Android™
Database Best Practices
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*. 
Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2016941977

Copyright © 2017 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearsoned.com/permissions/.

The following are registered trademarks of Google: Android™, Google Play™. Google and the Google logo are registered trademarks of Google Inc., used with permission.

The following are trademarks of HWACI: SQLite, sqlite.org, HWACI.

Gradle is a trademark of Gradle, Inc.

Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

Square is a registered trademark of Square, Inc.

Facebook is a trademark of Facebook, Inc.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

MySQL trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

The following are registered trademarks of IBM: IBM, IMS, Information Management System.

PostgreSQL is copyright © 1996-8 by the PostgreSQL Global Development Group, and is distributed under the terms of the Berkeley license.

Some images in the book originated from the sqlite.org and used with permission.

Twitter is a trademark of Twitter, Inc.


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

First printing, July 2016
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

Preface xv
Acknowledgments xix
About the Author xxi

1 Relational Databases 1
2 An Introduction to SQL 17
3 An Introduction to SQLite 39
4 SQLite in Android 47
5 Working with Databases in Android 79
6 Content Providers 101
7 Databases and the UI 137
8 Sharing Data with Intents 163
9 Communicating with Web APIs 177
10 Data Binding 231

Index 249
This page intentionally left blank
Contents

Preface xv
Acknowledgments xix
About the Author xxi

1 Relational Databases 1
   History of Databases 1
   Hierarchical Model 2
   Network Model 2
   The Introduction of the Relational Model 3
   The Relational Model 3
   Relation 3
   Properties of a Relation 5
   Relationships 6
   Relational Languages 9
   Relational Algebra 9
   Relational Calculus 13
   Database Languages 14
   ALPHA 14
   QUEL 14
   SEQUEL 14
   Summary 15

2 An Introduction to SQL 17
   Data Definition Language 17
   Tables 18
   Indexes 20
   Views 23
   Triggers 24
   Data Manipulation Language 28
   INSERT 28
   UPDATE 30
   DELETE 31
   Queries 32
   ORDER BY 32
   Joins 34
   Summary 37
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transactions</td>
<td>87</td>
</tr>
<tr>
<td>Using a Transaction</td>
<td>87</td>
</tr>
<tr>
<td>Transactions and Performance</td>
<td>88</td>
</tr>
<tr>
<td>Running Queries</td>
<td>89</td>
</tr>
<tr>
<td>Query Convenience Methods</td>
<td>89</td>
</tr>
<tr>
<td>Raw Query Methods</td>
<td>91</td>
</tr>
<tr>
<td>Cursors</td>
<td>91</td>
</tr>
<tr>
<td>Reading Cursor Data</td>
<td>91</td>
</tr>
<tr>
<td>Managing the Cursor</td>
<td>94</td>
</tr>
<tr>
<td>CursorLoader</td>
<td>94</td>
</tr>
<tr>
<td>Creating a CursorLoader</td>
<td>94</td>
</tr>
<tr>
<td>Starting a CursorLoader</td>
<td>97</td>
</tr>
<tr>
<td>Restarting a CursorLoader</td>
<td>98</td>
</tr>
<tr>
<td>Summary</td>
<td>99</td>
</tr>
<tr>
<td><strong>6 Content Providers</strong></td>
<td>101</td>
</tr>
<tr>
<td>REST-Like APIs in Android</td>
<td>101</td>
</tr>
<tr>
<td>Content URIs</td>
<td>102</td>
</tr>
<tr>
<td>Exposing Data with a Content Provider</td>
<td>102</td>
</tr>
<tr>
<td>Implementing a Content Provider</td>
<td>102</td>
</tr>
<tr>
<td>Content Resolver</td>
<td>108</td>
</tr>
<tr>
<td>Exposing a Remote Content Provider to</td>
<td>108</td>
</tr>
<tr>
<td>External Apps</td>
<td>108</td>
</tr>
<tr>
<td>Provider-Level Permission</td>
<td>109</td>
</tr>
<tr>
<td>Individual Read/Write Permissions</td>
<td>109</td>
</tr>
<tr>
<td>URI Path Permissions</td>
<td>109</td>
</tr>
<tr>
<td>Content Provider Permissions</td>
<td>110</td>
</tr>
<tr>
<td>Content Provider Contract</td>
<td>112</td>
</tr>
<tr>
<td>Allowing Access from an External App</td>
<td>114</td>
</tr>
<tr>
<td>Implementing a Content Provider</td>
<td>115</td>
</tr>
<tr>
<td>Extending android.content.ContentProvider</td>
<td>115</td>
</tr>
<tr>
<td>insert()</td>
<td>119</td>
</tr>
<tr>
<td>delete()</td>
<td>120</td>
</tr>
<tr>
<td>update()</td>
<td>122</td>
</tr>
<tr>
<td>query()</td>
<td>124</td>
</tr>
<tr>
<td>getType()</td>
<td>130</td>
</tr>
</tbody>
</table>
7 Databases and the UI 137
Getting Data from the Database to the UI 137
Using a Cursor Loader to Handle Threading 137
Binding Cursor Data to a UI 138
Cursors as Observers 143
registerContentObserver(ContentObserver) 143
registerDataSetObserver(DataSetObserver) 144
unregisterContentObserver
(ContentObserver) 144
unregisterDataSetObserver
(DataSetObserver) 144
setNotificationUri(ContentResolver, Uri uri) 145
Accessing a Content Provider from an Activity 145
Activity Layout 145
Activity Class Definition 147
Creating the Cursor Loader 148
Handling Returned Data 149
Reacting to Changes in Data 156
Summary 161
8 Sharing Data with Intents 163
Sending Intents 163
Explicit Intents 163
Implicit Intents 164
Starting a Target Activity 164
Receiving Implicit Intents 166
Building an Intent 167
Actions 168
Extras 168
Extra Data Types 169
What Not to Add to an Intent 172
ShareActionProvider 173
Share Action Menu 174
Summary 175
9 Communicating with Web APIs 177
   REST and Web Services 177
      REST Overview 177
      REST-like Web API Structure 178
   Accessing Remote Web APIs 179
      Accessing Web Services with Standard Android APIs 179
      Accessing Web Services with Retrofit 189
      Accessing Web Services with Volley 197
   Persisting Data to Enhance User Experience 206
      Data Transfer and Battery Consumption 206
      Data Transfer and User Experience 207
      Storing Web Service Response Data 207
   Android SyncAdapter Framework 207
      AccountAuthenticator 208
      SyncAdapter 212
   Manually Synchronizing Remote Data 218
      A Short Introduction to RxJava 218
      Adding RxJava Support to Retrofit 219
      Using RxJava to Perform the Sync 222
   Summary 229

10 Data Binding 231
   Adding Data Binding to an Android Project 231
   Data Binding Layouts 232
      Binding an Activity to a Layout 234
      Using a Binding to Update a View 235
      Reacting to Data Changes 238
   Using Data Binding to Replace Boilerplate Code 242
   Data Binding Expression Language 246
   Summary 247

Index 249
This page intentionally left blank
The explosion in the number of mobile devices in all parts of the world 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.

Register your copy of *Android™ Database Best Practices* at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780134437996) and click Submit. Once the process is complete, you will find any available bonus content under “Registered Products.”
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
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.
Working with Databases in Android

The previous chapter introduced the SQLiteOpenHelper and SQLiteDatabase classes and discussed how to create databases. Of course, this is only the first step as a database is not very useful until it contains data and allows software to run queries against that data. This chapter explains how that is done in Android by discussing which Android SDK classes can be used to manipulate a database as well as query a database.

Manipulating Data in Android

The Android SDK contains many classes to support database operations. Along with the classes to support create, read, update, and delete (CRUD) operations, the SDK contains classes to help generate the queries that read the database. Following are the classes introduced in this chapter and a summary of how they are used to work with databases in Android:

- **SQLiteDatabase**: Represents a database in Android. It contains methods to perform standard database CRUD operations as well as control the SQLite database file used by an app.
- **Cursor**: Holds the result set from a query on a database. An app can read the data from a cursor and display it to a user or perform business logic based on the data contained in the cursor.
- ** ContentValues**: A key/value store that inserts data into a row of a table. In most cases, the keys map to the column names of the table, and the values are the data to enter into the table.
- **CursorLoader**: Part of the loader framework that handles cursor objects.
- **LoaderManager**: Manages all loaders for an activity or fragment. The LoaderManager contains the API for initializing and resetting a loader that may be used by Android components.

Working with SQL is a vital part of working with databases in Android. In Chapter 2, “An Introduction to SQL,” we saw how SQL is used to both create and upgrade a
database. SQL can also be used to read, update, and delete information from a database in Android. The Android SDK provides useful classes to assist in creating SQL statements, while also supporting the use of Java string processing to generate SQL statements.

Working with SQL in Android involves calling methods on an SQLiteDatabase object. This class contains methods for building SQL statements as well as convenience methods to make issuing SQL statements to the database easy.

