17

Android
Persistency: SQL Databases

Victor Matos
Cleveland State University

Notes are based on:

Android Developers
http://developer.android.com/index.html

SQL Databases

Using SQL databases in Android.

Android (as well as iPhone OS) uses an embedded standalone program called **sqlite3** which can be used to:

create a database, define SQL tables, indices, queries, views, triggers	Insert rows, delete rows, change rows, run queries and administer a SQLite database file.

SQL Databases

Using SQLite

1. SQLite implements most of the SQL-92 standard for SQL.
2. It has partial support for triggers and allows most complex queries (exception made for *outer joins*).
3. SQLITE *does not implement referential integrity constraints* through the *foreign key* constraint model.
4. SQLite uses a *relaxed data typing model*.
5. Instead of assigning a type to an entire column, types are assigned to individual values. This is similar to the *Variant* type in Visual Basic.
6. Therefore it is possible to insert a string into numeric column and so on.

Documentation on SQLITE available at http://www.sqlite.org/sqlite.html
Good GUI tool for SQLITE available at: http://sqliteadmin.orbmu2k.de/

SQL Databases

How to create a SQLite database?
Method 1

public static *SQLiteDatabase*.**openDatabase** (
String path, *SQLiteDatabase.CursorFactory* factory, *int* flags)

Open the database according to the flags OPEN_READWRITE OPEN_READONLY CREATE_IF_NECESSARY . Sets the locale of the database to the the system's current locale.

Parameters

path	to database file to open and/or create
factory	an optional factory class that is called to instantiate a cursor when query is called, or null for default
flags	to control database access mode
Returns	the newly opened database
Throws	*SQLiteException* if the database cannot be opened

