Chapter 22

Managing and Accessing Local Databases

SQLite is a very popular embedded database, as it combines a clean SQL interface with a very small memory footprint and decent speed. Moreover, it is public domain, so everyone can use it. Many firms (e.g., Adobe, Apple, Google, Sun, and Symbian) and open source projects (e.g., Mozilla, PHP, and Python) ship products with SQLite.

For Android, SQLite is "baked into" the Android runtime, so every Android application can create SQLite databases. Since SQLite uses a SQL interface, it is fairly straightforward to use for people with experience in other SQL-based databases. However, its native API is not JDBC, and JDBC might be too much overhead for a memory-limited device like a phone, anyway. Hence, Android programmers have a different API to learn. The good news is that it is not that difficult.

This chapter will cover the basics of SQLite use in the context of working on Android. It by no means is a thorough coverage of SQLite as a whole. If you want to learn more about SQLite and how to use it in environments other than Android, a fine book is *The Definitive Guide to SQLite* by Michael Owens (Apress, 2006).

The Database Example

Much of the sample code shown in this chapter comes from the `Database/Constants` application. This application presents a list of physical constants, with names and values culled from Android's `SensorManager`, as shown in Figure 22–1.

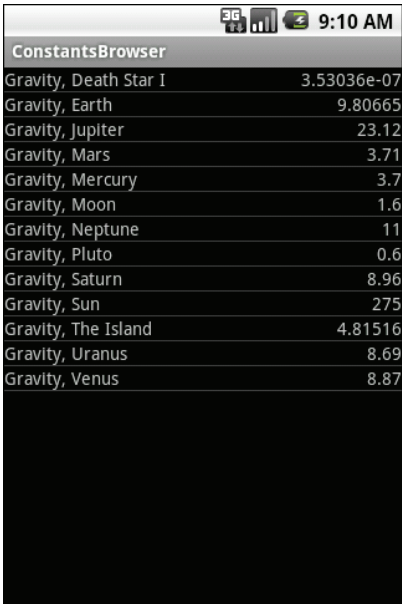

Figure 22–1. *The Constants sample application, as initially launched*

You can pop up a menu to add a new constant, which brings up a dialog to fill in the name and value of the constant, as shown in Figure 22–2.

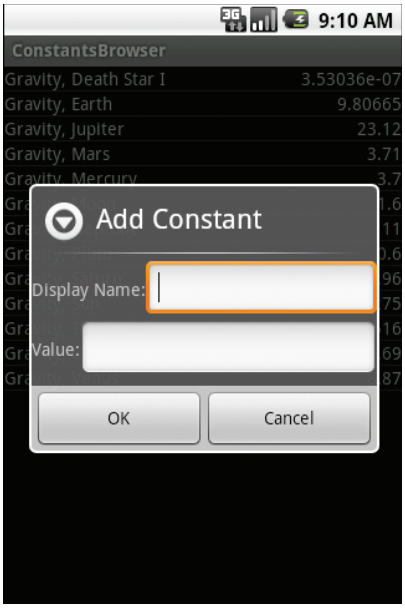

Figure 22–2. *The Constants sample application's Add Constant dialog*

The constant is then added to the list. A long-tap on an existing constant will bring up a context menu with a Delete option, which, after confirmation, will delete the constant.

And, of course, all of this is stored in a SQLite database.

A Quick SQLite Primer

SQLite, as the name suggests, uses a dialect of SQL for queries (`SELECT`), data manipulation (`INSERT`, et. al.), and data definition (`CREATE TABLE`, et. al.). SQLite has a few places where it deviates from the SQL-92 standard, as is common for most SQL databases. The good news is that SQLite is so space-efficient that the Android runtime can include all of SQLite, not some arbitrary subset to trim it down to size.

A big difference between SQLite and other SQL databases is the data typing. While you can specify the data types for columns in a `CREATE TABLE` statement, and SQLite will use those as a hint, that is as far as it goes. You can put whatever data you want in whatever column you want. Put a string in an `INTEGER` column? Sure, no problem! Vice versa? That works, too! SQLite refers to this as *manifest typing*, as described in the documentation:

> *In manifest typing, the datatype is a property of the value itself, not of the column in which the value is stored. SQLite thus allows the user to store any value of any datatype into any column regardless of the declared type of that column.*

In addition, a handful of standard SQL features are not supported in SQLite, notably `FOREIGN KEY` constraints, nested transactions, `RIGHT OUTER JOIN`, `FULL OUTER JOIN`, and some flavors of `ALTER TABLE`.