In a typical database use case, inserting data into the database is the step that follows creating the database. This makes sense since a database is useful only after it contains data. The steps to create a database were covered in the previous chapter, so this discussion starts with inserting data into a database.

**Inserting Rows into a Table**

The SQLiteDatabase class contains multiple convenience methods that can be used to perform insert operations. In most cases, one of the following three methods is used to perform an insert operation:

- `long insert(String table, String nullColumnHack, ContentValues values)`
- `long insertOrThrow(String table, String nullColumnHack, ContentValues values)`
- `long insertWithOnConflict(String table, String nullColumnHack, ContentValues values, int conflictAlgorithm)`

Notice that the parameter lists for all the variations of the insert methods contain (as the first three parameters) a `String tableName`, a `String nullColumnHack`, and `ContentValues values`. SQLiteDatabase.insertWithOnConflict() contains a fourth parameter which will be discussed soon. The common three parameters for the insert methods are

- `String table`: Gives the name of the table on which to perform the insert operation. This name needs to be the same as the name given to the table when it was created.
- `String nullColumnHack`: Specifies a column that will be set to null if the ContentValues argument contains no data.
- `ContentValues values`: Contains the data that will be inserted into the table.

ContentValues is a maplike class that matches a value to a String key. It contains multiple overloaded put methods that enforce type safety. Here is a list of the put methods supported by ContentValues:

- `void put(String key, Byte value)`
- `void put(String key, Integer value)`
- `void put(String key, Float value)`
- `void put(String key, Short value)`
Manipulating Data in Android

- void put(String key, byte[] value)
- void put(String key, String value)
- void put(String key, Double value)
- void put(String key, Long value)
- void put(String key, Boolean value)

Each put method takes a String key and a typed value as parameters. When using ContentValues to insert data into a database, the key parameter must match the name of the column for the table that is targeted by the insert.

In addition to the overloaded put methods just listed, there is also a put(ContentValues other) method that can be used to add all the values from another ContentValues object, and a putNull(String key) method that adds a null value to a column of a table.

In a typical use case, a new instance of ContentValues is created and populated with all the values that should be inserted into the table. The ContentValues object is then passed to one of the insert methods from SQLiteDatabase. Listing 5.1 shows typical ContentValues usage.

Listing 5.1 Inserting Data with SQLiteDatabase.insert()

```java
int id = 1;
String firstName = "Bob";
String lastName = "Smith";

ContentValues contentValues = new ContentValues();
contentValues.put("id", id);
contentValues.put("first_name", firstName);
contentValues.put("last_name", lastName);

SQLiteDatabase db = getDatabase();
db.insert("people", null, contentValues);
```

The code in Listing 5.1 passes a null for the value of the nullColumnHack to the SQLiteDatabase.insert() method. This is primarily because the code in Listing 5.1 "knows" what values were used to populate the values parameter and can ensure that there is at least one column represented in the ContentValues object. However, this is not always the case, and this is why the nullColumnHack parameter exists.

To explain nullColumnHack, consider the case where a ContentValues object that is inserted into a table contains no key/value pairs. This would amount to attempting to perform an insert operation without specifying any columns to insert data into. Such an insert statement is illegal in SQL because an insert statement must specify at least one
column to insert data into. The nullColumnHack parameter can be used to guard against the “empty ContentValues” use case by specifying the name of a column that should be set to null in the case that the ContentValues object contains no data. Like the keys in the ContentValues instance, the string value for nullColumnHack must match the name of a column in the table that is targeted by the insert statement.

Listing 5.2 contains a usage of the nullColumnHack parameter. After the code in Listing 5.2 is run, column last_name will contain a value of null.

**Listing 5.2 Specifying Null Columns with nullColumnHack**

```java
ContentValues contentValues = new ContentValues();
SQLiteDatabase db = getDatabase();
db.insert("people", "last_name", contentValues);
```

All three insert methods of SQLiteDatabase return a `long`. The value returned by the methods is the row ID of the inserted row, or a value of -1 if there was an error performing the insert.

Both Listings 5.1 and 5.2 used the simplest insert method to put a row into a table of the database, SQLiteDatabase.insert(). This method attempts to perform the insert and returns -1 if there is an error. The other two insert methods can be used to handle error cases differently.

SQLiteDatabase.insertOrThrow() is similar to SQLiteDatabase.insert(). However, it throws an SQLException if there was an error inserting the row. SQLiteDatabase.insertOrThrow() takes the same parameter list and has the same return type as SQLiteDatabase.insert(). It takes a String as the table parameter, a String as the nullColumnHack parameter, and a ContentValues object as the values parameter.

SQLiteDatabase.insertWithConflict(String table, String nullColumnHack, ContentValues values, int conflictAlgorithm) operates a little differently from the other two insert methods. It supports conflict resolution during the insert operation. Insertion conflicts occur when an attempt is made to insert a row into a table that would produce duplicates in a column that has the UNIQUE constraint applied to it, or duplicate data for the primary key. For example, consider the database table represented by Table 5.1.

<table>
<thead>
<tr>
<th>first_name</th>
<th>last_name</th>
<th>id*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>Smith</td>
<td>1</td>
</tr>
<tr>
<td>Ralph</td>
<td>Taylor</td>
<td>2</td>
</tr>
<tr>
<td>Sabrina</td>
<td>Anderson</td>
<td>3</td>
</tr>
<tr>
<td>Elizabeth</td>
<td>Hoffman</td>
<td>4</td>
</tr>
<tr>
<td>Abigail</td>
<td>Elder</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 5.1 Example Database Table
In Table 5.1, the id column is the primary key and must hold a unique value for all rows across the entire table. Therefore, an attempt to insert a row containing an id of 1 would be an illegal operation in SQL because it would cause a UNIQUE constraint violation.

In this scenario, the two previous insert methods would indicate the error by either returning a value of -1 (SQLiteDatabase.insert()) or throwing an exception (SQLiteDatabase.insertOrThrow()). However, SQLiteDatabase.insertWithOnConflict() takes a fourth int parameter that can be used to tell the method how to handle the insertion conflict. The conflict resolution algorithms are defined as constants in SQLiteDatabase and can be one of the following:

- SQLiteDatabase.CONFLICT_ROLLBACK: Aborts the current insert statement. If the insert was part of a transaction, any previous statements are also undone and the value of SQLiteDatabase.CONFLICT_FAIL is returned by the insertWithOnConflict() method.
- SQLiteDatabase.CONFLICT_ABORT: Aborts the current statement. If the statement was part of a transaction, all previous statements are left untouched.
- SQLiteDatabase.CONFLICT_FAIL: Similar to SQLiteDatabase.CONFLICT_ABORT. In addition to aborting the current statement, this flag causes the method to return SQLITE_CONSTRAINT as a return code.
- SQLiteDatabase.CONFLICT_IGNORE: Skips the current statement and all other statements in the transaction are processed. When using this flag, no error value is returned.
- SQLiteDatabase.CONFLICT_REPLACE: Removes conflicting rows currently in the table, and the new row is inserted. An error will not be returned when using this flag.
- SQLiteDatabase.NONE: No conflict resolution is applied.

**Updating Rows in a Table**

Once data has been inserted into a database, it often needs to be updated. Like the three insert methods discussed previously, SQLiteDatabase has a couple of update methods that can be used to perform update operations on tables in a database:

- int update(String table, ContentValues values, String whereClause, String[] whereArgs)
- int updateWithOnConflict(String table, ContentValues values, String whereClause, String[] whereArgs, int conflictAlgorithm)

Much like the insert methods, both update methods take the same first four parameters, and updateWithOnConflict() takes a fifth parameter to define how a conflict should be resolved.
The common parameters for the update methods are

- **String table**: Defines the name of the table on which to perform the update. As with the insert statements, this string needs to match the name of a table in the database schema.
- **ContentValues values**: Contains the key/value pairs that map the columns and values to be updated by the update statement.
- **String whereClause**: Defines the WHERE clause of an UPDATE SQL statement. This string can contain the “?” character that will be replaced by the values in the whereArgs parameter.
- **String[] whereArgs**: Provides the variable substitutions for the whereClause argument.

Listing 5.3 shows an example of the SQLiteDatabase.update() call.

**Listing 5.3 Example Update Call**

```java
String firstName = "Robert";

ContentValues contentValues = new ContentValues();
contentValues.put("first_name", firstName);

SQLiteDatabase db = getDatabase();
db.update("people", contentValues, "id = ?", new String[] {"1"});
```

Listing 5.3 updates the first name of the person that has an id of 1. The code first creates and populates a ContentValues object to hold the values that will be updated. It then makes the call to SQLiteDatabase.update() to issue the statement to the database. The rows are selected for the update() method using the whereClause and whereArgs parameters, which are in bold in Listing 5.3. The “?” in the whereClause parameter of the update() method serves as a placeholder for the statement. The whereArgs parameter, containing an array of strings, holds the value(s) that will replace the placeholder(s) when the statement is sent to the database. Since Listing 5.3 contains only a single placeholder, the string array only needs to be of size 1. When multiple placeholders are used, they will be replaced in order using the values from the string array. Passing null values for the whereClause and whereArgs parameters will cause the update statement to be run against every row in the table.

