

# 55. An Overview of Android SQLite Databases

Mobile applications that do not need to store at least some amount of persistent data are few and far between. The use of databases is an essential aspect of most applications, ranging from applications that are almost entirely data driven, to those that simply need to store small amounts of data such as the prevailing score of a game.

The importance of persistent data storage becomes even more evident when taking into consideration the somewhat transient lifecycle of the typical Android application. With the ever-present risk that the Android runtime system will terminate an application component to free up resources, a comprehensive data storage strategy to avoid data loss is a key factor in the design and implementation of any application development strategy.

This chapter will provide an overview of the SQLite database management system bundled with the Android operating system, together with an outline of the Android SDK classes that are provided to facilitate persistent SQLite based database storage from within an Android application. Before delving into the specifics of SQLite in the context of Android development, however, a brief overview of databases and SQL will be covered.

## 55.1 Understanding Database Tables

Database *Tables* provide the most basic level of data structure in a database. Each database can contain multiple tables and each table is designed to hold information of a specific type. For example, a database may contain a *customer* table that contains the name, address and telephone number for each of the customers of a particular business. The same database may also include a *products* table used to store the product descriptions with associated product codes for the items sold by the business.

Each table in a database is assigned a name that must be unique within that particular database. A table name, once assigned to a table in one database, may not be used for another table except within the context of another database.

## 55.2 Introducing Database Schema

*Database Schemas* define the characteristics of the data stored in a database table. For example, the table schema for a customer database table might define that the customer name is a string of no more than 20 characters in length, and that the customer phone number is a numerical data field of a certain format.

Schemas are also used to define the structure of entire databases and the relationship between the various tables contained in each database.

## 55.3 Columns and Data Types

It is helpful at this stage to begin to view a database table as being similar to a spreadsheet where data is stored in rows and columns.

Each column represents a data field in the corresponding table. For example, the name, address and telephone data fields of a table are all *columns*.

Each column, in turn, is defined to contain a certain type of data. A column designed to store numbers would, therefore, be defined as containing numerical data.

## 55.4 Database Rows

Each new record that is saved to a table is stored in a row. Each row, in turn, consists of the columns of data associated with the saved record.

Once again, consider the spreadsheet analogy described earlier in this chapter. Each entry in a customer table is equivalent to a row in a spreadsheet and each column contains the data for each customer (name, address, telephone etc). When a new customer is added to the table, a new row is created and the data for that customer stored in the corresponding columns of the new row.

*Rows* are also sometimes referred to as *records* or *entries* and these terms can generally be used interchangeably.

## 55.5 Introducing Primary Keys

Each database table should contain one or more columns that can be used to identify each row in the table uniquely. This is known in database terminology as the *Primary Key*. For example, a table may use a bank account number column as the primary key. Alternatively, a customer table may use the customer's social security number as the primary key.

Primary keys allow the database management system to identify a specific row in a table uniquely. Without a primary key it would not be possible to retrieve or delete a specific row in a table because there can be no certainty that the correct row has been selected. For example, suppose a table existed where the customer's last name had been defined as the primary key. Imagine then the problem that might arise if more than one customer named "Smith" were recorded in the database. Without some guaranteed way to identify a specific row uniquely, it would be impossible to ensure the correct data was being accessed at any given time.

Primary keys can comprise a single column or multiple columns in a table. To qualify as a single column primary key, no two rows can contain matching primary key values. When using multiple columns to construct a primary key, individual column values do not need to be unique, but all the columns' values combined together must be unique.

## 55.6 What is SQLite?

SQLite is an embedded, relational database management system (RDBMS). Most relational databases (Oracle, SQL Server and MySQL being prime examples) are standalone server processes that run independently, and in cooperation with, applications that require database access. SQLite is referred to as *embedded* because it is provided in the form of a library that is linked into applications. As such, there is no standalone database server running in the background. All database operations are handled internally within the application through calls to functions contained in the SQLite library.

The developers of SQLite have placed the technology into the public domain with the result that it is now a widely deployed database solution.

SQLite is written in the C programming language and as such, the Android SDK provides a Java based "wrapper" around the underlying database interface. This essentially consists of a set of classes that may be utilized within the Java or Kotlin code of an application to create and manage SQLite based databases.

For additional information about SQLite refer to <http://www.sqlite.org>.

