CHAPTER 10

# Databases and Content Providers

In the abstract, working with SQLite databases and Android-style content providers is fairly straight-forward. Each supports a CRUD-style interface (`query()`, `insert()`, `update()`, `delete()`) using `Cursor` objects for query results. While implementing a `ContentProvider` is no picnic for non-SQLite data stores, everything else is fairly rote.

In reality, though, databases and content providers cause more than their fair share of hassles. Mostly, this comes from everything *outside* of simple CRUD operations, such as:

- How do we get a database into our application?
- How do we get data into our application on initial install? On an update?
- Where is the documentation for the built-in Android content providers?
- How do we deal with joins between data stores, such as merging contacts with our own database data?

In this chapter, we explore these issues, to show how you can better work with databases and content providers in the real world.

## Distributed Data

Some databases used by Android applications naturally start empty. For example, a "password safe" probably has no passwords when initially launched by the user, and an expense-tracking application probably does not have any expenses recorded at the outset.

However, sometimes, there are databases that need to ship with an application that must be pre-populated with data. For example, you might be implementing an online catalog, with a database of items for sale installed with the application and updated as needed via calls to some Web service. The same structure would hold true for any sort of reference, from chemicals to word translations to historical sports records.

Unfortunately, there is no way to ship a database with data in it via the Android APK packaging mechanism. An APK is an executable blob, from the standpoint of Android and Dalvik. More importantly, it is stored read-only in a ZIP file, which makes updates to that data doubly impossible.

The next-best option is to ship your data with the application by some other means and load it into a newly-created database when the application is first run. This does involve two copies of the data: the original in your application and the working copy in the database. That may seem wasteful in terms of space. However, courtesy of ZIP compression, the original copy may not take up all that much space. Also, you can turn this into a feature, offering some sort of "reset" mechanism to reload the working database from the original if needed.

The challenge then becomes how to package the database contents into the APK and load it into the working database. Ideally, this involves as little work as possible from the developer, can fit into the existing build system, and can take advantage of existing database manipulation tools (versus, say, hand-writing hundreds of `SQL INSERT` statements).

Note that another possibility exists: package the binary SQLite database file in the APK (e.g., in `res/raw/`) and copy it into position using binary streams.

This assumes the SQLite database file your development environment would create is the same as what is expected by the SQLite engine baked into Android. This can work, but is likely to be more prone to versioning issues – for example, if your development environment is upgraded to a newer SQLite that has a slightly different file format.

## SQLite: On-Device, On-Desktop

This becomes much simpler when you realize that Android uses SQLite for the database, and SQLite works on just about every platform you might need. It is trivial to work with SQLite databases on your development workstation, even easier than working with databases inside an Android emulator or device.

The plan, therefore, is to allow developers to create the database to be "shipped" as a SQLite database, then build tools that package the SQLite contents into the Android APK and turn it back into a database when the application needs it.

This allows developers to use whatever tools they want to manipulate the SQLite database, ranging from typical database management UIs to specialized conversion scripts to whatever.

To make this plan work, though, we need two bits of code:

1. We need something that extracts the data out of the SQLite database the developer has prepared and puts it someplace inside the Android APK
2. We need something that ties in with `SQLiteOpenHelper` that takes the APK-packaged data and turns it into an on-device database when the database is first accessed.

## Exporting a Database

Fortunately, the `sqlite3` command-line executable that comes standard with SQLite offers a `.dump` command to dump the contents of a table as a

series of SQL statements: one to create the table, plus the necessary `SQL INSERT` statements to populate it. All we need to do is tie this into the build system, so the act of compiling the APK also deals with the database.

You can find some sample code that handles this in the `Database/Packager` sample application. Specifically:

- There is a SQLite database containing data in the `db/` project directory – in this case, it is the database from the `ContentProvider/Constants` project from *The Busy Coder's Guide to Android Development*
- There is a `package_db.rb` Ruby script that wraps around the `.dump` command to export the data
- There is a change to the `build.xml` Ant script to use this Ruby script

## *The Ruby Script*

You may or may not be a fan of Ruby. While this sample code shows this utility as a Ruby script, rest assured that SQLite has interfaces to most programming languages (though its Java support is not the strongest), so you can create your own edition of this script in whatever language suits you.

The script is fairly short:

```
require 'rubygems'
require 'sqlite3'

Dir['db/*'].each do |path|
  db=SQLite3::Database.new(path)

  begin
    db.execute("SELECT name FROM sqlite_master WHERE type='table'") do |row|
      if ARGV.include?(row[0])
        puts `sqlite3 #{path} ".dump #{row[0]}"`
      end
    end
  ensure
    db.close
  end
end
```

It iterates over every file in the `db/` directory and opens each as a SQLite database. It then queries the database for the list of tables (`SELECT name FROM sqlite_master WHERE type = 'table'`). Any table matching a table name passed in on the command line is assumed to be one needing to be exported, so it prints to `stdout` the results of the `sqlite3 .dump` command, run on that database and table. We use `sqlite3` because there does not appear to be an API call that implements the `.dump` functionality.

To run this script, you need SQLite3 installed, with `sqlite3` in your PATH, and you need the Ruby interpreter. You also need to run it from the project directory, with a `db/` directory containing one or more database files.