Beyond that, though, you get a full SQL system, complete with triggers, transactions, and the like. Stock SQL statements, like `SELECT`, work pretty much as you might expect.

NOTE: If you are used to working with a major database, like Oracle, you may look upon SQLite as being a "toy" database. Please bear in mind that Oracle and SQLite are meant to solve different problems, and that you will not be seeing a full copy of Oracle on a phone any time soon, in all likelihood.

Start at the Beginning

No databases are automatically supplied to you by Android. If you want to use SQLite, you will need to create your own database, and then populate it with your own tables, indexes, and data.

To create and open a database, your best option is to craft a subclass of `SQLiteOpenHelper`. This class wraps up the logic to create and upgrade a database, per your specifications, as needed by your application. Your subclass of `SQLiteOpenHelper` will need three methods:

- The constructor, chaining upward to the SQLiteOpenHelper constructor. This takes the Context (e.g., an Activity), the name of the database, an optional cursor factory (typically, just pass null), and an integer representing the version of the database schema you are using.
- onCreate(), which passes you a SQLiteDatabase object that you need to populate with tables and initial data, as appropriate.
- onUpgrade(), which passes you a SQLiteDatabase object and the old and new version numbers, so you can figure out how best to convert the database from the old schema to the new one. The simplest, albeit least friendly, approach is to drop the old tables and create new ones.

For example, here is a DatabaseHelper class from Database/Constants that, in onCreate(), creates a table and adds a number of rows, and in onUpgrade() cheats by dropping the existing table and executing onCreate():

```
package com.commonsware.android.constants;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteDatabase;
import android.hardware.SensorManager;

public class DatabaseHelper extends SQLiteOpenHelper {
  private static final String DATABASE_NAME="db";
  public static final String TITLE="title";
  public static final String VALUE="value";

  public DatabaseHelper(Context context) {
    super(context, DATABASE_NAME, null, 1);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    db.execSQL("CREATE TABLE constants (_id INTEGER PRIMARY KEY AUTOINCREMENT,↩
 title TEXT, value REAL);");

    ContentValues cv=new ContentValues();

    cv.put(TITLE, "Gravity, Death Star I");
    cv.put(VALUE, SensorManager.GRAVITY_DEATH_STAR_I);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Gravity, Earth");
    cv.put(VALUE, SensorManager.GRAVITY_EARTH);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Gravity, Jupiter");
    cv.put(VALUE, SensorManager.GRAVITY_JUPITER);
```

```
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Gravity, Mars");
    cv.put(VALUE, SensorManager.GRAVITY_MARS);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Gravity, Mercury");
    cv.put(VALUE, SensorManager.GRAVITY_MERCURY);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Gravity, Moon");
    cv.put(VALUE, SensorManager.GRAVITY_MOON);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Gravity, Neptune");
    cv.put(VALUE, SensorManager.GRAVITY_NEPTUNE);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Gravity, Pluto");
    cv.put(VALUE, SensorManager.GRAVITY_PLUTO);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Gravity, Saturn");
    cv.put(VALUE, SensorManager.GRAVITY_SATURN);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Gravity, Sun");
    cv.put(VALUE, SensorManager.GRAVITY_SUN);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Gravity, The Island");
    cv.put(VALUE, SensorManager.GRAVITY_THE_ISLAND);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Gravity, Uranus");
    cv.put(VALUE, SensorManager.GRAVITY_URANUS);
    db.insert("constants", TITLE, cv);

    cv.put(TITLE, "Gravity, Venus");
    cv.put(VALUE, SensorManager.GRAVITY_VENUS);
    db.insert("constants", TITLE, cv);
  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    android.util.Log.w("Constants", "Upgrading database, which will destroy all old
data");
    db.execSQL("DROP TABLE IF EXISTS constants");
    onCreate(db);
  }
}
```

To use your SQLiteOpenHelper subclass, create an instance and ask it to getReadableDatabase() or getWriteableDatabase(), depending on whether or not you will be changing its contents. For example, our ConstantsBrowser activity opens the database in onCreate():

```
db=(new DatabaseHelper(this)).getWritableDatabase();
```

This will return a SQLiteDatabase instance, which you can then use to query the database or modify its data.