Table 5.2 shows the result of running the code in Listing 5.3 on Table 5.1. The changes to the row with id 1 are in bold.

The basic whereClause in Listing 5.3 matches the value of a single column. When using either update method, any legal SQL whereClause can be used to build the statement.
Both update methods in SQLiteDatabase return an integer that represents the number of rows that were affected by the update statement.

Replacing Rows in a Table

In addition to insert and update operations, SQLiteDatabase supports the SQL replace operation with the SQLiteDatabase.replace() methods. In SQLite, a replace operation is an alias for INSERT OR REPLACE. It inserts the row if it does not already exist in a table, or updates the row if it already exists.

**Note**
This is different from an update operation because an update operation does not insert a row if it does not already exist.

There are two versions of the replace() method in SQLiteDatabase: SQLiteDatabase.replace() and SQLiteDatabase.replaceOrThrow(). Both methods have the same parameter list:

- String table: The name of the table on which to perform the operation
- String nullColumnHack: The name of a column to set a null value in case of an empty ContentValues object
- ContentValues initialValues: The values to insert into the table

Both replace() methods return a long indicating the row ID of the new row, or a value of -1 if an error occurs. In addition, replaceOrThrow() can also throw an exception in the case of an error.

Listing 5.4 shows an example of the SQLiteDatabase.replace() call.

**Listing 5.4 Example Replace Call**

```java
String firstName = "Bob";

ContentValues contentValues = new ContentValues();
contentValues.put("first_name", firstName);
contentValues.put("id", 1);
```
SQLiteDatabase db = getDatabase();
db.replace("people", null, contentValues);

Table 5.3 shows the state of the people table after running the SQLiteDatabase.
replace() call in Listing 5.4. Notice that the last_name attribute for the first row is
now blank. This is because there was a conflict when processing the SQLiteDatabase.
replace() method. The ContentValues object passed to SQLiteDatabase.replace()
specified a value of 1 for the id attribute. The conflict arises because the id attribute
is the primary key for the table, and there is already a row that contains an id of 1. To
resolve the conflict, the SQLiteDatabase.replace() method removes the conflicting
row and inserts a new row containing the values specified in the ContentValues object.
Because the ContentValues object passed to SQLiteDatabase.replace() contains
values for only the first_name and id attributes, only those attributes are populated in
the new row.

Deleting Rows from a Table
Unlike the update and insert operations, SQLiteDatabase has only a single method
for deleting rows: SQLiteDatabase.delete(String table, String whereClause,
String[] whereArgs). The delete() method's signature is similar to the signature of
the update() method. It takes three parameters representing the name of the table from
which to delete rows, the whereClause, and a string array of whereArgs. The processing
of the whereClause and the whereArgs for the delete() method matches the
whereClause processing for the update() method. The whereClause parameter contains
question marks as placeholders, and the whereArgs parameter contains the values for the
placeholders. Listing 5.5 shows a delete() method example.

Listing 5.5 Example Delete Method
SQLiteDatabase db = getDatabase();
db.delete("people", "id = ?", new String[] {"1"});

<table>
<thead>
<tr>
<th>first_name</th>
<th>last_name</th>
<th>id*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Ralph</td>
<td>Taylor</td>
<td>2</td>
</tr>
<tr>
<td>Sabrina</td>
<td>Anderson</td>
<td>3</td>
</tr>
<tr>
<td>Elizabeth</td>
<td>Hoffman</td>
<td>4</td>
</tr>
<tr>
<td>Abigail</td>
<td>Elder</td>
<td>5</td>
</tr>
</tbody>
</table>
The results of running the code in Listing 5.5 are shown in Table 5.4, where there is no longer a row with an id of 1.

**Transactions**

All of the previously discussed insert, update, and delete operations manipulate tables and rows in a database. While each operation is atomic (will either succeed or fail on its own), it is sometimes necessary to group a set of operations together and have the set of operations be atomic. There are times when a set of related operations should be allowed to manipulate the database only if all operations succeed to maintain database integrity. For these cases, a database transaction is usually used to ensure that the set of operations is atomic. In Android, the SQLiteDatabase class contains the following methods to support transaction processing:

- void beginTransaction(): Begins a transaction
- void setTransactionSuccessful(): Indicates that the transaction should be committed
- void endTransaction(): Ends the transaction causing a commit if setTransactionSuccessful() has been called

**Using a Transaction**

A transaction is started with the SQLiteDatabase.beginTransaction() method. Once a transaction is started, calls to any of the data manipulation method calls (insert(), update(), delete()) may be made. Once all of the manipulation calls have been made, the transaction is ended with SQLiteDatabase.endTransaction(). To mark the transaction as successful, allowing all the operations to be committed, SQLiteDatabase.setTransactionSuccessful() must be called before the call to SQLiteDatabase.endTransaction() is made. If endTransaction() is called without a call to setTransactionSuccessful(), the transaction will be rolled back, undoing all of the operations in the transaction.

Because the call to setTransactionSuccessful() affects what happens during the endTransaction() call, it is considered a best practice to limit the number of non-database operations between a call to setTransactionSuccessful() and endTransaction(). Additionally, do not perform any additional database manipulation

<table>
<thead>
<tr>
<th>first_name</th>
<th>last_name</th>
<th>id*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ralph</td>
<td>Taylor</td>
<td>2</td>
</tr>
<tr>
<td>Sabrina</td>
<td>Anderson</td>
<td>3</td>
</tr>
<tr>
<td>Elizabeth</td>
<td>Hoffman</td>
<td>4</td>
</tr>
<tr>
<td>Abigail</td>
<td>Elder</td>
<td>5</td>
</tr>
</tbody>
</table>
operations between the call to `setTransactionSuccessful()` and `endTransaction()`. Once the call to `setTransactionSuccessful()` is made, the transaction is marked as clean and is committed in the call to `endTransaction()` even if errors have occurred after the call to `setTransactionSuccessful()`.

Listing 5.6 shows how a transaction should be started, marked successful, and ended in Android.

**Listing 5.6 Transaction Example**

```java
SQLiteDatabase db = getDatabase();
db.beginTransaction();

try {
    // insert/update/delete
    // insert/update/delete
    // insert/update/delete
    db.setTransactionSuccessful();
} finally {
    db.endTransaction();
}
```

Database operations that happen in a transaction as well as the call to `setTransaction()` should take place in a try block with the call to `endTransaction()` happening in a finally block. This ensures that the transaction will be ended even if an unhandled exception is thrown while modifying the database.

**Transactions and Performance**

While transactions can help maintain data integrity by ensuring that multiple data manipulation operations occur atomically, they can also be used purely to increase database performance in Android. Like any operation performed in Java, there is overhead that is associated with running SQL statements inside a transaction. While a single transaction may not inject large amounts of overhead into a data manipulation routine, it is important to remember that every call to `insert()`, `update()`, and `delete()` is performed in its own transaction. Thus inserting 100 records into a table would mean that 100 individual transactions will get started, cleaned, and closed. This can cause a severe slowdown when attempting to perform a large number of data manipulation method calls.

To make multiple data manipulation calls run as fast as possible, it is generally a good idea to combine them into a single transaction manually. If the Android SDK determines that a call to `insert()`, `update()`, `delete()` is already inside of an open transaction, it will not attempt to start another transaction for the single operation. With a few lines
of code, an app can dramatically speed up data manipulation operations. It is common to see a speed increase of five to ten times when wrapping even 100 data manipulation operations into a single transaction. These performance gains can increase as the number and complexity of operations increase.

**Running Queries**

Previous sections of this chapter discussed inserting, updating, and deleting data from a database. The last piece of database CRUD functionality is retrieving data from the database. As with the insert and update database operations, `SQLiteDatabase` contains multiple methods to support retrieving data. In addition to a series of query convenience methods, `SQLiteDatabase` includes a set of methods that support more free-form “raw” queries that can be generated via standard Java string manipulation methods. There is also an `SQLiteQueryBuilder` class that can further aid in developing complex queries such as joins.

**Query Convenience Methods**

The simplest way to issue a query to a database in Android is to use one of the query convenience methods located in `SQLiteDatabase`. These methods are the overloaded variations of `SQLiteDatabase.query()`. Each variant of the `query()` method takes a parameter list that includes the following:

- **String table**: Indicates the table name of the query.
- **String[] columns**: Lists the columns that should be included in the result set of the query.
- **String selection**: Specifies the `WHERE` clause of the selection statement. This string can contain “?” characters that can be replaced by the `selectionArgs` parameter.
- **String[] selectionArgs**: Contains the replacement values for the “?” of the selection parameter.
- **String groupBy**: Controls how the result set is grouped. This parameter represents the `GROUP BY` clause in SQL.
- **String having**: Contains the `HAVING` clause from an SQL `SELECT` statement. This clause specifies search parameters for grouping or aggregate SQL operators.
- **String orderBy**: Controls how the results from the query are ordered. This defines the `ORDER BY` clause of the `SELECT` statement.