Running the Ruby script will dump the specified tables as a set of SQL statements:

```
BEGIN TRANSACTION;
CREATE TABLE constants (_id INTEGER PRIMARY KEY AUTOINCREMENT, title TEXT, value
REAL);
INSERT INTO "constants" VALUES(1,'Gravity, Death Star I',3.53036142541896e-07);
INSERT INTO "constants" VALUES(2,'Gravity, Earth',9.80665016174316);
INSERT INTO "constants" VALUES(3,'Gravity, Jupiter',23.1200008392334);
INSERT INTO "constants" VALUES(4,'Gravity, Mars',3.71000003814697);
INSERT INTO "constants" VALUES(5,'Gravity, Mercury',3.70000004768372);
INSERT INTO "constants" VALUES(6,'Gravity, Moon',1.60000002384186);
INSERT INTO "constants" VALUES(7,'Gravity, Neptune',11.0);
INSERT INTO "constants" VALUES(8,'Gravity, Pluto',0.600000023841858);
INSERT INTO "constants" VALUES(9,'Gravity, Saturn',8.96000003814697);
INSERT INTO "constants" VALUES(10,'Gravity, Sun',275.0);
INSERT INTO "constants" VALUES(11,'Gravity, The Island',4.81516218185425);
INSERT INTO "constants" VALUES(12,'Gravity, Uranus',8.6899995803833);
INSERT INTO "constants" VALUES(13,'Gravity, Venus',8.86999988555908);
COMMIT;
```

In this case, the constants table is empty, so there are no `SQL INSERT` statements. However, you could easily add some rows to the `constants` table – perhaps constants not available in Android itself – and ship those along with the table schema.

## Loading the Exported Database

The other end of his process is to take the raw SQL stores in `res/raw/packaged_db.txt` and "inflate" it at runtime into a database. Since

`SQLiteOpenHelper` is designed to handle such operations, it seems to make sense to implement this logic as a subclass. You can find such a class – `DatabaseInstaller` – in the `Database/Packager` sample project:

```
import android.content.Context;
import android.database.SQLException;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteQueryBuilder;
import java.io.*;


abstract class DatabaseInstaller extends SQLiteOpenHelper {
  abstract void handleInstallError(Throwable t);

  private Context ctxt=null;

  public DatabaseInstaller(Context context, String name,
                           SQLiteDatabase.CursorFactory factory,
                           int version)  {
    super(context, name, factory, version);

    this.ctxt=context;
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    try {
      InputStream stream=ctxt
                         .getResources()
                         .openRawResource(R.raw.packaged_db);
      InputStreamReader is=new InputStreamReader(stream);
      BufferedReader in=new BufferedReader(is);
      String str;

      while ((str = in.readLine()) != null) {
        if (!str.equals("BEGIN TRANSACTION;") && !str.equals("COMMIT;")) {
          db.execSQL(str);
        }
      }

      in.close();
    }
    catch (IOException e) {
      handleInstallError(e);
    }
  }
}
```

This class is abstract, expecting subclasses to implement both the `onUpgrade()` path from `SQLiteOpenHelper` and a `handleInstallError()` callback in case something fails during `onCreate()`.

Most of the smarts are found in `DatabaseInstaller`'s `onCreate()` implementation. Since `SQLiteDatabase` has no means to execute SQL statements contained in an `InputStream`, we are stuck opening the `R.raw.packaged_db` resource and reading the statements out ourselves, one at a time.

However, the exported SQL will likely contain `BEGIN TRANSACTION;` and `COMMIT;` statements, since `sqlite3` expects that `sqlite3` itself would be used to re-executed the dumped SQL script. Since transactions are handled via API calls with `SQLiteDatabase`, we cannot execute `BEGIN TRANSACTION;` and `COMMIT;` statements via `execSQL()` without getting a "nested transaction" error. So, we skip those two statements and execute everything else, one line at a time.

The net result: `onCreate()` takes our raw SQL and turns it into a table in our on-device database.

Of course, to really use this, you will need to create a `DatabaseInstaller` subclass, such as `ConstantsInstaller`:

```
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.util.Log;


class ConstantsInstaller  extends DatabaseInstaller {
  public ConstantsInstaller(Context context, String name,
                            SQLiteDatabase.CursorFactory factory,
                            int version)  {
    super(context, name, factory, version);
  }

  void handleInstallError(Throwable t) {
    Log.e("Constants", "Exception installing database", t);
  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion,
```

```
                        int newVersion) {
    db.execSQL("DROP TABLE IF EXISTS constants");
    onCreate(db);
  }
}
```

The rest of this project is largely identical to the `ContentProvider/Constants` sample from *The Busy Coder's Guide to Android Development*.

One possible enhancement to `DatabaseInstaller` is to create our own transaction around the loop of `execSQL()` calls. This would improve performance dramatically, as otherwise, each `execSQL()` call is its own transaction. The proof of this is left to the reader as an exercise.

# Examining Your Relationships

Android has a built-in contact manager, integrated with the phone dialer. You can work with the contacts via the `Contacts` content provider.

However, compared to content providers found in, say, simplified book examples, the `Contacts` content provider is rather intimidating. After all, there are 16 classes and 9 interfaces all involved in accessing this content provider. This section will attempt to illustrate some of the patterns for making use of `Contacts`.

## Contact Permissions

Since contacts are privileged data, you need certain permissions to work with them. Specifically, you need the `READ_CONTACTS` permission to query and examine the `Contacts` content and `WRITE_CONTACTS` to add, modify, or remove contacts from the system.