When you are finished with the database (e.g., your activity is being closed), simply call close() on the SQLiteDatabase to release your connection.

Setting the Table

For creating your tables and indexes, you will need to call execSQL() on your SQLiteDatabase, providing the Data Definition Language (DDL) statement you wish to apply against the database. Barring a database error, this method returns nothing.

So, for example, you can call execSQL() to create the constants table, as shown in the DatabaseHelper onCreate() method:

```
db.execSQL("CREATE TABLE constants (_id INTEGER PRIMARY KEY AUTOINCREMENT, title
TEXT,➥
 value REAL);");
```

This will create a table, named constants, with a primary key column named _id that is an autoincremented integer (i.e., SQLite will assign the value for you when you insert rows), plus two data columns: title (text) and value (a float, or *real* in SQLite terms). SQLite will automatically create an index for you on your primary key column. You could add other indexes here via some CREATE INDEX statements.

Most likely, you will create tables and indexes when you first create the database, or possibly when the database needs upgrading to accommodate a new release of your application. If you do not change your table schemas, you might never drop your tables or indexes, but if you do, just use execSQL() to invoke DROP INDEX and DROP TABLE statements as needed.

Makin' Data

Given that you have a database and one or more tables, you probably want to put some data in them. You have two major approaches for doing this.

- Use execSQL(), just as you did for creating the tables. The execSQL() method works for any SQL that does not return results, so it can handle INSERT, UPDATE, DELETE, and so on just fine.

- Use the `insert()`, `update()`, and `delete()` methods on the `SQLiteDatabase` object. These are "builder" sorts of methods, in that they break down the SQL statements into discrete chunks, then take those chunks as parameters.

For example, here we `insert()` a new row into our `constants` table:

```
private void processAdd(DialogWrapper wrapper) {
  ContentValues values=new ContentValues(2);

  values.put("title", wrapper.getTitle());
  values.put("value", wrapper.getValue());

  db.insert("constants", "title", values);
  constantsCursor.requery();
}
```

These methods make use of `ContentValues` objects, which implement a `Map`-esque interface, albeit one that has additional methods for working with SQLite types. For example, in addition to `get()` to retrieve a value by its key, you have `getAsInteger()`, `getAsString()`, and so forth.

The `insert()` method takes the name of the table, the name of one column as the "null column hack," and a `ContentValues` with the initial values you want put into this row. The null column hack is for the case where the `ContentValues` instance is empty. The column named as the null column hack will be explicitly assigned the value `NULL` in the SQL `INSERT` statement generated by `insert()`.

The `update()` method takes the name of the table, a `ContentValues` representing the columns and replacement values to use, an optional `WHERE` clause, and an optional list of parameters to fill into the `WHERE` clause, to replace any embedded question marks (`?`). Since `update()` replaces only columns with fixed values, versus ones computed based on other information, you may need to use `execSQL()` to accomplish some ends. The `WHERE` clause and parameter list work akin to the positional SQL parameters you may be used to from other SQL APIs.

The `delete()` method works similar to `update()`, taking the name of the table, the optional `WHERE` clause, and the corresponding parameters to fill into the `WHERE` clause. For example, here we delete a row from our `constants` table, given its `_ID`:

```
private void processDelete(long rowId) {
  String[] args={String.valueOf(rowId)};

  db.delete("constants", "_ID=?", args);
  constantsCursor.requery();
}
```

What Goes Around Comes Around

As with INSERT, UPDATE, and DELETE, you have two main options for retrieving data from a SQLite database using SELECT:

- Use rawQuery() to invoke a SELECT statement directly.
- Use query() to build up a query from its component parts.

Confounding matters further is the SQLiteQueryBuilder class and the issue of cursors and cursor factories. Let's take this one piece at a time.

Raw Queries

The simplest solution, at least in terms of the API, is rawQuery(). Just call it with your SQL SELECT statement. The SELECT statement can include positional parameters; the array of these forms your second parameter to rawQuery(). So, we wind up with this:

```
constantsCursor=db.rawQuery("SELECT _ID, title, value "+
                            "FROM constants ORDER BY title",
                            null);
```

The return value is a Cursor, which contains methods for iterating over results (discussed in the "Using Cursors" section a little later in the chapter).