## 55.7 Structured Query Language (SQL)

Data is accessed in SQLite databases using a high-level language known as

Structured Query Language. This is usually abbreviated to SQL and pronounced *sequel*. SQL is a standard language used by most relational database management systems. SQLite conforms mostly to the SQL-92 standard.

SQL is essentially a very simple and easy to use language designed specifically to enable the reading and writing of database data. Because SQL contains a small set of keywords, it can be learned quickly. In addition, SQL syntax is more or less identical between most DBMS implementations, so having learned SQL for one system, it is likely that your skills will transfer to other database management systems.

While some basic SQL statements will be used within this chapter, a detailed overview of SQL is beyond the scope of this book. There are, however, many other resources that provide a far better overview of SQL than we could ever hope to provide in a single chapter here.

## 55.8 Trying SQLite on an Android Virtual Device (AVD)

For readers unfamiliar with databases in general and SQLite in particular, diving right into creating an Android application that uses SQLite may seem a little intimidating. Fortunately, Android is shipped with SQLite pre-installed, including an interactive environment for issuing SQL commands from within an *adb shell* session connected to a running Android AVD emulator instance. This is both a useful way to learn about SQLite and SQL, and also an invaluable tool for identifying problems with databases created by applications running in an emulator.

To launch an interactive SQLite session, begin by running an AVD session. This can be achieved from within Android Studio by launching the Android Virtual Device Manager (*Tools -> Android -> AVD Manager*), selecting a previously configured AVD and clicking on the start button.

Once the AVD is up and running, open a Terminal or Command-Prompt window and connect to the emulator using the *adb* command-line tool as follows (note that the *-e* flag directs the tool to look for an emulator with which to connect, rather than a physical device):

```
adb -e shell
```

Once connected, the shell environment will provide a command prompt at

which commands may be entered. Begin by obtaining super user privileges using the *su* command:

```
Generic_x86:/ su  
root@android:/ #
```

If a message appears indicating that super user privileges are not allowed, it is likely that the AVD instance includes Google Play support. To resolve this create a new AVD and, on the “Choose a device definition” screen, select a device that does not have a marker in the “Play Store” column.

Data stored in SQLite databases are actually stored in database files on the file system of the Android device on which the application is running. By default, the file system path for these database files is as follows:

```
/data/data/<package name>/databases/<database filename>.db
```

For example, if an application with the package name *com.example.MyDBApp* creates a database named *mydatabase.db*, the path to the file on the device would read as follows:

```
/data/data/com.example.MyDBApp/databases/mydatabase.db
```

For the purposes of this exercise, therefore, change directory to */data/data* within the adb shell and create a sub-directory hierarchy suitable for some SQLite experimentation:

```
cd /data/data  
mkdir com.example.dbexample  
cd com.example.dbexample  
mkdir databases  
cd databases
```

With a suitable location created for the database file, launch the interactive SQLite tool as follows:

```
root@android:/data/data/databases # sqlite3 ./mydatabase.db  
sqlite3 ./mydatabase.db  
SQLite version 3.8.10.2 2015-05-20 18:17:19  
Enter ".help" for usage hints.  
sqlite>
```

At the *sqlite>* prompt, commands may be entered to perform tasks such as creating tables and inserting and retrieving data. For example, to create a new table in our database with fields to hold ID, name, address and phone number fields the following statement is required:

```
create table contacts (_id integer primary key autoincrement, name  
text, address text, phone text);
```

Note that each row in a table should have a *primary key* that is unique to that row. In the above example, we have designated the ID field as the primary key, declared it as being of type *integer* and asked SQLite to increment the number automatically each time a row is added. This is a common way to make sure that each row has a unique primary key. On most other platforms, the choice of name for the primary key is arbitrary. In the case of Android, however, it is essential that the key be named *\_id* in order for the database to be fully accessible using all of the Android database related classes. The remaining fields are each declared as being of type *text*.

To list the tables in the currently selected database, use the *.tables* statement:

```
sqlite> .tables
contacts
```

To insert records into the table:

```
sqlite> insert into contacts (name, address, phone) values ("Bill
Smith", "123 Main Street, California", "123-555-2323");
sqlite> insert into contacts (name, address, phone) values ("Mike
Parks", "10 Upping Street, Idaho", "444-444-1212");
```

To retrieve all rows from a table:

```
sqlite> select * from contacts;
1|Bill Smith|123 Main Street, California|123-555-2323
2|Mike Parks|10 Upping Street, Idaho|444-444-1212
```

To extract a row that meets specific criteria:

```
sqlite> select * from contacts where name="Mike Parks";
2|Mike Parks|10 Upping Street, Idaho|444-444-1212
```

To exit from the sqlite3 interactive environment:

```
sqlite> .exit
```

When running an Android application in the emulator environment, any database files will be created on the file system of the emulator using the previously discussed path convention. This has the advantage that you can connect with adb, navigate to the location of the database file, load it into the sqlite3 interactive tool and perform tasks on the data to identify possible problems occurring in the application code.

