

SQLite & Content Providers

SQLite database is used for saving large amount of repeating data such as contacts information, the music library or photos information in the internal storage of the android device. The SQLite database information is saved in the internal storage in a secure not accessible to other apps section. This guide assumes you are familiar with SQL queries and shows how to save data in an SQL local database on the android device.

open database and creating tables

First you need to create an SQLiteDatabase instance using the **openOrCreateDatabase** (table_name, SQLiteDatabase.CREATE_IF_NECESSARY, null); the first parameter is the database path the second is the database flags and the null parameter at the end can be an error handling object.

After creating the database you can add as many tables to it as you want. here is an example of an SQL query designed to open a table called "tbl_programmers" with three columns, the first is an integer that is the primary key (unique identifier for every record) it is designed to be an auto increase integer for each record, and another two representing the first and last name of type text, to be saved in the table.

```
final String CREATE_TABLE_CMD="CREATE TABLE IF NOT EXISTS tbl_programmers(id  
INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT, family TEXT);";
```

When you want to execute the query on the database you created before you can use the database function **execSQL**(query) which is used to Execute a single SQL statement that is NOT a SELECT or any other SQL statement that returns data.

Adding information to the database

First create a new **ContentValues** object and add the values you want according to each key you created before. the keys are the columns names and the values should be you desired data for each column. each ContentValues will be a single line (record) in your table.

For example, to the table above we will add a record like this:

```
ContentValues values=new ContentValues();  
values.put("name", nameEt.getText().toString());  
values.put("family",family Et.getText().toString());
```

when you want to add the record to the table use the database **insert** function:

```
database.insert("tbl_programmers", null, values);
```

Querying the database for information

If we want all the records and all the details(columns for each record at no specific order) we can use the database **query** method setting only the table name and setting all the other parameters to null, the method return a **Cursor** object which is a marker designed to go over the table record after record:

```
Cursor cursor=database.query("tbl_programmers", null,null, null, null, null, null);
```

The cursor points to one record (one line) in the table at one time in a linear order. You can access the first line using the cursor.**moveToFirst()** method that will return false on an empty table and when you want to advance the cursor to the next record use cursor.**moveToNext()** which will return true only if there is a next line so you can use it in a while loop. when you finished going over the record close the cursor using cursor.**close()**; Cursor action are relatively slow and should be done from a background thread.

Retrieving the data from the record: when you want a specific column get her index using the cursor **getColumnIndex(column_name)** and to get the values stored in the column using the cursor **getString(index)** passing the index you got before. If you want a more complex SELECT

queries you can pass more parameters to the query function, the query function builds the sql query for us with the params passed to it:

`query(String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy)`

table	The table name to compile the query against.
columns	A list of which columns to return. Passing null will return all columns, which is discouraged to prevent reading data from storage that isn't going to be used.
selection	A filter declaring which rows to return, formatted as an SQL WHERE clause (excluding the WHERE itself). Passing null will return all rows for the given table. for example "name LIKE a%" this will gives us all the names the start with "a"
selectionArgs	You may include ?s in selection, which will be replaced by the values from selectionArgs, in order that they appear in the selection. The values will be bound as Strings.
groupBy	A filter declaring how to group rows, formatted as an SQL GROUP BY clause (excluding the GROUP BY itself). Passing null will cause the rows to not be grouped.
having	A filter declare which row groups to include in the cursor, if row grouping is being used, formatted as an SQL HAVING clause (excluding the HAVING itself). Passing null will cause all row groups to be included, and is required when row grouping is not being used.
sorderBy	How to order the rows, formatted as an SQL ORDER BY clause (excluding the ORDER BY itself). Passing null will use the default sort order, which may be unordered.

SimpleCursorAdapter

In the previous example we managed the cursor ourselves, meaning, we read each line one after the other while advancing the cursor. The api also helps us manage the cursor and retrieve the data from it using the SimpleCursorAdapter. This adapter gets a cursor to extract the data

from, a cell layout, an array of int id[] and a string[] of columns names. For each string at position i in the column array it takes the data from the cursor and maps it to a view with an id mentioned in the same position i at the int[] id array. All the ids must be a valid view's id the xml layout file passed. For example to the table shown above lets assume that there is a line xml file named listline with two textviews with id's nameTv and familyTv then we should create these two arrays:

```
String[] from={"name","family"};
```

```
// these are the column names to retrieve from the cursor - these columns must be included in  
the query that returns the cursor - the second param - projection
```

```
int[] to={R.id.name, R.id.familyTv};
```

```
// these are the ids of the two textviews in the listline xml file we created for the table row
```

We create the adapter as follows:

```
new SimpleCursorAdapter(context, R.layout.listline, cursor, from, to, 0);
```

CursorLoader

In the example above we can't close the cursor after assigning the adapter to the list since the adapter is loaded asynchronously. Before android 3.0 we would have asked the activity to managed the cursor opening and closing when needed by calling **startManagingCursor(cursor)** and **stopManagingCursor(cursor)**. This Tells the activity to take care of managing the cursor's lifecycle based on the activity's lifecycle. The cursor will automatically be deactivated (**deactivate()**) when the activity is stopped, and will automatically be closed (**close()**) when the activity is destroyed. When the activity is stopped and then later restarted, the Cursor is re-queried (**requery()**) for the most up-to-date data (same with managedQuery). Now this is deprecated.

Having the activity manage the cursor seems convenient at first, as we no longer need to worry about deactivating/closing the cursor ourselves. However, this signals the activity to call **requery()** on the cursor **each time the activity returns from a stopped state**, and therefore puts the UI thread at risk. This cost significantly outweighs the convenience of having the activity deactivate/close the cursor for us.