If your queries are pretty much baked into your application, this is a very straightforward way to use them. However, it gets complicated if parts of the query are dynamic, beyond what positional parameters can really handle. For example, if the set of columns you need to retrieve is not known at compile time, puttering around concatenating column names into a comma-delimited list can be annoying, which is where query() comes in.

Regular Queries

The query() method takes the discrete pieces of a SELECT statement and builds the query from them. The pieces, in the order they appear as parameters to query(), are as follows:

- The name of the table to query against
- The list of columns to retrieve
- The WHERE clause, optionally including positional parameters
- The list of values to substitute in for those positional parameters
- The GROUP BY clause, if any
- The ORDER BY clause, if any
- The HAVING clause, if any

These can be null when they are not needed (except the table name, of course):

```
String[] columns={"ID", "inventory"};
String[] parms={"snicklefritz"};
Cursor result=db.query("widgets", columns, "name=?",
                       parms, null, null, null);
```

Building with Builders

Yet another option is to use SQLiteQueryBuilder, which offers much richer query-building options, particularly for nasty queries involving things like the union of multiple subquery results.

The SQLiteQueryBuilder interface dovetails nicely with the ContentProvider interface for executing queries. Hence, a common pattern for your content provider's query() implementation is to create a SQLiteQueryBuilder, fill in some defaults, and then allow it to build up (and optionally execute) the full query combining the defaults with what is provided to the content provider on the query request.

For example, here is a snippet of code from a content provider using SQLiteQueryBuilder:

```
@Override
public Cursor query(Uri url, String[] projection, String selection,
                    String[] selectionArgs, String sort) {
  SQLiteQueryBuilder qb=new SQLiteQueryBuilder();

  qb.setTables(getTableName());

  if (isCollectionUri(url)) {
    qb.setProjectionMap(getDefaultProjection());
  }
  else {
    qb.appendWhere(getIdColumnName()+"="+url.getPathSegments().get(1));
  }

  String orderBy;

  if (TextUtils.isEmpty(sort)) {
    orderBy=getDefaultSortOrder();
  } else {
    orderBy=sort;
  }

  Cursor c=qb.query(db, projection, selection, selectionArgs,
                    null, null, orderBy);
  c.setNotificationUri(getContext().getContentResolver(), url);
  return c;
}
```

Content providers are explained in greater detail in Chapters 26 and 27, so some of this you will have to take on faith until then. Here, you see the following:

- A `SQLiteQueryBuilder` is constructed.
- It is told the table to use for the query (`setTables(getTableName())`).
- It is told the default set of columns to return (`setProjectionMap()`), or it is given a piece of a `WHERE` clause to identify a particular row in the table by an identifier extracted from the `Uri` supplied to the `query()` call (`appendWhere()`).
- Finally, it is told to execute the query, blending the preset values with those supplied on the call to `query()` (`qb.query(db, projection, selection, selectionArgs, null, null, orderBy)`).

Instead of having the `SQLiteQueryBuilder` execute the query directly, we could have called `buildQuery()` to have it generate and return the SQL `SELECT` statement we needed, which we could then execute ourselves.

Using Cursors

No matter how you execute the query, you get a `Cursor` back. This is the Android/SQLite edition of the database cursor, a concept used in many database systems. With the cursor, you can do the following:

- Find out how many rows are in the result set via `getCount()`.
- Iterate over the rows via `moveToFirst()`, `moveToNext()`, and `isAfterLast()`.
- Find out the names of the columns via `getColumnNames()`, convert those into column numbers via `getColumnIndex()`, and get values for the current row for a given column via methods like `getString()`, `getInt()`, and so on.
- Reexecute the query that created the cursor via `requery()`.
- Release the cursor's resources via `close()`.