The table name, column list selection string, and selection arguments parameters operate in the same manner as other operations discussed earlier in the chapter. What is different about the `query()` methods is the inclusion of the `GROUP BY`, `HAVING`, and `ORDER BY` clauses. These clauses allow an app to specify additional query attributes in the same way that an SQL `SELECT` statement would.
Each query method returns a cursor object that contains the result set for the query. Listing 5.7 shows a query returning data from the people table used in previous listings.

**Listing 5.7 Simple Query**

```java
SQLiteDatabase db = getDatabase();

Cursor result = db.query("people",
    new String[] {"first_name", "last_name"},
    "id = ?",
    new String[] {"1"},
    null,
    null,
    null);
```

Listing 5.7 returns the first_name and last_name columns for the row that has an id of 1. The query statement passes null values for the GROUP BY, HAVING, and ORDER BY clauses since the result set should be of size 1 and these clauses have no effect on a result set with size 1.

The query() method also supports passing a null value for the columns parameter which will cause the query to return all the table’s columns in the result set. It is usually better to specify the desired table columns rather than letting the Android SDK return all columns from a table and making the caller ignore the columns it does not need.

To return all the rows from a table, pass null values for the selection and selectionArgs parameters. A query returning all rows in a table is shown in Listing 5.8; the result set is sorted by ID in descending order.

**Listing 5.8 Returning All Rows in a Table**

```java
SQLiteDatabase db = getDatabase();

Cursor result = db.query("people",
    new String[] {"first_name", "last_name"},
    null,
    null,
    null,
    null,
    "id DESC");
```
**Raw Query Methods**

If the `query()` convenience methods do not provide enough flexibility for a query that an app needs to run, the `SQLiteDatabase.rawQuery()` methods can be used instead. Like the convenience query methods, the `rawQuery()` methods are an overloaded set of methods. However, unlike the `query()` methods, the `rawQuery()` methods take two parameters as input: a `String` parameter representing the query to run, and a `String[]` to support query placeholder substitution. Listing 5.9 shows the same query as Listing 5.6 using the `rawQuery()` method instead of the `query()` convenience method.

**Listing 5.9 Using the `rawQuery()` Method**

```java
SQLiteDatabase db = getDatabase();
Cursor result = db.rawQuery("SELECT first_name, last_name " +
    "FROM people " +
    "WHERE id = ?",
new String[] {"1"});
```

Like the `query()` method, `rawQuery()` returns a cursor containing the result set for the query. The caller can read and process the resulting cursor in the same way that it processes the result from the `query()` methods.

The `rawQuery()` method allows an app to have great flexibility and construct more complex queries using joins, sub-queries, unions, or any other SQL construct supported by SQLite. However, it also forces the app developer to build the query in Java code (or perhaps from reading a string resource), which can be cumbersome for really complex queries.

To aid in building more complex queries, the Android SDK contains the `SQLiteQueryBuilder` class. The `SQLiteQueryBuilder` class is discussed in more detail in the next chapter with the discussion of Content Providers.

**Cursors**

Cursors are what contain the result set of a query made against a database in Android. The `Cursor` class has an API that allows an app to read (in a type-safe manner) the columns that were returned from the query as well as iterate over the rows of the result set.

**Reading Cursor Data**

Once a cursor has been returned from a database query, an app needs to iterate over the result set and read the column data from the cursor. Internally, the cursor stores the rows of data returned by the query along with a position that points to the current row of data in the result set. When a cursor is returned from a `query()` method, its position points to the spot before the first row of data. This means that before any rows of data can be read from the cursor, the position must be moved to point to a valid row of data.
The `Cursor` class provides the following methods to manipulate its internal position:

- `boolean Cursor.move(int offset)`: Moves the position by the given offset
- `boolean Cursor.moveToFirst()`: Moves the position to the first row
- `boolean Cursor.moveToLast()`: Moves the position to the last row
- `boolean Cursor.moveToNext()`: Moves the cursor to the next row relative to the current position
- `boolean Cursor.moveToPosition(int position)`: Moves the cursor to the specified position
- `Cursor.moveToPrevious()`: Moves the cursor to the previous row relative to the current position

Each `move()` method returns a `boolean` to indicate whether the operation was successful or not. This flag is useful for iterating over the rows in a cursor.

Listing 5.10 shows the code to read data from a cursor containing all the data from the `people` table.

**Listing 5.10 Reading Cursor Data**

```java
SQLiteDatabase db = getDatabase();

String[] columns = {"first_name",
    "last_name",
    "id"};

Cursor cursor = db.query("people",
    columns,
    null,
    null,
    null,
    null,
    null);

while(cursor.moveToNext()) {
    int index;

    index = cursor.getColumnIndexOrThrow("first_name");
    String firstName = cursor.getString(index);
```
The code in Listing 5.10 uses a `while` loop to iterate over the rows in the cursor returned from the `query()` method. This pattern is useful if the code performing the iteration “controls” the cursor and has sole access to it. If other code can access the cursor (for example, if the cursor is passed into a method as a parameter), the cursor should also be set to a known position as the current position may not be the position ahead of the first row.

Once the cursor’s position is pointing to a valid row, the columns of the row can be read from the cursor. To read the data, the code in Listing 5.10 uses two methods from the `Cursor` class: `Cursor.getColumnIndexOrThrow()` and one of the type `get()` methods from the `Cursor` class.

The `Cursor.getColumnIndexOrThrow()` method takes a `String` parameter that indicates which column to read from. This `String` value needs to correspond to one of the strings in the `columns` parameter that was passed to the `query()` method. Recall that the `columns` parameter determines what table columns are part of the result set. `Cursor.getColumnIndexOrThrow()` throws an exception if the column name does not exist in the cursor. This usually indicates that the column was not part of the `columns` parameter of the `query()`. The `Cursor` class also contains a `Cursor.getColumnIndex()` method that does not throw an exception if the column name is not found. Instead, `Cursor.getColumnIndex()` returns a `-1` value to represent an error.

Once the column index is known, it can be passed to one of the cursor’s `get()` methods to return the typed data of the row. The `get()` methods return the data from the column in the row which can then be used by the app. The `Cursor` class contains the following methods for retrieving data from a row:

- `byte[] Cursor.getBlob(int columnIndex)`: Returns the value as a `byte[]`
- `double Cursor.getDouble(int columnIndex)`: Returns the value as a `double`
- `float Cursor.getFloat(int columnIndex)`: Returns the value as a `float`
- `int Cursor.getInt(int columnIndex)`: Returns the value as an `int`
- `long Cursor.getLong(int columnIndex)`: Returns the value as a `long`
- `short Cursor.getShort(int columnIndex)`: Returns the value as a `short`
- `String Cursor.getString(int columnIndex)`: Returns the value as a `String`
Managing the Cursor

The internals of a cursor can contain a lot of resources such as all the data returned from the query along with a connection to the database. Because of this, it is important to handle a cursor appropriately and tell it to clean up when it is no longer in use to prevent memory leaks. To perform the cleanup, the Cursor class contains the `Cursor.close()` method, which needs to be called when an activity or fragment no longer needs the cursor.

In versions of Android before 3.0, cursor maintenance was left to developers. They either had to handle the closing of the cursor themselves or had to make sure they informed an activity that it was using a cursor so the activity would close the cursor at an appropriate time.

Android 3.0 introduced the loader framework that takes care of managing cursors for activities/fragments. To support older versions of Android, the loader framework has also been backported and added to the support library. When using the loader framework, apps no longer need to worry about calling `Cursor.close()` or informing an activity/fragment of a cursor that it needs to manage.

CursorLoader

The previous section discussed the low-level details of how to perform database operations in Android using `SQLiteDatabase`. However, it did not discuss the fact that databases on Android are stored on the file system, meaning that accessing a database from the main thread should be avoided in order to keep an app responsive for the user. Accessing a database from a non-UI thread typically involves some type of asynchronous mechanism, where a request for database access is made and the response to the request is delivered at some point in the future. Because views can be updated only from the UI thread, apps need to make calls to update views on the UI thread even though the results to a database query may be delivered on a different thread.

Android provides multiple tools for executing potentially long-running code off the UI thread while having results processed in the UI thread. One such tool is the loader framework. For accessing databases, there is a specialized component of the Loader called CursorLoader, which, in addition to managing a cursor’s lifecycle with regard to an activity lifecycle, also takes care of running queries in a background thread and presenting the results on the main thread, making it easy to update the display.