SQL Databases

Example 1. Create a SQLite Database

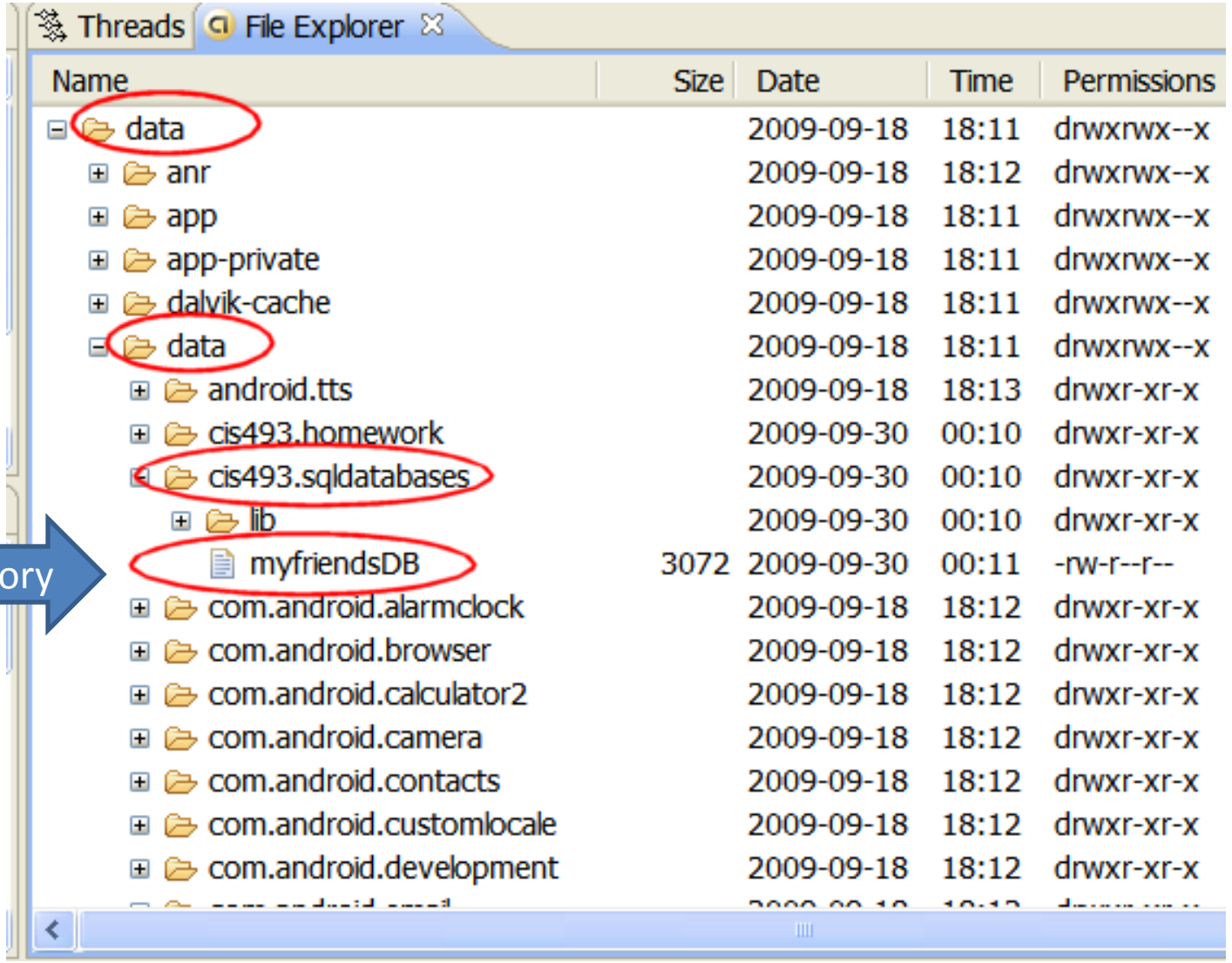
```
package cis493.sqldatabases;
import android.app.Activity;
import android.database.sqlite.*;
import android.os.Bundle;
import android.widget.Toast;

public class SQLDemo1 extends Activity {
SQLiteDatabase db;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
       //   filePath is a complete destination of the form
        //  "/data/data/<namespace>/<databaseName>"
        //  "/sdcard/<databasename>"
        try {
           db = SQLiteDatabase.openDatabase(
                    "/data/data/cis493.sqldatabases/myfriendsDB",
                    null,
                    SQLiteDatabase.CREATE_IF_NECESSARY);
           db.close();
           }
        catch (SQLiteException e) {
           Toast.makeText(this, e.getMessage(), 1).show();
        }
    }// onCreate
}//class
```