For example, here is the manifest for the `Database/Contacts` sample application:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
```

```
      package="com.commonsware.android.database"
      android:versionCode="1"
      android:versionName="1.0">
  <uses-permission android:name="android.permission.READ_CONTACTS" />
    <application android:label="@string/app_name">
        <activity android:name=".ContactsDemo"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

## Pre-Joined Data

While the database underlying the `Contacts` content provider is private, one can imagine that it has several tables: one for people, one for their phone numbers, one for their email addresses, etc. These are tied together by typical database relations, most likely 1:N, so the phone number and email address tables would have a foreign key pointing back to the table containing information about people.

To simplify accessing all of this through the content provider interface, Android pre-joins queries against some of the tables. For example, one can query for phone numbers and get the contact name and other data along with the number, without having to somehow do a join operation yourself.

## The Sample Activity

The `ContactsDemo` activity is simply a `ListActivity`, though it sports a `Spinner` to go along with the obligatory `ListView`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
  <Spinner android:id="@+id/spinner"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
```

```
    android:drawSelectorOnTop="true"
  />
  <ListView
    android:id="@android:id/list"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:drawSelectorOnTop="false"
  />
</LinearLayout>
```

The activity itself sets up a listener on the `Spinner` and toggles the list of information shown in the `ListView` when the `Spinner` value changes:

```
package com.commonsware.android.database;

import android.app.ListActivity;
import android.database.Cursor;
import android.os.Bundle;
import android.provider.Contacts;
import android.view.View;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.ListAdapter;
import android.widget.SimpleCursorAdapter;
import android.widget.Spinner;

public class ContactsDemo extends ListActivity
  implements AdapterView.OnItemSelectedListener {
  private static String[] options={"Contact Names",
                                   "Contact Names & Numbers",
                                   "Contact Names & Email Addresses"};
  private ListAdapter[] listAdapters=new ListAdapter[3];

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    initListAdapters();

    Spinner spin=(Spinner)findViewById(R.id.spinner);
    spin.setOnItemSelectedListener(this);

    ArrayAdapter<String> aa=new ArrayAdapter<String>(this,
                            android.R.layout.simple_spinner_item,
                            options);

    aa.setDropDownViewResource(
            android.R.layout.simple_spinner_dropdown_item);
    spin.setAdapter(aa);
  }
```

```
public void onItemSelected(AdapterView<?> parent,
                           View v, int position, long id) {
  setListAdapter(listAdapters[position]);
}

public void onNothingSelected(AdapterView<?> parent) {
  // ignore
}

private void initListAdapters() {
  listAdapters[0]=buildNameAdapter();
  listAdapters[1]=buildPhonesAdapter();
  listAdapters[2]=buildEmailAdapter();
}

private ListAdapter buildNameAdapter() {
  String[] PROJECTION=new String[] {  Contacts.People._ID,
                                      Contacts.PeopleColumns.NAME
                                    };
  Cursor c=managedQuery(Contacts.People.CONTENT_URI,
                        PROJECTION, null, null,
                        Contacts.People.DEFAULT_SORT_ORDER);

  return(new SimpleCursorAdapter(  this,
                                   android.R.layout.simple_list_item_1,
                                   c,
                                   new String[] {
                                     Contacts.PeopleColumns.NAME
                                   },
                                   new int[] {
                                     android.R.id.text1
                                   }));
}

private ListAdapter buildPhonesAdapter() {
  String[] PROJECTION=new String[] {  Contacts.Phones._ID,
                                      Contacts.Phones.NAME,
                                      Contacts.Phones.NUMBER
                                    };
  Cursor c=managedQuery(Contacts.Phones.CONTENT_URI,
                        PROJECTION, null, null,
                        Contacts.Phones.DEFAULT_SORT_ORDER);

  return(new SimpleCursorAdapter(  this,
                                   android.R.layout.simple_list_item_2,
                                   c,
                                   new String[] {
                                     Contacts.Phones.NAME,
                                     Contacts.Phones.NUMBER
                                   },
                                   new int[] {
                                     android.R.id.text1,
                                     android.R.id.text2
                                   }));
```

```
  }

  private ListAdapter buildEmailAdapter() {
    String[] PROJECTION=new String[] {  Contacts.ContactMethods._ID,
                                        Contacts.ContactMethods.DATA,
                                        Contacts.PeopleColumns.NAME
                                      };
    Cursor c=managedQuery(Contacts.ContactMethods.CONTENT_EMAIL_URI,
                          PROJECTION, null, null,
                          Contacts.ContactMethods.DEFAULT_SORT_ORDER);

    return(new SimpleCursorAdapter(  this,
                                     android.R.layout.simple_list_item_2,
                                     c,
                                     new String[] {
                                       Contacts.PeopleColumns.NAME,
                                       Contacts.ContactMethods.DATA
                                     },
                                     new int[] {
                                       android.R.id.text1,
                                       android.R.id.text2
                                     }));
  }
}
```

When the activity is first opened, it sets up three `Adapter` objects, one for each of three perspectives on the contacts data. The `Spinner` simply resets the list to use the `Adapter` associated with the `Spinner` value selected.

## Accessing People

The first `Adapter` shows the names of all of the contacts. Since all the information we seek is in the contact itself, we can use the `CONTENT_URI` provider, retrieve all of the contacts in the default sort order, and pour them into a `SimpleCursorAdapter` set up to show each person on its own row:

```
private ListAdapter buildNameAdapter() {
  String[] PROJECTION=new String[] {  Contacts.People._ID,
                                      Contacts.PeopleColumns.NAME
                                    };
  Cursor c=managedQuery(Contacts.People.CONTENT_URI,
                        PROJECTION, null, null,
                        Contacts.People.DEFAULT_SORT_ORDER);

  return(new SimpleCursorAdapter(  this,
                                   android.R.layout.simple_list_item_1,
                                   c,
                                   new String[] {
```

```
                                  Contacts.PeopleColumns.NAME
                                },
                                new int[] {
                                  android.R.id.text1
                                }));
}
```

Assuming you have some contacts in the database, they will appear when you first open the `ContactsDemo` activity, since that is the default perspective:

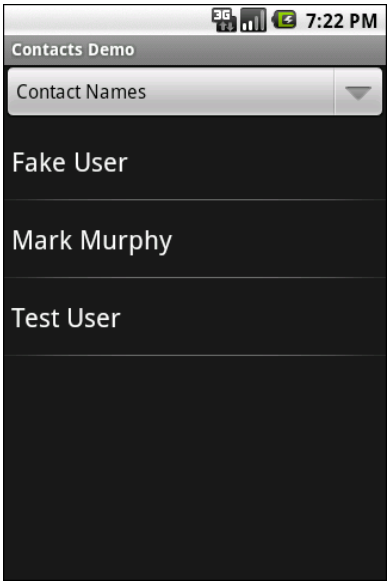


**Figure 37. The ContactsDemo sample application, showing all contacts**

## Accessing Phone Numbers

Retrieving a list of contacts by their phone number can be done by querying the `CONTENT_URI` content provider:

```
private ListAdapter buildPhonesAdapter() {
  String[] PROJECTION=new String[] {  Contacts.Phones._ID,
                                      Contacts.Phones.NAME,
                                      Contacts.Phones.NUMBER
                                    };
  Cursor c=managedQuery(Contacts.Phones.CONTENT_URI,
                        PROJECTION, null, null,
                        Contacts.Phones.DEFAULT_SORT_ORDER);
```

```
  return(new SimpleCursorAdapter(  this,
                                   android.R.layout.simple_list_item_2,
                                   c,
                                   new String[] {
                                     Contacts.Phones.NAME,
                                     Contacts.Phones.NUMBER
                                   },
                                   new int[] {
                                     android.R.id.text1,
                                     android.R.id.text2
                                   }));
}
```

Since the documentation for `Contacts.Phones` shows that it incorporates `Contacts.PeopleColumns` and `Contacts.PhonesColumns`, we know we can get the phone number and the contact's name in one query, which is why both are included in our projection of columns to retrieve.

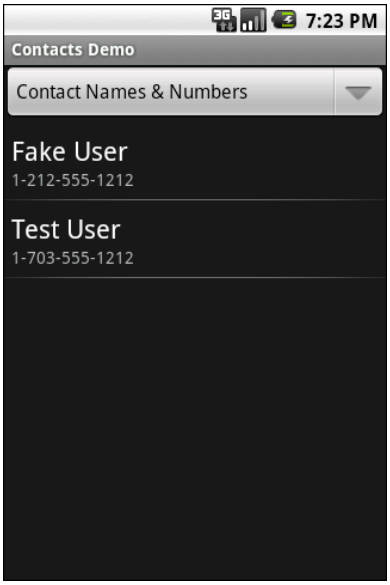


**Figure 38. The ContactsDemo sample application, showing all contacts that have phone numbers**

## Accessing Email Addresses

Similarly, to get a list of all the email addresses, we can use the `CONTENT_EMAIL_URI` content provider, which incorporates the

`Contacts.ContactMethodsColumns` and `Contacts.PeopleColumns`, so we can get access to the contact name as well as the email address itself (`DATA`):

```
private ListAdapter buildEmailAdapter() {
  String[] PROJECTION=new String[] {  Contacts.ContactMethods._ID,
                                      Contacts.ContactMethods.DATA,
                                      Contacts.PeopleColumns.NAME
                                    };
  Cursor c=managedQuery(Contacts.ContactMethods.CONTENT_EMAIL_URI,
                        PROJECTION, null, null,
                        Contacts.ContactMethods.DEFAULT_SORT_ORDER);

  return(new SimpleCursorAdapter(  this,
                                   android.R.layout.simple_list_item_2,
                                   c,
                                   new String[] {
                                     Contacts.PeopleColumns.NAME,
                                     Contacts.ContactMethods.DATA
                                   },
                                   new int[] {
                                     android.R.id.text1,
                                     android.R.id.text2
                                   }));
}
```

Again, the results are displayed via a two-line `SimpleCursorAdapter`:


**Figure 39. The ContactsDemo sample application, showing all contacts with email addresses**

# Rummaging Through Your Phone Records

The `CallLog` content provider in Android gives you access to the calls associated with your phone: the calls you placed, the calls you received, and the calls that you missed. This is a much simpler structure than the `Contacts` content provider described in the previous section.

The columns available to you can be found in the `CallLog.Calls` class. The commonly-used ones include:

- `NUMBER`: the phone number associated with the call
- `DATE`: when the call was placed, in milliseconds-since-the-epoch format
- `DURATION`: how long the call lasted, in seconds
- `TYPE`: indicating if the call was incoming, outgoing, or missed

These, of course, are augmented by the stock `BaseColumns`, which `CallLog.Calls` inherits from.

So, for example, here is a projection used against the call log, from the `JoinDemo` activity in the `Database/JoinCursor` project:

```
private static String[] PROJECTION=new String[] { CallLog.Calls._ID,
                                                  CallLog.Calls.NUMBER,
                                                  CallLog.Calls.DATE,
                                                  CallLog.Calls.DURATION
                                                };
```

Here is where we get a `Cursor` on that projection, with the most-recent calls first in the list:

```
Cursor c=managedQuery(android.provider.CallLog.Calls.CONTENT_URI,
                      PROJECTION, null, null,
                      CallLog.Calls.DATE+" DESC");
```

Unlike contacts, the call log appears unmodifiable by Android applications. So while you can query the log, you cannot add your own calls, delete calls, etc.

Also note that, to access the call log, you need the `READ_CONTACTS` permission.

## Come Together, Right Now

If you have multiple tables within a database, and you want a `Cursor` that represents a join of those tables, you can accomplish that simply through a well-constructed query. However, if you have multiple databases, or you wish to join data in your database with data from a third-party `ContentProvider`, the join becomes significantly more difficult. You cannot simply construct a query, since SQLite has no facility (today) to query a `ContentProvider`, let alone join a `ContentProvider`'s contents with those from native tables.

One solution is to do the join at the `Cursor` itself. Android's `Cursors` offer a fairly vanilla interface, and Android even supplies a `CursorWrapper` class that can handle much of the effort for us. In this section, we will examine the use of `CursorWrapper` to create a `JoinCursor`, blending data from a SQLite table with that from the `CallLog`.

Note that the implementation shown here is for illustrative purposes only. It may suffer from significant performance issues, particularly memory consumption, that would need to be addressed in a serious production application. If you are interested in perhaps pursuing an open source project to implement a better version of `JoinCursor`, contact the author.

Also note that there is a `CursorJoiner` class in the `android.database` package in the SDK. A `CursorJoiner` takes two `Cursor` objects plus a list of key columns, using the key columns to join the `Cursor` values together. This is more efficient but somewhat less flexible that the implemenation shown here.

## CursorWrapper

As the name suggests, `CursorWrapper` wraps a `Cursor` object. Specifically, `CursorWrapper` implements the `Cursor` interface itself and delegates all of the interface's calls to the wrapped `Cursor`.

On the surface, this seems pointless. After all, if `CursorWrapper` simply serves as a pass-through to the `Cursor`, why not use the underlying `Cursor` directly?

The key is not `CursorWrapper` itself, but rather custom subclasses of `CursorWrapper`. You can then override certain `Cursor` methods, to perform work in addition to, or perhaps instead of, passing the call to the wrapped `Cursor`.

In this case, we want to create a `CursorWrapper` subclass that allows us to inject additional columns into the results. These columns will be the result of a join operation between a SQLite table and the `CallLog`.

Specifically, the Database/JoinCursor project adds "call notes" – a block of text about a specific call one made. You could use this concept in a contact management system, for example, to annotate what all was discussed in a call or otherwise document the call itself. Since `CallLog` is not modifiable and has no field for "call notes" anyway, we cannot store such notes in the `CallLog`. Instead, we store those notes in a call_notes SQLite table, mapping the `CallLog` row `_id` to the note.

For simplicity, this example will assume that there are 0 or 1 notes per call, not several. That allows the `JoinCursor` to simply inject the call note into the `CallLog` `Cursor` results, without having to worry about dealing with several possible notes. We do, however, need to deal with the case where the call does not yet have a note.

## Implementing a JoinCursor

A `JoinCursor` is a relatively complex class. Some of that complexity is due to repeated boilerplate code, and some is due to the problem being solved.

What we need the `JoinCursor` to do is:

- Override `Cursor`-related methods that involve the columns
- Check to see if there is a note for the current row
- Adjust the results of the method to accomodate the possibility (or reality) of a note

You can see an implementation of this in the `JoinCursor` class in the `Database/JoinCursor` project:

```
import android.content.ContentValues;
import android.database.Cursor;
import android.database.CursorWrapper;
import java.util.LinkedHashMap;
import java.util.Map;

class JoinCursor extends CursorWrapper {
  private I_JoinHandler join=null;
  private JoinCache cache=new JoinCache(100);

  JoinCursor(Cursor main, I_JoinHandler join) {
    super(main);

    this.join=join;
  }

  public int getColumnCount() {
    return(super.getColumnCount()+join.getColumnNames().length);
  }

  public int getColumnIndex(String columnName) {
    for (int i=0;i<join.getColumnNames().length;i++) {
      if (columnName.equals(join.getColumnNames()[i])) {
        return(super.getColumnCount()+i);
      }
    }

    return(super.getColumnIndex(columnName));
  }

  public int getColumnIndexOrThrow(String columnName) {
    for (int i=0;i<join.getColumnNames().length;i++) {
      if (columnName.equals(join.getColumnNames()[i])) {
        return(super.getColumnCount()+i);
      }
    }

    return(super.getColumnIndexOrThrow(columnName));
  }
```

```
  public String getColumnName(int columnIndex) {
    if (columnIndex>=super.getColumnCount()) {
      return(join.getColumnNames()[columnIndex-super.getColumnCount()]);
    }

    return(super.getColumnName(columnIndex));
  }

  public byte[] getBlob(int columnIndex) {
    if (columnIndex>=super.getColumnCount()) {
      ContentValues extras=cache.get(join.getCacheKey(this));
      int offset=columnIndex-super.getColumnCount();