The solution to this is to use CursorLoader. The new **Loader** API is a huge step forward, and significantly improves the user experience. **Loaders** ensure that all cursor operations are done asynchronously, thus eliminating the possibility of blocking the UI thread. Further, when managed by the **LoaderManager**, **Loaders** retain their existing cursor data across the activity instance (for example, when it is restarted due to a configuration change), thus saving the cursor from unnecessary, potentially expensive re-queries. As an added bonus, **Loaders** are intelligent enough to monitor the underlying data source for updates, re-querying automatically when the data is changed.

The LoaderManager does not know how data is loaded, nor does it need to. Rather, the LoaderManager instructs its Loaders when to start/stop/reset their load, retaining their state across configuration changes and providing a simple interface for delivering results back to the client. In this way, the LoaderManager is a much more intelligent and generic implementation of the now-deprecated **startManagingCursor** method.

So the steps to use the CursorLoader are:

1. Create the adapter as before but pass a null cursor. We will pass the adapter a Cursor only when the data has finished loading for the first time (i.e. when the LoaderManager delivers the data to onLoadFinished callback method). Also pass '0' flag as the last argument. This prevents the adapter from registering a ContentObserver(monitors data changes) for the Cursor (the CursorLoader will do this for us!).
2. Get the loader manager instance by calling getLoaderManager() and init it with a unique id, Loader ids are specific to the Activity or Fragment in which they reside, this will identify the specific loader in the callback method, and a reference to the LoaderManager.LoaderCallbacks instance.
3. Implement the callback methods: first the **onCreateLoader**, this is called when the LoaderManager first create the Loader here we will create the CursorLoader as we created the Cursor before. in the **onLoadFinished(Loader<Cursor> loader, Cursor cursor)** check the loader id and now when the data is available we can associate the queried Cursor passed with the SimpleCursorAdapter that we have created before like this

mAdapter.swapCursor(cursor) - this will update the UI and the ListView. The **onLoaderReset(Loader<Cursor> loader)** is called the Loader's data is now unavailable. Remove any references to the old data by replacing it with **mAdapter.swapCursor(null)**.

Delete Information from a Database

To delete rows from a table, you need to provide selection criteria that identify the rows. and call the database delete function. You can also pass the selection arguments as the third param that way you can protect against sql injection, but we don't have to use it. for example if we want to delete all rows starts with 'a': String where=**"name LIKE 'a%'"**;
database.delete("tbl_programmers",where,null);

Update a Database

When you need to modify a subset of your database values, use the update() method. Updating the table combines the content values syntax of insert() with the where syntax of delete().

```
ContentValues values = new ContentValues();  
values.put("name", "moshe");  
  
String selection = "name LIKE 'dave'";  
int count = db.update("tbl_programmers",values,selection,null);
```

Content providers

Content providers manage access to a structured set of data. Content providers are the standard interface that connects data in one process with code running in another process. A provider is part of an Android application, which often provides its own UI for working with the data. **However, content providers are primarily intended to be used by other applications to retrieve data.**

When you want to access data in a content provider, you use the ContentResolver object in your application's Context to communicate with the provider as a client. The provider object receives data requests from clients, performs the requested action, and returns the results.

Android itself includes content providers that manage data such as audio, video, images, and personal contact information. In the package **android.provider** you can find convenience classes to access the content providers supplied by Android. Android ships with a number of content providers that store common data such as contact information, calendar information, and media files. The classes in that package provide simplified methods of adding or retrieving data from these content providers.

contacts example

The class `ContactsContract` is the contract between the contacts provider and applications. It contains definitions for contacts supported URIs (all the tables addresses) and the column's title. There are few URIs you can use for example, if we want all the information about the contact we need the `ContactsContract.Contacts.CONTENT_URI`

and if we want the phones we need the following table

`ContactsContract.CommonDataKinds.Phone.CONTENT_URI`

and the most used columns in that table are

`ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME` and

`ContactsContract.CommonDataKinds.Phone.NUMBER`

An application accesses the data from a content provider with a **ContentResolver** client object. So to retrieve a cursor of all the phones table all we need is

```
getContentResolver().query(ContactsContract.CommonDataKinds.Phone.CONTENT_URI, null, null, null, null);
```

Which returns a cursor allows us to extract information from that table manually, by iterating with that cursor or use the `CursorAdapter` to extract the information for us.

Here is an example of doing so manually:

```
ArrayList<String> contacts = new ArrayList<String>();
Cursor phones = getContentResolver().query(ContactsContract.CommonDataKinds.Phone.
    CONTENT_URI, null, null, null, null);
while (phones.moveToNext())
{
    String name = phones.getString(phones.getColumnIndex(ContactsContract.
        CommonDataKinds.Phone.DISPLAY_NAME));
```

```

        String phoneNumber = phones.getString(phones.getColumnIndex(ContactsContract.
            CommonDataKinds.Phone.NUMBER));
        contacts.add(name+" ", "+phoneNumber);
    }
    phones.close();

```

Note: The query() method should be called from a separate thread to avoid blocking your app's UI thread. (For simplicity of the sample, this code doesn't do that. Consider using CursorLoader to perform the query.

And here is an example of using the simple cursor adapter:

```

Cursor c = getContentResolver().query(ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
    null, null, null, null);
String[] columns = new String[] {ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME,
    ContactsContract.CommonDataKinds.Phone.NUMBER};
int[] views = new int[] { android.R.id.text1, android.R.id.text2 };
SimpleCursorAdapter adapter = new SimpleCursorAdapter(this,android.R.layout.simple_list_item_2, c,
    columns, views, 0);
setListAdapter(adapter);

```