SQL Databases

Example 1. Create a SQLite Database

Android's System Image

Device's memory



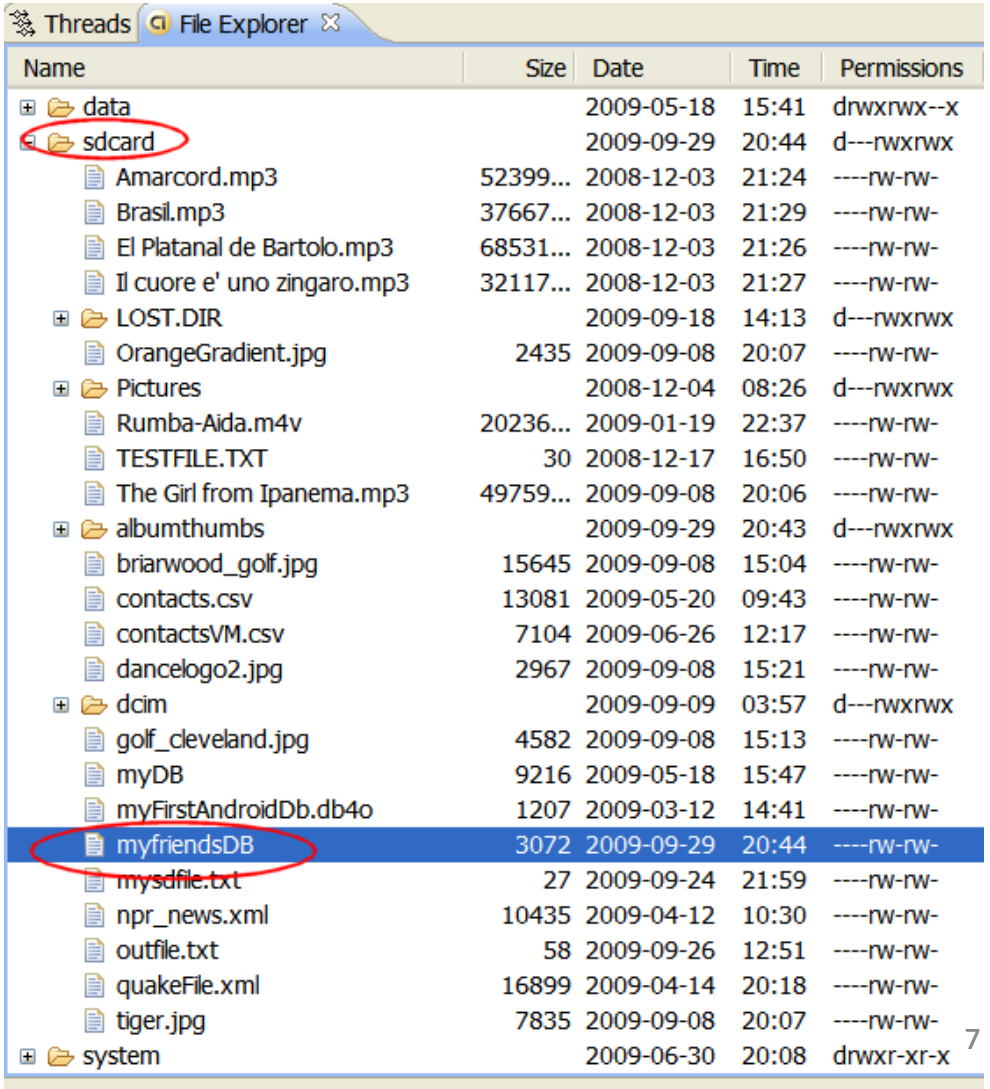
SQL Databases

Example 1. Create a SQLite Database

Creating the database file in the SD card

Using:

```
db = SQLiteDatabase.openDatabase(
  "sdcard/myfriendsDB",
   null,
  SQLiteDatabase.CREATE_IF_NECESSARY);
```



SQL Databases

Warning

1. Beware of sharing issues. You *cannot* access other people's database resources (instead use Content Providers or SD resident DBs).

2. An SD resident database requires the Manifest to include:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

NOTE: *SQLITE (as well as most DBMSs) is not case sensitive.*

SQL Databases

Example2

An alternative way of opening/creating a SQLITE database in your local Android's data space is given below

```
SQLiteDatabase  db =  this.openOrCreateDatabase(
                                        "myfriendsDB",
                                        MODE_PRIVATE,
                                        null);
```

where the assumed prefix for the database stored in the devices ram is: "**/data/data/<CURRENT_namespace>/databases/**". For instance if this app is created in a namespace called "**cis493.sql1**", the full name of the newly created database will be: "**/data/data/cis493.sql1/databases/myfriendsDB**".

This file could later be used by other activities in the app or exported out of the emulator (adb push…) and given to a tool such as SQLITE_ADMINISTRATOR (see notes at the end).

SQL Databases

Example2

An alternative way of opening/creating a SQLITE database in your local Android's System Image is given below

```
SQLiteDatabase  db =  this.openOrCreateDatabase(
                              "myfriendsDB2",
                               MODE_PRIVATE,
                               null);
```

Where:

1. **"myFriendsDB2"** is the abbreviated file path. The prefix is assigned by Android as: /data/data/<app namespace>/databases/myFriendsDB2.

2. **MODE** could be: MODE_PRIVATE, MODE_WORLD_READABLE, and MODE_WORLD_WRITEABLE. Meaningful for apps consisting of multiples activities.