      return(extras.getAsByteArray(join.getColumnNames()[offset]));
    }

    return(super.getBlob(columnIndex));
  }

  public double getDouble(int columnIndex) {
    if (columnIndex>=super.getColumnCount()) {
      ContentValues extras=cache.get(join.getCacheKey(this));
      int offset=columnIndex-super.getColumnCount();

      return(extras.getAsDouble(join.getColumnNames()[offset]));
    }

    return(super.getDouble(columnIndex));
  }

  public float getFloat(int columnIndex) {
    if (columnIndex>=super.getColumnCount()) {
      ContentValues extras=cache.get(join.getCacheKey(this));
      int offset=columnIndex-super.getColumnCount();

      return(extras.getAsFloat(join.getColumnNames()[offset]));
    }

    return(super.getFloat(columnIndex));
  }

  public int getInt(int columnIndex) {
    if (columnIndex>=super.getColumnCount()) {
      ContentValues extras=cache.get(join.getCacheKey(this));
      int offset=columnIndex-super.getColumnCount();

      return(extras.getAsInteger(join.getColumnNames()[offset]));
    }

    return(super.getInt(columnIndex));
  }

  public long getLong(int columnIndex) {
```

```
    if (columnIndex>=super.getColumnCount()) {
      ContentValues extras=cache.get(join.getCacheKey(this));
      int offset=columnIndex-super.getColumnCount();

      return(extras.getAsLong(join.getColumnNames()[offset]));
    }

    return(super.getLong(columnIndex));
  }

  public short getShort(int columnIndex) {
    if (columnIndex>=super.getColumnCount()) {
      ContentValues extras=cache.get(join.getCacheKey(this));
      int offset=columnIndex-super.getColumnCount();

      return(extras.getAsShort(join.getColumnNames()[offset]));
    }

    return(super.getShort(columnIndex));
  }

  public String getString(int columnIndex) {
    if (columnIndex>=super.getColumnCount()) {
      ContentValues extras=cache.get(join.getCacheKey(this));
      int offset=columnIndex-super.getColumnCount();

      return(extras.getAsString(join.getColumnNames()[offset]));
    }

    return(super.getString(columnIndex));
  }

  public boolean isNull(int columnIndex) {
    if (columnIndex>=super.getColumnCount()) {
      ContentValues extras=cache.get(join.getCacheKey(this));
      int offset=columnIndex-super.getColumnCount();

      return(extras.get(join.getColumnNames()[offset])==null);
    }

    return(super.isNull(columnIndex));
  }

  public boolean requery() {
    cache.clear();

    return(super.requery());
  }

  class JoinCache extends LinkedHashMap<String, ContentValues> {
    private int capacity=100;

    JoinCache(int capacity) {
      super(capacity+1, 1.1f, true);
```

```
      this.capacity=capacity;
    }

    protected boolean removeEldestEntry(Entry<String, ContentValues> eldest) {
      return(size()>capacity);
    }

    ContentValues get(String key) {
      ContentValues result=super.get(key);

      if (result==null) {
        result=join.getJoin(JoinCursor.this);
        put(key, result);
      }

      return(result);
    }
  }
}
```

`JoinCursor`, when instantiated, gets both the `Cursor` to wrap and an `I_JoinHandler` instance. The join handler is responsible for getting the extra columns for a given row:

```
import android.content.ContentValues;
import android.database.Cursor;
import java.util.Map;

public interface I_JoinHandler {
  String[] getColumnNames();
  String getCacheKey(Cursor c);
  ContentValues getJoin(Cursor c);
}
```

Most of `JoinCursor` is then using the `I_JoinHandler` information to adjust the results of various `Cursor` methods. For example:

- `getColumnCount()` returns the sum of the `Cursor`'s column count and the number of extra columns returned by the join handler
- `getColumnIndex()` and kin need to search through the join handler's columns as well as the `Cursor`'s to find the match, if any
- `getInt()`, `isNull()`, and kin need to support retrieving values from both the `Cursor` and the join handler

To improve performance, `JoinCursor` keeps a cache of the extra values for requested rows, using an "LRU cache"-style `LinkedHashMap` and an inner `JoinCache` class. The `JoinCache` keeps the `ContentValues` returned by `I_JoinHandler` on a `getJoin()` call, representing the extra columns (if any) for that particular `Cursor` row. Since we are caching data, however, we need to flush that cache sometimes; in this case, we override `requery()` to flush the cache if the `Cursor` itself is being proactively updated.

## Using a JoinCursor

To use a `JoinCursor`, of course, you need an implementation of `I_JoinHandler`, such as this one from the `JoinDemo` activity:

```
I_JoinHandler join=new I_JoinHandler() {
  String[] columns={NOTE_ID, NOTE};

  public String[] getColumnNames() {
      return(columns);
  }

  public String getCacheKey(Cursor c) {
      return(String.valueOf(c.getInt(c.getColumnIndex(CallLog.Calls._ID))));
  }

  public ContentValues getJoin(Cursor c) {
    String[] args={getCacheKey(c)};
    Cursor j=getDb().rawQuery("SELECT _ID, note FROM call_notes WHERE
call_id=?", args);
    ContentValues result=new ContentValues();

    j.moveToFirst();

    if (j.isAfterLast()) {
      result.put(columns[0], -1);
      result.put(columns[1], (String)null);
    }
    else {
      result.put(columns[0], j.getInt(0));
      result.put(columns[1], j.getString(1));
    }

    j.close();

    return(result);
  }
};
```

The columns are a fixed pair (the note's ID and the note itself). These are retrieved via `getJoin()` from the `call_notes` SQLite table. The call notes themselves are keyed by the call's own `_id`, which is also used as the key for the `JoinCursor`'s cache of results. The net effect is that we only ever retrieve a note once for a given call, at least until a `requery()`. And, if there is no note for the call, we use a `null` note to indicate that we are, indeed, note-free for this call.

The note information is then used by our `CursorAdapter` subclass (`CallPlusAdapter`) and its associated `ViewWrapper`, also found in the `JoinDemo` activity:

```
class CallPlusAdapter extends CursorAdapter {
  CallPlusAdapter(Cursor c) {
    super(JoinDemo.this, c);
  }

  @Override
  public void bindView(View row, Context ctxt,
                       Cursor c) {
    ViewWrapper wrapper=(ViewWrapper)row.getTag();

    wrapper.update(c);
  }

  @Override
  public View newView(Context ctxt, Cursor c,
                      ViewGroup parent) {
    LayoutInflater inflater=getLayoutInflater();

    View row=inflater.inflate(R.layout.row, null);
    ViewWrapper wrapper=new ViewWrapper(row);

    row.setTag(wrapper);
    wrapper.update(c);

    return(row);
  }
}

class ViewWrapper {
  View base;
  TextView number=null;
  TextView duration=null;
  TextView time=null;
  ImageView icon=null;

  ViewWrapper(View base) {
    this.base=base;
```

```
  }

  TextView getNumber() {
    if (number==null) {
      number=(TextView)base.findViewById(R.id.number);
    }

    return(number);
  }

  TextView getDuration() {
    if (duration==null) {
      duration=(TextView)base.findViewById(R.id.duration);
    }

    return(duration);
  }

  TextView getTime() {
    if (time==null) {
      time=(TextView)base.findViewById(R.id.time);
    }

    return(time);
  }

  ImageView getIcon() {
    if (icon==null) {
      icon=(ImageView)base.findViewById(R.id.note);
    }

    return(icon);
  }

  void update(Cursor c) {
    getNumber().setText(c.getString(c.getColumnIndex(CallLog.Calls.NUMBER)));
    getTime().setText(FORMAT.format(c.getInt(c.getColumnIndex(CallLog.Calls.DATE
))));
    getDuration().setText(c.getString(c.getColumnIndex(CallLog.Calls.DURATION))
+" seconds");

    String note=c.getString(c.getColumnIndex(NOTE));

    if (note!=null && note.length()>0) {
      getIcon().setVisibility(View.VISIBLE);
    }
    else {
      getIcon().setVisibility(View.GONE);
    }
  }
}
```

Mostly, we are populating a row to go in a `ListView` based off of the call data (e.g., duration). However, if there is a non-`null` note, we also display an icon in the row, indicating that a note is available.

The `JoinDemo` activity itself is just a `ListActivity`, using the `CallPlusAdapter` and the `CallLog` `Cursor` we saw in the previous section:

```
import android.app.ListActivity;
import android.content.ContentValues;
import android.content.Context;
import android.content.Intent;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.os.Bundle;
import android.provider.CallLog;
import android.view.View;
import android.view.ViewGroup;
import android.view.LayoutInflater;
import android.widget.CursorAdapter;
import android.widget.ImageView;
import android.widget.ListView;
import android.widget.TextView;
import java.text.SimpleDateFormat;

public class JoinDemo extends ListActivity {
  public static String NOTE="_NOTE";
  private static String NOTE_ID="NOTE_ID";
  private static String[] PROJECTION=new String[] { CallLog.Calls._ID,
                                                    CallLog.Calls.NUMBER,
                                                    CallLog.Calls.DATE,
                                                    CallLog.Calls.DURATION
                                                  };
  private static SimpleDateFormat FORMAT=new SimpleDateFormat("MM/d h:mm a");
  private Cursor cursor=null;
  private int noteColumn=-1;
  private int idColumn=-1;
  private int noteIdColumn=-1;
  private SQLiteDatabase db=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    Cursor c=managedQuery(android.provider.CallLog.Calls.CONTENT_URI,
                          PROJECTION, null, null,
                          CallLog.Calls.DATE+" DESC");

    cursor=new JoinCursor(c, join);
    noteColumn=cursor.getColumnIndex(NOTE);
    idColumn=cursor.getColumnIndex(CallLog.Calls._ID);
    noteIdColumn=cursor.getColumnIndex(NOTE_ID);
```

```
    setListAdapter(new CallPlusAdapter(cursor));
  }

  @Override
  public void onResume() {
    super.onResume();

    cursor.requery();
  }

  @Override
  public void onDestroy() {
    super.onDestroy();

    if (db!=null) {
      db.close();
    }
  }

  @Override
  protected void onListItemClick(ListView l, View v,
                                 int position, long id) {
    cursor.moveToPosition(position);