Creating a CursorLoader

There are multiple pieces to the CursorLoader API. A CursorLoader is a specialized member of Android’s loader framework specifically designed to handle cursors. In a typical implementation, a CursorLoader uses a `ContentProvider` to run a query against a database, then returns the cursor produced from the `ContentProvider` back to an activity or fragment.
Note

ContentProviders are discussed in detail in Chapter 6, “Content Providers.” For now, it is enough to know that they abstract the functionality provided by SQLiteDatabase away from an activity (or fragment) so the activity does not need to worry about making method calls on an SQLiteDatabase object.

An activity only needs to use the LoaderManager to start a CursorLoader and respond to callbacks for CursorLoader events.

In order to use a CursorLoader, an activity gets an instance of the LoaderManager. The LoaderManager manages all loaders for an activity or fragment, including a CursorLoader.

Once an activity or fragment has a reference to its LoaderManager, it tells the LoaderManager to initialize a loader by providing the LoaderManager with an object that implements the LoaderManager.LoaderCallbacks interface in the LoaderManager.initLoader() method. The LoaderManager.LoaderCallbacks interface contains the following methods:

- Loader<T> onCreateLoader(int id, Bundle args)
- void onLoadFinished(Loader<T>, T data)
- void onLoaderReset(Loader<T> loader)

LoaderCallbacks.onCreate() is responsible for creating a new loader and returning it to the LoaderManager. To use a CursorLoader, LoaderCallbacks.onCreate() creates, initializes, and returns a CursorLoader object that contains the information necessary to run a query against a database (through a ContentProvider).

Listing 5.11 shows the implementation of the onCreateLoader() method returning a CursorLoader.

Listing 5.11 Implementing onCreateLoader()

```java
@override
public Loader<Cursor> onCreateLoader(int id, Bundle args) {
    Loader<Cursor> loader = null;
    switch (id) {
        case LOADER_ID_PEOPLE:
            loader = new CursorLoader(this,
                PEOPLE_URI,
                new String[] {"first_name", "last_name", "id"},
                null,
                null,
                "id ASC");
    }
    return loader;
}
```
In Listing 5.11, the onCreateLoader() method first checks the ID it was passed to know which loader it needs to create. It then instantiates a new CursorLoader object and returns it to the caller.

The constructor of CursorLoader can take parameters that allow the CursorLoader to run a query against a database. The CursorLoader constructor called in Listing 5.11 takes the following parameters:

- **Content context**: Provides the application context needed by the loader
- **Uri uri**: Defines the table against which to run the query
- **String[] projection**: Specifies the SELECT clause for the query
- **String selection**: Specifies the WHERE clause which may contain “?” as placeholders
- **String[] selectionArgs**: Defines the substitution variables for the selection placeholders
- **String sortOrder**: Defines the ORDER BY clause for the query

The last four parameters, projection, selection, selectionArgs, and sortOrder, are similar to parameters passed to the SQLiteDatabase.query() discussed earlier in this chapter. In fact, they also do the same thing: define what columns to include in the result set, define which rows to include in the result set, and define how the result set should be sorted.

Once the data is loaded, Loader.Callbacks.onLoadFinished() is called, allowing the callback object to use the data in the cursor. Listing 5.12 shows a call to onLoadFinished().

**Listing 5.12 Implementing onLoadFinished()**

```java
@Override
public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
    while(data.moveToNext()) {
        int index;

        index = data.getColumnIndexOrThrow("first_name");
        String firstName = data.getString(index);
```
Notice how similar the code in Listing 5.12 is to the code in Listing 5.10 where a direct call to SQLiteDatabase.query() was made. The code to process the results of the query is nearly identical. Also, when using the LoaderManager, the activity does not need to worry about calling Cursor.close() or making the database query on a non-UI thread. That is all handled by the loader framework.

There is one other important point to note about onLoadFinished(). It is not only called when the initial data is loaded; it is also called when changes to the data are detected by the Android database. There is one line of code that needs to be added to the ContentProvider to trigger this, and that is discussed next chapter. However, having a single point in the code that receives query data and can update the display can be really convenient. This architecture allows activities to easily react to changes in data without the developer worrying about explicitly notifying the activities of changes to the data. The LoaderManager handles the lifecycle and knows when to requery and pass the data to the LoaderManager.Callbacks when it needs to.

There is one more method in the LoaderManager.Callbacks interface that needs to be implemented to use a CursorLoader: LoaderManager.Callbacks.onLoaderReset(Loader<T> loader). This method is called by the LoaderManager when a loader that was previously created is reset and its data should no longer be used. For a CursorLoader, this typically means that any references to the cursor that was provided by onLoadFinished() need to be discarded as they are no longer active. If a reference to the cursor is not persisted, the onLoadReset() method can be empty.

**Starting a CursorLoader**

Now that the mechanics of using a CursorLoader have been discussed, it is time to focus on how to start a data load operation with the LoaderManager. For most use cases, an activity or a fragment implements the LoaderManager.Callbacks interface since it makes sense for the activity/fragment to process the cursor result in order to update its display. To start the load, LoaderManager.initLoader() is called. This ensures that the loader is created, calling onCreateLoader(), loading the data, and making a call to onLoadFinished().

```java
index = data.getColumnIndexOrThrow("last_name");
String lastName = data.getString(index);

index = data.getColumnIndexOrThrow("id");
long id = data.getLong(index);

//... do something with data
}
```
Both activities and fragments can get their LoaderManager object by calling getLoaderManager(). They can then start the load process by calling LoaderManager. initLoader(). initLoader() takes the following parameters:

- int id: The ID of the loader. This is the same ID that is passed to onCreateLoader() and can be used to identify a loader (see Listing 5.11).
- Bundle args: Extra data that might be needed to create the loader. This is also passed to onCreateLoader() (see Listing 5.11). This value can be null.
- LoaderManager.LoaderCallbacks callbacks: An object to handle the LoaderManager callbacks. This is typically the activity or fragment that is making the call to initLoader().

The call to initLoader() should happen early in an Android component's lifecycle. For activities, initLoader() is usually called in onCreate(). Fragments should call initLoader() in onActivityCreated() (calling initLoader() in a fragment before its activity is created can cause problems).

Once initLoader() is called, the LoaderManager checks to see if there is already a loader associated with the ID passed to initLoader(). If there is no loader associated with the ID, LoaderManager makes a call to onCreateLoader() to get the loader and associate it with the ID. If there is currently a loader associated with the ID, initLoader() continues to use the preexisting loader object. If the caller is in the started state, and there is already a loader associated with the ID, and the associated loader has already loaded its data, then a call to onLoadFinished() is made directly from initLoader(). This usually happens only if there is a configuration change.

One detail to note about initLoader() is that it cannot be used to alter the query that was used to create the CursorLoader that gets associated with an ID. Once the loader is created (remember, the query is used to define the CursorLoader), it is reused only on subsequent calls to initLoader(). If an activity/fragment needs to alter the query that was used to create a CursorLoader with a given ID, it needs to make a call to restartLoader().

**Restarting a CursorLoader**

Unlike the call to LoaderManager.initLoader(), a call to LoaderManager. restartLoader() disassociates a loader with a given ID and allows it to be re-created. This results in onCreateLoader() being called again, allowing a new CursorLoader object to be made which can contain a different query for a given ID. LoaderManager. restartLoader() takes the same parameter list as initLoader() (int id, Bundle, args, LoaderManager. Callbacks, and callbacks) and discards the old loader. This makes restartLoader() useful for when the query of a CursorLoader needs to change. However, the restartLoader() method should not be used to simply handle activity/fragment lifecycle changes as they are already handled by the LoaderManager.
Summary

This chapter presented the basic API for working with databases in Android and built upon the concepts introduced in Chapter 4, “SQLite in Android,” where database creation was discussed. By using SQLiteDatabase and its create(), insert(), update(), replace(), and delete() methods, an app is able to manipulate an internal database.

In addition, an app can call the query and rawQuery() methods to retrieve the data from a database to perform actions on that data, or just display it to a user.

Query data is returned in the form of a cursor that can be iterated over to access the result set returned by a query.