3. **null** refers to optional **factory** class parameter (skip for now)

SQL Databases

Example2

An alternative way of opening/creating a SQLITE database in your local Android's System Image is given below

```
SQLiteDatabase  db =  this.openOrCreateDatabase(
                              "myfriendsDB2",
                               MODE_PRIVATE,
                               null);
```

Where:

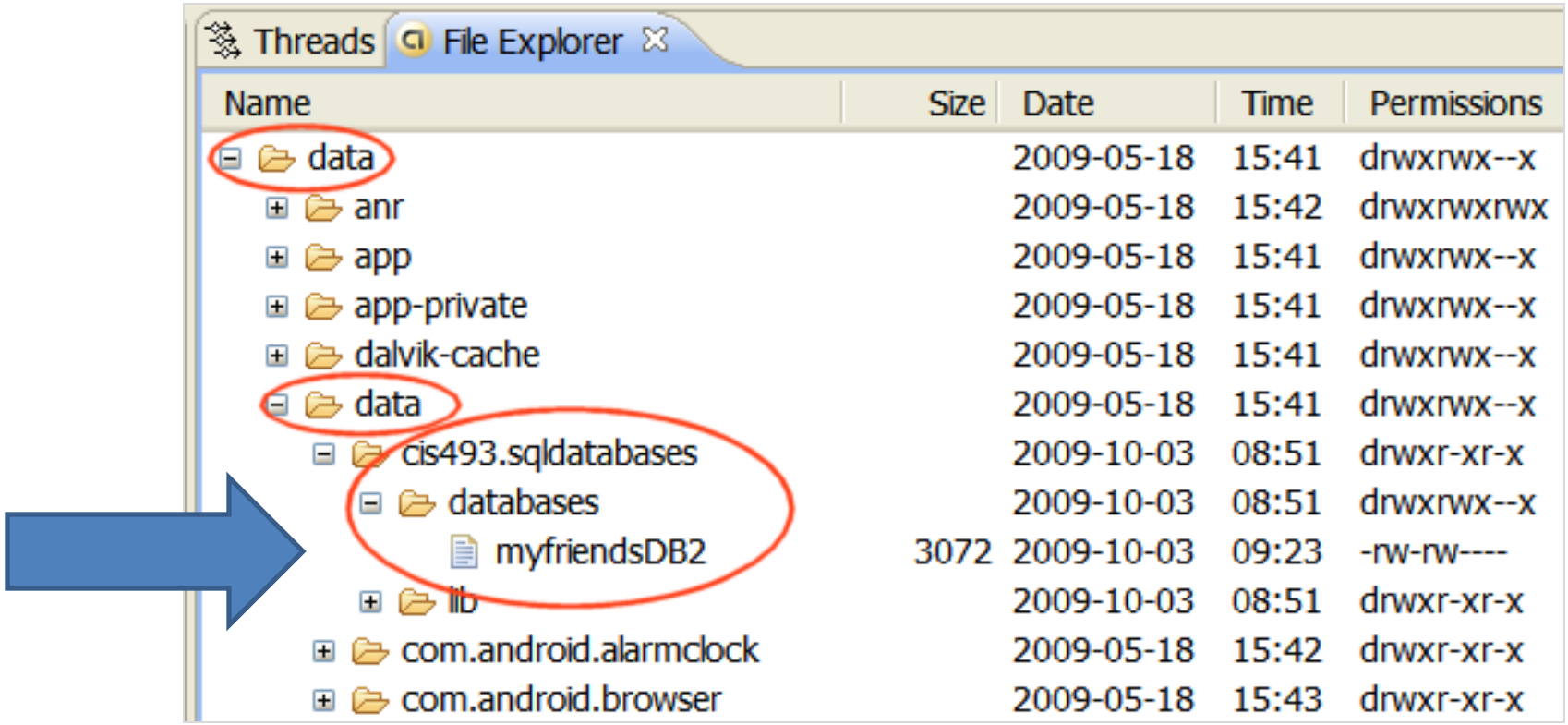
1. **"myFriendsDB2"** is the abbreviated file path. The prefix is assigned by Android as: /data/data/<app namespace>/databases/myFriendsDB2.

2. **MODE** could be: MODE_PRIVATE, MODE_WORLD_READABLE, and MODE_WORLD_WRITEABLE. Meaningful for apps consisting of multiples activities.

