a relation.
- **Primary key**: The primary key is a candidate key that is chosen to be the primary key. It holds all the properties of a candidate key but has the added distinction of being the primary key. While there may be multiple candidate keys in a relation that all uniquely identify a single row, there can be only *one* primary key.
- **Foreign key**: A foreign key is a set of attributes in a relation that map to a candidate key in another relation.

The foreign key is what allows two relations to be related to one another. Such relationships can be any of three different types:

- **One-to-one relationship**: The one-to-one relationship maps a single row in table A to a single row in table B. Additionally, the row in table B *only* maps back to the single row in table A (see Figure 1.1).

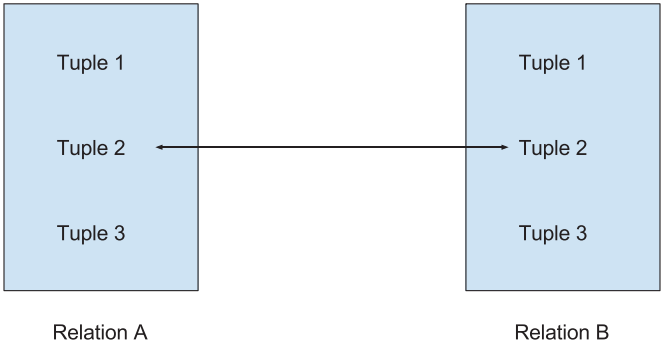

Figure 1.1 One-to-one relationship