While this chapter introduced some of the low-level “plumbing” needed to use an in-app database, there are higher-level components that allow apps to both abstract some of the data access details away from components that define and drive user interaction (activities and fragments) as well as allow data to be shared across apps and across processes. These concepts are introduced in the next chapter with the discussion of content providers.
This page intentionally left blank
Symbols

?? (null coalescing operator), 247

A

accessing databases. See also content providers
with adb utility, 61–62
  connecting with sqlite3, 67–72
  copying files to development machines, 73
  finding file location, 64–67
  permissions, 62–64
  allowing external apps, 114–115
  main thread and, 60–61
  with Stetho, 73–75
  via Web services
    with Android SDK (Software Development Kit), 179–187
    REST and, 177–179
    with Retrofit, 188–194
    with Volley, 194–203
AccountAuthenticator class, 204–208
ACTION_SEND, 168
ACTION_SEND_MULTIPLE, 168
ActionProvider class, 174
actions in implicit intents, 167–168
activities
  accessing content providers
    activity class definition, 147–148
    activity layout, 145–147
    creating cursor loader, 148–149
    handling returned data, 149–156
    reacting to data changes, 156–161
    binding to layouts, 234–235
    starting, 164–166
  UI (user interface). See UI (user interface)
Activity class
  getIntent() method, 167
  onCreate() method, 167
  onCreateOptionsMenu() method, 174–175
adb (Android Debug Bridge) utility
  accessing databases, 61–62
    connecting with sqlite3, 67–72
    copying files to development machines, 73
    finding file location, 64–67
    permissions, 62–64
  viewing SQLite version, 40
with ListViews, 139–142
with RecyclerViews, 142, 145–147
BLOB storage class, 43
boilerplate code, replacing, 242–245
BR class, 238
bulkInsert() method, 105–108

C
candidate keys, 6
cardinality, 4
Cartesian product, 11
checkpoints, 41
close() method, 94
closing databases, 133–134
Codd, Edgar, 3
conflict resolution in insert operations, 82–83
connections
to HTTP servers, 179–184
with sqlite3, 67–72
candidates
constraint candidate keys, 6
foreign keys definition, 6
in SQLite, 40
keys, 6
primary keys, 6
REST, 177–178
superkeys, 6
constructors
CursorLoader class, 96
SQLiteOpenHelper class, 48–50
content observers, cursors as, 143
reacting to data changes, 156–161
registerContentObserver() method, 143–144
registerDataSetObserver() method, 144
unregisterContentObserver() method, 144
unregisterDataSetObserver() method, 144
content providers
accessing from activities
activity class definition, 147–148
activity layout, 145–147
creating cursor loader, 148–149
handling returned data, 149–156
reacting to data changes, 156–161
content resolvers, role of, 108
ContentProvider class, 102–103
applyBatch() method, 105–108
bulkInsert() method, 105–108
delete() method, 104
extend, 115–118
get(), method, 104–105
insert() method, 103–104
onCreate() method, 103
query() method, 105
update() method, 105
DevicesProvider implementation, 115
class declaration, 115–118
delete() method, 120–122
get(), method, 130–132
insert() method, 119–120

B
BaseActivity class, 147
battery consumption, Web services and, 203
beginTransaction() method, 87–88
binding. See also data binding library
activities to layouts, 234–235
cursor data to UI, 138
with ListViews, 139–142
with RecyclerViews, 142, 145–147
BLOB storage class, 43
boilerplate code, replacing, 242–245
BR class, 238
bulkInsert() method, 105–108

C
candidate keys, 6
cardinality, 4
Cartesian product, 11
checkpoints, 41
close() method, 94
closing databases, 133–134
Codd, Edgar, 3
conflict resolution in insert operations, 82–83
connections
to HTTP servers, 179–184
with sqlite3, 67–72
candidates
constraint candidate keys, 6
foreign keys definition, 6
in SQLite, 40
keys, 6
primary keys, 6
REST, 177–178
superkeys, 6
constructors
CursorLoader class, 96
SQLiteOpenHelper class, 48–50
content observers, cursors as, 143
reacting to data changes, 156–161
registerContentObserver() method, 143–144
registerDataSetObserver() method, 144
unregisterContentObserver() method, 144
unregisterDataSetObserver() method, 144
content providers
accessing from activities
activity class definition, 147–148
activity layout, 145–147
creating cursor loader, 148–149
handling returned data, 149–156
reacting to data changes, 156–161
content resolvers, role of, 108
ContentProvider class, 102–103
applyBatch() method, 105–108
bulkInsert() method, 105–108
delete() method, 104
extend, 115–118
get(), method, 104–105
insert() method, 103–104
onCreate() method, 103
query() method, 105
update() method, 105
DevicesProvider implementation, 115
class declaration, 115–118
delete() method, 120–122
get(), method, 130–132
insert() method, 119–120

B
BaseActivity class, 147
battery consumption, Web services and, 203
beginTransaction() method, 87–88
binding. See also data binding library
activities to layouts, 234–235
cursor data to UI, 138
with ListViews, 139–142
with RecyclerViews, 142, 145–147
BLOB storage class, 43
boilerplate code, replacing, 242–245
BR class, 238
bulkInsert() method, 105–108
CursorLoader class, 142–143
update() method, 122–124
exposing to external apps, 108–109
allowing access, 114–115
app-specific permissions, 110–112
contracts, 112–114
path permissions, 109–110
provider-level permissions, 109
read/write permissions, 109
finding file location, 64–67
limitations, 132–134
RESTful APIs compared, 101
strengths, 134–135
UI (user interface). See UI (user interface)
URL scheme conventions, 101–102

content resolvers, role of, 108
ContentProvider class, 102–103
applyBatch() method, 105–108
bulkInsert() method, 105–108
delete() method, 104
extending, 115–118
getType() method, 104–105
insert() method, 103–104
onCreate() method, 103
query() method, 105
update() method, 105

ContentResolver class, 159–160
ContentValues class, 79, 80–81
Context class
getResolver() method, 108
startActivity() method, 164
contracts for content providers, 112–114
converting view layouts to data binding layouts, 232–233
copying
databases to development machines, 73
tables, 59–60
CREATE INDEX statement, 22
CREATE TABLE statement, 18–19
CREATE TRIGGER statement, 24–27
CREATE VIEW statement, 23–24
createChooser() method, 164–166
CREATOR member variable, 172

Cursor class, 79, 92
managing cursors, 94. See also CursorLoader class
reading data, 92–94
registerContentObserver() method, 143–144
registerDataSetObserver() method, 144
setNotificationUri() method, 145
unregisterContentObserver() method, 144
unregisterDataSetObserver() method, 144
cursor loaders. See also CursorLoader class
content providers and, 134
creating, 95–98, 148–149
reacting to data changes, 156–161
restarting, 99
starting, 98–99
threads and, 137
CursorAdapter class, 140–142
CursorLoader class, 79, 94–95. See also cursor loaders

creating CursorLoaders, 95–98
restarting CursorLoaders, 99
starting CursorLoaders, 98–99
cursors
binding data to UI, 138
with ListViews, 139–142
with RecyclerViews, 142, 145–147
creating CursorLoaders, 95–98
definition, 92
intents and, 172–173
managing, 94. See also CursorLoader class
objects versus, 133
as observers, 143
reacting to data changes, 156–161
registerContentObserver() method, 143–144
registerDataSetObserver() method, 144
setNotificationUri() method, 145
unregisterContentObserver() method, 144
unregisterDataSetObserver() method, 144
ORM versus, 142
reading data, 92–94
restarting CursorLoaders, 99
starting CursorLoaders, 98–99
threads and, 94–95

D
data binding library, 231. See also binding
adding to projects, 200, 231
binding activities to layouts, 234–235
converting view layouts to data binding layouts, 232–233
expression language, 246–247
reacting to data changes, 238–242
replacing boilerplate code, 242–245
updating views, 235–238

