Zigurd MEDNIEKS
Series Editor

ANDROID™
DEEP
DIVE

Adam STROUD

ANDROID™ DATABASE BEST PRACTICES

FREE SAMPLE CHAPTER

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

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

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.

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.

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

5

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)`

- `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()`

```
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`

```
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 5.1 Example Database Table

first_name	last_name	id*
Bob	Smith	1
Ralph	Taylor	2
Sabrina	Anderson	3
Elizabeth	Hoffman	4
Abigail	Elder	5

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

```
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.

Table 5.2 `person` Table after Call to `update()`

first_name	last_name	id*
Robert	Smith	1
Ralph	Taylor	2
Sabrina	Anderson	3
Elizabeth	Hoffman	4
Abigail	Elder	5

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
- `ContcentValues 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

```
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 5.3 `person` Table after `replace()` Call

`first_name`	`last_name`	`id*`
Bob		1
Ralph	Taylor	2
Sabrina	Anderson	3
Elizabeth	Hoffman	4
Abigail	Elder	5

Table 5.4 Row Deleted from the Table

first_name	last_name	id*
Ralph	Taylor	2
Sabrina	Anderson	3
Elizabeth	Hoffman	4
Abigail	Elder	5

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 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

```
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

```
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

```
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

```
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 `ContentProviders`.

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

```
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);
```

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

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

    //... do something with data
}
```

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()

```
@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");
```

```
            break;
    }

    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()`

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

    index = data.getColumnIndexOrThrow("first_name");
    String firstName = data.getString(index);
```

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

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

    //... do something with data
}
```

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()`.

Both activities and fragments can get their `LoaderManager` object by calling `getLoaderManager()`. They can then start the load process by calling `LoaderManager.initLoader()`. `LoaderManager.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

Index

Symbols

A

D

E

F

G

M

N

O

W