For example, here we iterate over a `widgets` table entries:

```
Cursor result=
  db.rawQuery("SELECT ID, name, inventory FROM widgets");

result.moveToFirst();

while (!result.isAfterLast()) {
 int id=result.getInt(0);
 String name=result.getString(1);
 int inventory=result.getInt(2);

 // do something useful with these
```

```
 result.moveToNext();
}

result.close();
```

You can also wrap a `Cursor` in a `SimpleCursorAdapter` or other implementation, and then hand the resulting adapter to a `ListView` or other selection widget. For example, after retrieving the sorted list of constants, we pop those into the `ListView` for the `ConstantsBrowser` activity in just a few lines of code:

```
ListAdapter adapter=new SimpleCursorAdapter(this,
                    R.layout.row, constantsCursor,
                    new String[] {"title", "value"},
                    new int[] {R.id.title, R.id.value});

setListAdapter(adapter);
```

TIP: There may be circumstances in which you want to use your own `Cursor` subclass, rather than the stock implementation provided by Android. In those cases, you can use `queryWithFactory()` and `rawQueryWithFactory()`, which take a `SQLiteDatabase.CursorFactory` instance as a parameter. The factory is responsible for creating new cursors via its `newCursor()` implementation.

Data, Data, Everywhere

If you are used to developing for other databases, you are also probably used to having tools to inspect and manipulate the contents of the database, beyond merely the database's API. With Android's emulator, you have two main options for this.

First, the emulator is supposed to bundle in the `sqlite3` console program and make it available from the `adb shell` command. Once you are in the emulator's shell, just execute `sqlite3`, providing it the path to your database file. Your database file can be found at the following location:

```
/data/data/your.app.package/databases/your-db-name
```

Here, `your.app.package` is the Java package for your application (e.g., `com.commonsware.android`), and `your-db-name` is the name of your database, as supplied to `createDatabase()`.

The `sqlite3` program works, and if you are used to poking around your tables using a console interface, you are welcome to use it. If you prefer something a little friendlier, you can always copy the SQLite database off the device onto your development machine, and then use a SQLite-aware client program to putter around. Note, though, that you are working off a copy of the database; if you want your changes to go back to the device, you will need to transfer the database back over.

To get the database off the device, you can use the `adb pull` command (or the equivalent in your IDE, or File Manager in the Dalvik Debug Monitor Service, discussed

in Chapter 35), which takes the path to the on-device database and the local destination as parameters. To store a modified database on the device, use `adb push`, which takes the local path to the database and the on-device destination as parameters.

One of the most accessible SQLite clients is the SQLite Manager extension for Firefox, shown in Figure 22–2, as it works across all platforms.

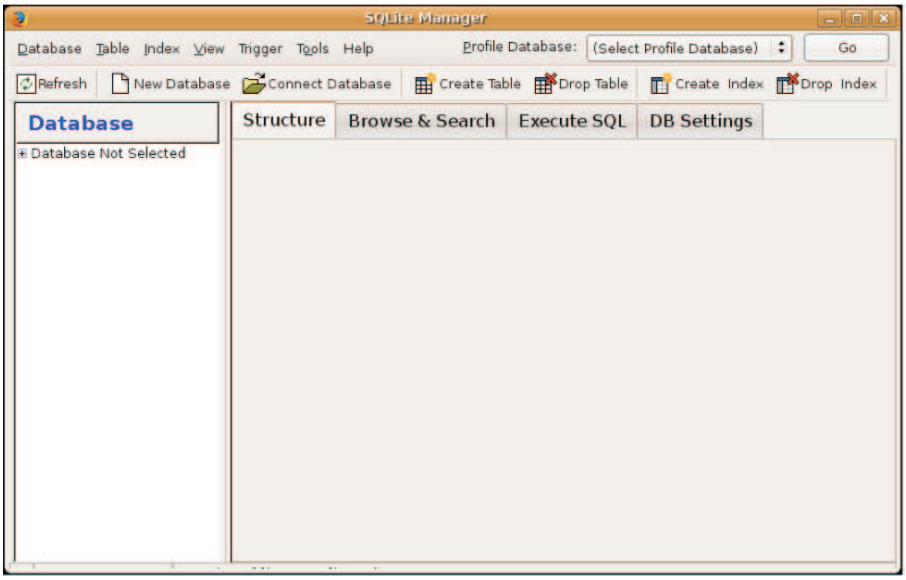

Figure 22–3. *SQLite Manager Firefox extension*

You can find other client tools on the SQLite web site.