Data Definition Language (DDL). See DDL (Data Definition Language)
Data Manipulation Language (DML). See DML (Data Manipulation Language)
data persistence, 47
for Web services, 204
AccountAuthenticator class, 204–208
manual synchronization with RxJava, 213–223
SyncAdapter class, 209–213
SyncAdapter framework, 204
data transfer. See Web services
data types
for intent extras, 169
for observable fields, 240
in SQLite, 43
storage classes, 43
type affinity, 44
databases
accessing. See accessing databases
closing, 133–134
hierarchical model, 2
history of, 1–2
languages, 14
ALPHA, 14
QUEL, 14
databases (continued)
    SEQUEL, 14
    SQL. See SQL (Structured Query Language)
    SQLite. See SQLite
    network model, 2
    relational model, 3
    attributes, 4
    first normal form, 5
    intension/extension, 4
    referential integrity, 7–9
    relational algebra, 9–13
    relational calculus, 13
    relational languages, 9
    relations, 3
    relationships, 6–7
    schemas, 5, 17
    tuples, 4
    upgrading. See upgrading databases
DataBindingUtil class, 234–235
DDL (Data Definition Language), 17
    for indexes, 20–21
        CREATE INDEX statement, 22
        DROP INDEX statement, 22–23
    for tables, 18
        ALTER TABLE statement, 19–20, 59
        CREATE TABLE statement, 18–19
        DROP TABLE statement, 20
    for triggers, 24
        CREATE TRIGGER statement, 24–27
        DROP TRIGGER statement, 27–28
    for views, 23
        CREATE VIEW statement, 23–24
        DROP VIEW statement, 24
    DEFAULT keyword in INSERT statements, 29–30
    degrees, 4
    delete() method
        ContentProvider class, 104
        DevicesProvider class, 120–122
        SQLiteDatabase class, 86–87
    DELETE statement, 31
    deleting table rows, 86–87, 104
    deliverSelfNotification() method, 143
DeviceCursorAdapter class
    getItemCount() method, 151–152
    implementation, 153–156
    onBindViewHolder() method, 152–153
    swapCursor() method, 151
DevicesOpenHelper implementation, 54–57
DevicesProvider implementation, 115
    class declaration, 115–118
    delete() method, 120–122
    getType() method, 130–132
    insert() method, 119–120
    query() method, 124–130, 156–159
    update() method, 122–124
DeviceViewHolder class, 153–155
    difference operator, 11
DML (Data Manipulation Language), 28
    DELETE statement, 31
    INSERT statement, 28–29
    DEFAULT keyword, 29–30
    SELECT statement in, 29
    VALUES keyword, 28–29
    UPDATE statement, 30–31
    domain relational calculus, 13
    domains, 4
    DROP INDEX statement, 22–23
    DROP TABLE statement, 20
    DROP TRIGGER statement, 27–28
    DROP VIEW statement, 24
    dropping
        indexes, 22–23
        tables, 20, 59–60
        triggers, 27–28
        views, 24
    endTransaction() method, 87–88
    explicit intents, 163
    expression language for data binding, 246–247
    extending ContentProvider class, 115–118
    extension, 4
    external apps
        exposing content providers to, 108–109
        allowing access, 114–115
        app-specific permissions, 110–112
        contracts, 112–114
        path permissions, 109–110
        provider-level permissions, 109
        read/write permissions, 109
        sharing data via intents, 164
        actions, 167–168
        extras, 168–169
        Parcelable interface, 170–172
        receiving implicit intents, 166–167
        ShareActionProvider class, 173–175
        starting target activities, 164–166
    EXTRA_STREAM, 169
    EXTRA_TEXT, 169
    extras in implicit intents, 168–169
    finding database file location, 64–67
    findViewById() method, 242
    first normal form, 5
    foreign keys
        definition, 6
        in SQLite, 40
    FROM clause in SELECT statements, 32
    FTS (full text search) in SQLite, 40–41
    full table scans, 20
    get() methods, 94
    getColumnIndex() method, 93–94
    getColumnIndexOrThrow() method, 93–94
    getIntent() method, 167
    getItemCount() method, 151–152
getReadableDatabase() method, 58
getResolver() method, 108
getType() method
ContentProvider class, 104–105
DevicesProvider class, 130–132
getWritableDatabase() method, 58
GSON library, 188, 190

H
hierarchical model of databases, 2
history of databases, 1–2
HTTP clients, OkHttp library, 190–193
HTTP servers, connecting to, 179–184
HttpURLConnection class, 179–180

I
IBM Information Management System (IMS), 2
IDs
adding to views, 242–245
in content URLs, 102
implicit intents, 164
actions, 167–168
extras, 168–169
Parcelable interface, 170–172
receiving, 166–167
ShareActionProvider class, 173–175
starting target activities, 164–166
IMS (IBM Information Management System), 2
indexes
CREATE INDEX statement, 22
definition, 20–21
DROP INDEX statement, 22–23
initLoader() method, 95, 98–99
insert() method
ContentProvider class, 103–104
DevicesProvider class, 119–120
SQLiteDatabase class, 80–83
INSERT statement, 28–29
DEFAULT keyword, 29–30
SELECT statement in, 29
VALUES keyword, 28–29
inserting
null columns, 81–82
table rows, 80–83, 103–104
insertOrThrow() method, 80, 82
insertWithOnConflict() method, 80, 82–83
INTEGER storage class, 43
intension, 4
intent class
actions, 167–168
createChooser() method, 164–166
eXtras, 168–169
putExtra() method, 164, 169
resolveActivity() method, 164
setType() method, 164
intent cursors and, 172–173
definition, 163
explicit intents, 163
implicit intents, 164
actions, 167–168
extras, 168–169
Parcelable interface, 170–172
receiving, 166–167
ShareActionProvider class, 173–175
starting target activities, 164–166
interprocess communication, 135
intersection operator, 10

J
Jackson, parsing JSON with, 197–200
joins, 12–13, 34–37, 42
journal mode, 41
JSON
Android APIs for, 184–187
parsing with Jackson, 197–200

K
keys, 6

L
languages, 14
ALPHA, 14
QUEL, 14
SEQUEL, 14
SQL. See SQL (Structured Query Language)
SQLite. See SQLite
<layout> element, 232–233
layouts
for activities, 145–147
binding activities to, 234–235
view layouts, converting to data binding layouts, 232–233
listings
AccountService in manifest, 207
ACTION_SEND_MULTIPLE, 168
activity with intent filter, 166
adb shell dumpsys subcommand, 65
adding
cancel support to AsyncTask, 182–184
data binding library to build.gradle, 200
data binding support to build.gradle, 231
manufacturer reference to device table, 36
new row to device table, 20
Retrofit to build.gradle, 188
RxJava adapter to Retrofit, 214–216
RxJava support to build.gradle, 214
Stetho to build.gradle, 74
transaction support to bulkInsert() and applyBatch(), 106–107
views with IDs, 242–244
Volley dependency, 195
Volley to settings.gradle, 195
attaching SyncAdapter with SyncService, 212
binding
cursor with CursorAdapter, 141
to framework with AuthenticatorService, 206
layout to activity, 234
DeviceCursorAdapter and DeviceViewHolder, 153–156
DeviceCursorAdapter.swapCursor(), 151
DevicesOpenHelper.onConfigure(), 54
DevicesOpenHelper.onCreate(), 50
DevicesOpenHelper.onUpgrade(), 51
DevicesProvider.onCreate(), 118
GetManufacturersAndDevicesRequest, 202–203
g getType(), 131
insert(), 119
notifyUris(), 160
ObservableDevice, 237
onCreateLoader(), 96
onLoadFinished(), 97
query(), 124–127
SQLiteOpenHelper constructor, 49
stub AccountAuthenticator, 204–206
SyncAdapter, 209–211
SyncManager.call(), 217–219
SyncManager.
update(), 122–123
VolleyApiClient, 196
inserting
data using contract class, 114
data with SQLiteDatabase.insert(), 81
manufacturers, 36
issuing .help to sqlite3, 67–69
joining tables with JOIN, 37
layout definition for DeviceListActivity, 145–146
<layout> element usage, 232–233
list_item_device.xml definition, 146
loading
devices with VolleyApiClient, 200–202
new cursor with onLoaderReset(), 151
making requests with AsyncTask, 180–182
making Retrofit call, 194
manually triggering SyncAdapter, 213
mapping UriMatcher, 117
null coalescing operator, 247
observable Web service call to DeviceService, 216
OkHttpLoggingInterceptor output, 191–193
onCreate() method implementation, 147–148
opening HttpURLConnection connections, 179–180
ordering rows with ORDER BY, 34
Parcelable implementation, 170–171
parsing JSON with JacksonRequest, 197–200
populating table with multiple INSERT statements, 30
processing
all rows with UPDATE, 31
cursor in onLoadFinished(), 149
protected call to Context.startActivity(), 165
pulling contact information with adb pull, 73
querying raw_contacts table, 70
reading cursor data, 92–93
removing
device table, 20
device_name view, 24
insert_date trigger, 27
rows with DELETE, 31
returning all rows in table, 90
number of items, 152
running .tables, 69–70
SELECT statement, 32
with WHERE clause, 32
sending JPEG extra, 169
updates from DevicesProvider.query(), 156–159
setting ObservableField values, 241
share action provider menu item, 174
simple query, 90
snippets of insert(), update(), and delete(), 159–160
specifying null columns with nullColumnHack, 82
standard SQL types, 44
SyncService manifest declaration, 212
transaction example, 88
UPDATE with WHERE clause, 31
updated ObservableDevice with ObservableField, 241
updating
bound view, 236
IDs, 244–245
layout to use ObservableDevice, 239–240
UI in onBindViewHolder(), 152–153
view from single row in cursor, 138
ListViews, 139–142
loader framework. See CursorLoader class
LoaderCallbacks interface, 95
onLoaderReset() method, 98, 150–151
onLoadFinished() method
binding data to UI, 138
creating cursor loaders, 97–98
processing cursors, 149–150
setting ObservableField values, 241
threads and, 137
updating views, 235–237
LoaderManager class, 79
creating CursorLoaders, 95–98
restarting CursorLoaders, 99
starting CursorLoaders, 98–99

M

main thread, database access and, 60–61
many-to-many relationships, 7
mapping URIs to tables, 117–118
menus, 174–175
MIME types, returning, 104–105
move() method, 92
moveToFirst() method, 92
moveToLast() method, 92
moveToNext() method, 92
moveToPosition() method, 92
moveToPrevious() method, 92
multithread support in SQLite, 42