It is also important to note that, while it is possible to connect with an adb shell to a physical Android device, the shell is not granted sufficient privileges by default to create and manage SQLite databases. Debugging of database problems is, therefore, best performed using an AVD session.

## 55.9 Android SQLite Classes

SQLite is, as previously mentioned, written in the C programming language while Android applications are primarily developed using Java. To bridge this “language gap”, the Android SDK includes a set of classes that provide a Java layer on top of the SQLite database management system. The remainder of this chapter will provide a basic overview of each of the major classes within this category. More details on each class can be found in the online Android documentation.

### 55.9.1 Cursor

A class provided specifically to provide access to the results of a database query. For example, a SQL SELECT operation performed on a database will potentially return multiple matching rows from the database. A Cursor instance can be used to step through these results, which may then be accessed from within the application code using a variety of methods. Some key methods of this class are as follows:

- **close()** – Releases all resources used by the cursor and closes it.
- **getCount()** – Returns the number of rows contained within the result set.
- **moveToFirst()** – Moves to the first row within the result set.
- **moveToLast()** – Moves to the last row in the result set.
- **moveToNext()** – Moves to the next row in the result set.
- **move()** – Moves by a specified offset from the current position in the result set.
- **get<type>()** – Returns the value of the specified <type> contained at the specified column index of the row at the current cursor position (variations consist of *getString()*, *getInt()*, *getShort()*, *getFloat()* and *getDouble()*).

### 55.9.2 SQLiteDatabase

This class provides the primary interface between the application code and underlying SQLite databases including the ability to create, delete and perform SQL based operations on databases. Some key methods of this class

are as follows:

- **insert()** – Inserts a new row into a database table.
- **delete()** – Deletes rows from a database table.
- **query()** – Performs a specified database query and returns matching results via a Cursor object.
- **execSQL()** – Executes a single SQL statement that does not return result data.
- **rawQuery()** – Executes a SQL query statement and returns matching results in the form of a Cursor object.

### 55.9.3SQLiteOpenHelper

A helper class designed to make it easier to create and update databases. This class must be subclassed within the code of the application seeking database access and the following callback methods implemented within that subclass:

- **onCreate()** – Called when the database is created for the first time. This method is passed the SQLiteDatabase object as an argument for the newly created database. This is the ideal location to initialize the database in terms of creating a table and inserting any initial data rows.
- **onUpgrade()** – Called in the event that the application code contains a more recent database version number reference. This is typically used when an application is updated on the device and requires that the database schema also be updated to handle storage of additional data.

In addition to the above mandatory callback methods, the *onOpen()* method, called when the database is opened, may also be implemented within the subclass.

The constructor for the subclass must also be implemented to call the super class, passing through the application context, the name of the database and the database version.

Notable methods of the SQLiteOpenHelper class include:

- **getWritableDatabase()** – Opens or creates a database for reading and



writing. Returns a reference to the database in the form of a SQLiteDatabase object.

- **getReadableDatabase()** – Creates or opens a database for reading only. Returns a reference to the database in the form of a SQLiteDatabase object.
- **close()** – Closes the database.

#### 55.9.4 ContentValues

ContentValues is a convenience class that allows key/value pairs to be declared consisting of table column identifiers and the values to be stored in each column. This class is of particular use when inserting or updating entries in a database table.

### 55.10 Summary

SQLite is a lightweight, embedded relational database management system that is included as part of the Android framework and provides a mechanism for implementing organized persistent data storage for Android applications. In addition to the SQLite database, the Android framework also includes a range of Java classes that may be used to create and manage SQLite based databases and tables.

The goal of this chapter was to provide an overview of databases in general and SQLite in particular within the context of Android application development. The next chapters will work through the creation of an example application intended to put this theory into practice in the form of a step-by-step tutorial. Since the user interface for the example application will require a forms based layout, the first chapter, entitled [\*“An Android TableLayout and TableRow Tutorial”\*](#), will detour slightly from the core topic by introducing the basics of the TableLayout and TableRow views.