    String note=cursor.getString(noteColumn);

    if (note==null || note.length()==0) {
      Intent i=new Intent(this, NoteEditor.class);

      i.putExtra(NOTE, note);
      i.putExtra("call_id", cursor.getInt(idColumn));
      i.putExtra("note_id", cursor.getInt(noteIdColumn));
      startActivityForResult(i, 1);
    }
    else {
      Intent i=new Intent(this, NoteActivity.class);

      i.putExtra(NOTE, note);
      startActivity(i);
    }
  }

  @Override
  protected void onActivityResult(int requestCode,
                                  int resultCode,
                                  Intent data) {
    String note=data.getStringExtra(NOTE);

    if (note!=null) {
      int noteId=data.getIntExtra(NOTE_ID, -1);
      ContentValues cv=new ContentValues();

      cv.put("note", note);
```

```
      if (noteId==-1) {
        int callId=data.getIntExtra("call_id", -1);

        cv.put("call_id", callId);

        getDb().insertOrThrow("call_notes", "_id", cv);
      }
      else {
        String[] args={String.valueOf(noteId)};

        getDb().update("call_notes", cv, "_ID", args);
      }
    }
  }

  SQLiteDatabase getDb() {
    if (db==null) {
      db=(new NotesInstaller(JoinDemo.this)).getWritableDatabase();
    }

    return(db);
  }

  I_JoinHandler join=new I_JoinHandler() {
    String[] columns={NOTE_ID, NOTE};

    public String[] getColumnNames() {
        return(columns);
    }

    public String getCacheKey(Cursor c) {
        return(String.valueOf(c.getInt(c.getColumnIndex(CallLog.Calls._ID))));
    }

    public ContentValues getJoin(Cursor c) {
      String[] args={getCacheKey(c)};
      Cursor j=getDb().rawQuery("SELECT _ID, note FROM call_notes WHERE
call_id=?", args);
      ContentValues result=new ContentValues();

      j.moveToFirst();

      if (j.isAfterLast()) {
        result.put(columns[0], -1);
        result.put(columns[1], (String)null);
      }
      else {
        result.put(columns[0], j.getInt(0));
        result.put(columns[1], j.getString(1));
      }

      j.close();

      return(result);
```

```
    }
  };

  class CallPlusAdapter extends CursorAdapter {
    CallPlusAdapter(Cursor c) {
      super(JoinDemo.this, c);
    }

    @Override
    public void bindView(View row, Context ctxt,
                         Cursor c) {
      ViewWrapper wrapper=(ViewWrapper)row.getTag();

      wrapper.update(c);
    }

    @Override
    public View newView(Context ctxt, Cursor c,
                        ViewGroup parent) {
      LayoutInflater inflater=getLayoutInflater();

      View row=inflater.inflate(R.layout.row, null);
      ViewWrapper wrapper=new ViewWrapper(row);

      row.setTag(wrapper);
      wrapper.update(c);

      return(row);
    }
  }

  class ViewWrapper {
    View base;
    TextView number=null;
    TextView duration=null;
    TextView time=null;
    ImageView icon=null;

    ViewWrapper(View base) {
      this.base=base;
    }

    TextView getNumber() {
      if (number==null) {
        number=(TextView)base.findViewById(R.id.number);
      }

      return(number);
    }

    TextView getDuration() {
      if (duration==null) {
        duration=(TextView)base.findViewById(R.id.duration);
      }
```

```
      return(duration);
    }

    TextView getTime() {
      if (time==null) {
        time=(TextView)base.findViewById(R.id.time);
      }

      return(time);
    }

    ImageView getIcon() {
      if (icon==null) {
        icon=(ImageView)base.findViewById(R.id.note);
      }

      return(icon);
    }

    void update(Cursor c) {
      getNumber().setText(c.getString(c.getColumnIndex(CallLog.Calls.NUMBER)));
      getTime().setText(FORMAT.format(c.getInt(c.getColumnIndex(CallLog.Calls.DA
TE))));
      getDuration().setText(c.getString(c.getColumnIndex(CallLog.Calls.DURATION)
)+" seconds");

      String note=c.getString(c.getColumnIndex(NOTE));

      if (note!=null && note.length()>0) {
        getIcon().setVisibility(View.VISIBLE);
      }
      else {
        getIcon().setVisibility(View.GONE);
      }
    }
  }
}
```

When the user clicks on a row, depending on whether there is a note, we either spawn a `NoteEditor` (to create a new note) or a `NoteActivity` (to view an existing note). In a real implementation of this functionality, of course, we would allow users to edit existing notes, delete notes, and the like, all of which is skipped in this simplified sample application.

Visually, the activity does not look like much, but you will see the note icon on calls containing notes (with some phone numbers smudged for privacy):

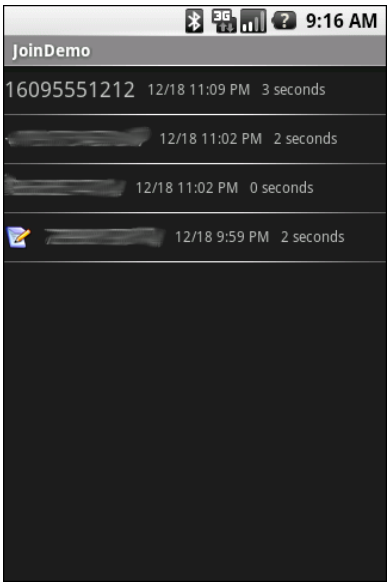


**Figure 40. The JoinCursor sample application, showing one call with a note**

CHAPTER 11

# Handling System Events

If you have ever looked at the list of available `Intent` actions in the SDK documentation for the `Intent` class, you will see that there are lots of possible actions.

Lots and lots and lots of possible actions.

There are even actions that are not listed in that spot in the documentation, but are scattered throughout the rest of the SDK documentation.

The vast majority of these you will never raise yourself. Instead, they are broadcast by Android, to signify certain system events that have occurred and that you might want to take note of, if they affect the operation of your application.

This chapter examines a few of these, to give you the sense of what is possible and how to make use of these sorts of events.

## Get Moving, First Thing

A popular request is to have a service get control when the device is powered on.