N

n-ary relations, 4
natural joins, 12–13
network model of databases, 2
notifyChange() method, 159–160
notifyPropertyChanged() method, 238
notifyUris() method, 160–161
null coalescing operator, 247
null columns, inserting, 81–82
NULL storage class, 43

O

object-relational mapping (ORM), 142
objects, cursors versus, 133
observable fields
data types, 240
setting values, 241–242
updating views, 237–238
ObservableDevice class, 237–242
ObservableField class, 240–242
observers, cursors as. See content observers, cursors as
OkHttp library, 190–193
onBindViewHolder() method, 152–153
onChange() method, 143–144
onConfigure() method, 53–54
onCreate() method
Activity class, 167
ContentProvider class, 103
DeviceListActivity class, 147–148
DevicesOpenHelper class, 50
DevicesProvider class, 118
LoaderCallbacks interface, 95
SQLiteOpenHelper class, 50
onCreateLoader() method
DeviceListActivity class, 148–149
LoaderCallbacks interface, 96
onCreateOptionsMenu() method, 174–175
onDowngrade() method, 54
one-to-many relationships, 7
one-to-one relationships, 6
onInvalidate() method, 144
onLoaderReset() method, 98, 150–151
onLoadFinished() method
binding data to UI, 138
creating cursor loaders, 97–98
processing cursors, 149–150
setting ObservableField values, 241
threads and, 137
updating views, 235–237
onUpgrade() method
DevicesOpenHelper class, 51
SQLiteOpenHelper class, 50–53
operators
  Cartesian product, 11
  data binding expression language, 246–247
  difference, 11
  intersection, 10
  projection operation, 12
  selection operation, 11–12
  union, 10
ORDER BY clause in SELECT statements, 32–34
ORM (object-relational mapping), 142

P
Parceled interface, 170–172
path permissions, 109–110
paths in content URIs, 102
performance of transactions, 88–89
permissions, 62–64
  app-specific, 110–112
  path, 109–110
  provider-level, 109
  read/write, 109
persisting data. See data persistence
primary keys, 6
projection operation, 12, 32
provider-level permissions, 109
pull command (adb), 73
put() methods, 80–81
putExtra() method, 164, 169

Q
QUEL, 14
queries
  cursors
    creating CursorLoaders, 95–98
    definition, 92
    managing, 94. See also CursorLoader class
    reading data, 92–94
    restarting CursorLoaders, 99
    starting CursorLoaders, 98–99
    threads and, 94–95
  joins, 34–37
SELECT statement, 32–34
SQLiteDatabase class, 89–91
query() method, 89–91
rawQuery() method, 91
query() method
  ContentProvider class, 105
  DevicesProvider class, 124–130, 156–159
  SQLiteDatabase class, 89–91

R
R class, 238
rawQuery() method, 91
reading
  cursor data, 92–94
  parcels, 172
  threads and, 137
read/write permissions, 109
REAL storage class, 43
rebuilding database as upgrade method, 58
receiving implicit intents, 166–167
RecyclerView, 142, 145–147
referential integrity, 7–9
registerContentObserver() method, 143–144
registerDataSetObserver() method, 144
relational algebra
  Cartesian product, 11
  difference operator, 11
  intersection operator, 10
  joins, 12–13
  projection operation, 12, 32
  selection operation, 11–12
  union operator, 10
relational calculus
  definition, 9, 13
  domain relational calculus, 13
  tuple relational calculus, 13
relational languages, 9
  relational algebra, 9–10
    Cartesian product, 11
    difference operator, 11
    intersection operator, 10
    joins, 12–13
    projection operation, 12, 32
    selection operation, 11–12
    union operator, 10
  relational calculus, 13
    domain relational calculus, 13
    tuple relational calculus, 13
relational model of databases, 3
  relational languages, 9
    relational algebra, 9–13
    relational calculus, 13
  relations, 3
    attributes, 4
    first normal form, 5
    intension/extension, 4
    schemas, 5, 17
    tuples, 4
    relationships
      definition, 6–7
      referential integrity, 7–9
relations, 3
  attributes, 4
  definition, 3
  first normal form, 5
  intension/extension, 4
  relationships
    definition, 6–7
    referential integrity, 7–9
    schemas, 5, 17
    tuples, 4
relationships
  definition, 6–7
  referential integrity, 7–9
remote data transfer. See Web services
removing table rows, 86–87, 104
replace() method, 85–86
replaceOrThrow() method, 85
replacing boilerplate code, 242–245
table rows, 85–86
RequestQueue (Volley), 195–197
resolveActivity() method, 164
REST (Representational State Transfer) constraints, 177–178
Web services and, 177–179
restarting CursorLoaders, 99
restartLoader() method, 99
RESTful APIs
content providers compared, 101
structure of, 178–179
Retrofit, 188–194
adding RxJava support, 214–216
adding to projects, 188
configuring, 188–190
OkHttp library, 190–193
Web service calls, 193–194
Web service interface, 188
rowid column (SQLite), 21
rows
deleting, 86–87, 104
inserting, 80–83, 103–104
replacing, 85–86
updating, 83–85, 105
RxJava, 213–214
adding support to Retrofit, 214–216
SyncManager implementation, 216–223
S
schemas, 5, 17
schemes in content URIs, 102
SELECT statement, 32–34
in INSERT statements, 29
join, 34–37
ORDER BY clause, 32–34
selection operation, 11–12
SEQUEL, 14
serialized mode in SQLite, 42
setContentView() method, 234–235
setNotificationUri() method Cursor class, 145
setTransactionSuccessful() method, 87–88
setType() method, 164
ShareActionProvider class, 173–175
sharing data
with content providers. See content providers
with intents. See intents
SimpleCursorAdapter class, 139–140
single-thread mode in SQLite, 42
SQL (Structured Query Language), 14
DDL (Data Definition Language), 17
for indexes, 20–23
for tables, 18–20
for triggers, 24–28
for views, 23–24
DML (Data Manipulation Language), 28
DELETE statement, 31
INSERT statement, 28–30
UPDATE statement, 30–31
queries
joins, 34–37
SELECT statement, 32–34
SQLite
Android SDK. See Android SDK
(Software Development Kit) characteristics, 39
data persistence, 47
data types, 43
storage classes, 43
type affinity, 44
features, 39–40
atomic transactions, 41–42
foreign key support, 40
full text search, 40–41
multithread support, 42
limitations, 42–43
threads and database access, 60–61
upgrading databases, 58
by manipulating database, 59–60
by rebuilding database, 58
sqlite3 command, 67–72
SQLiteOpenHelper class, 47–48
constructors, 48–50
onConfigure() method, 53–54
onCreate() method, 50
onDowngrade() method, 54
onUpgrade() method, 50–53
SQLiteQueryBuilder class, 91, 128–130
startActivity() method, 164
starting
CursorLoaders, 98–99
target activities, 164–166
statements (SQL)
ALTER TABLE, 19–20
in SQLite, 43
upgrading databases, 59
CREATE INDEX, 22
CREATE TABLE, 18–19
CREATE TRIGGER, 24–27
CREATE VIEW, 23–24
DELETE, 31
DROP INDEX, 22–23
DROP TABLE, 20
transactions
  in content providers, 105–108
  methods, 87–88
  performance, 88–89
  in SQLite, 41–42

triggers
  CREATE TRIGGER statement, 24–27
  definition, 24
  DROP TRIGGER statement, 27–28
  warning about, 28

tuple relational calculus, 13

tuples, 3, 4

type affinity in SQLite, 44

UI (user interface)
  binding cursor data to, 138
  with ListViews, 139–142
  with RecyclerViews, 142, 145–147

unary relations, 4

union operator, 10

update()
methode, 83–85

upgrading databases
  by manipulating database, 59–60
  onUpgrade() method, 50–53
  by rebuilding database, 58

URIs
  mapping to tables, 117–118
  scheme conventions, 101–102

URL scheme conventions, 101

user experience, Web services and, 203–204

user interface. See UI (user interface)
DROP VIEW statement, 24
in SQLite, 42
upating
with data binding, 235–238
with ListViews, 139–142
from onLoadFinished() method, 138
reacting to data changes, 156–161
with RecyclerViews, 142, 145–147

Volley, 194–203
adding to projects, 194–195
parsing JSON, 197–200
RequestQueue, 195–197
Web service calls, 200–203

W

WHERE clause
in SELECT statements, 32
in UPDATE statement, 30, 31
write permissions, 109
write-ahead-log (WAL model, 41–42
writeToParcel() method, 172

Web services
accessing databases with Android SDK (Software Development
Kit), 179–187
with Retrofit, 188–194
with Volley, 194–203
battery consumption and, 203
data persistence, 204
AccountAuthenticator class, 204–208
manual synchronization with RxJava,
213–223
SyncAdapter class, 209–213
SyncAdapter framework, 204
REST and, 177–179
user experience and, 203–204