3. **null** refers to optional **factory** class parameter (skip for now)

SQL Databases

Example2. Database is saved in the device's memory



SQL Databases

Executing SQL commands on the Database

Once created, the database is ready for normal operations such as:
creating, altering, dropping resources (tables, indices, triggers, views, queries etc.) or administrating database resources (containers, users, …).

***Action* queries** and ***Retrieval* queries** represent the most common operations against the database.

- A *retrieval* query is typically a *SQL-Select* command in which a table holding a number of fields and rows is produced as an answer to a data request.

- An *action* query usually performs maintenance and administrative tasks such as manipulating tables, users, environment, etc.

SQL Databases

Transaction Processing

Transactions are desirable because they contribute to maintain consistent data and prevent unwanted losses due to abnormal termination of execution.

In general it is convenient to process *action queries* inside the protective frame of a *database transaction* in which the policy of "*complete success or total failure*" is transparently enforced.

*This notion is called: **atomicity** to reflect that all parts of a method are fused in an indivisible-like statement.*

SQL Databases

Transaction Processing

The typical Android way of running transactions on a SQLiteDatabase is illustrated in the following fragment (Assume **db** is defined as a SQLiteDatabase)

```
db.beginTransaction();
try {
        //perform your database operations here ...
        db.setTransactionSuccessful(); //commit your changes
}
catch (SQLiteException e) {
        //report problem
}
finally {
        db.endTransaction();
}
```

The transaction is defined between the methods: *beginTransaction* and *endTransaction*. You need to issue the *setTransactionSuccessful()*call to commit any changes. The absence of it provokes an implicit *rollback; consequently the database is reset to the state previous to the beginning of the transaction*

SQL Databases

recID	name	phone
1	AAA	555
2	BBB	777
3	CCC	999

Creating-Populating a Table

SQL Syntax for the creating and populating of a table looks like this:

```
create table tblAMIGO (
            recID integer PRIMARY KEY autoincrement,
            name  text,
            phone text );
```

```
insert into tblAMIGO(name, phone) values ('AAA', '555' );
```

SQL Databases

recID	name	phone
1	AAA	555
2	BBB	777
3	CCC	999

Creating-Populating a Table

We will use the **execSQL(…)** method to manipulate *SQL action queries*. The following example creates a new table called **tblAmigo**.

The table has three fields: a numeric unique identifier called *recID*, and two string fields representing our friend's *name* and *phone*. If a table with such a name exists it is first dropped and then created anew. Finally three rows are inserted in the table.

***Note**: for presentation economy we do not show the entire code which should include a transaction frame.*

```
db.execSQL("create table tblAMIGO ("
                + " recID integer PRIMARY KEY autoincrement, "
                + " name  text, "
                + " phone text );  "    );

db.execSQL( "insert into tblAMIGO(name, phone) values ('AAA', '555' );"  );
db.execSQL( "insert into tblAMIGO(name, phone) values ('BBB', '777' );"  );
db.execSQL( "insert into tblAMIGO(name, phone) values ('CCC', '999' );"  );
```

SQL Databases

Creating-Populating a Table

Comments

1. The field **recID** is defined as **PRIMARY KEY** of the table. The "*autoincrement*" feature guarantees that each new record will be given a unique serial number (0,1,2,…).

2. The database data types are very simple, for instance we will use: ***text, varchar, integer, float, numeric, date, time, timestamp, blob, boolean,*** and so on.

1. In general, any well-formed SQL action command (insert, delete, update, create, drop, alter, etc.) could be framed inside an **execSQL(…)** method.

2. You *should make the call to execSQL inside of a try-catch-finally block*. Be aware of potential **SQLiteException** situations thrown by the method.

SQL Databases

Creating-Populating a Table

NOTE:
SQLITE uses an **invisible** field called ***ROWID*** to uniquely identify each row in each table.

Consequently in our example the field: *recID* and the database *ROWID* are functionally similar.

SQL Databases

Asking SQL Questions

1. *Retrieval queries* are *SQL-select* statements.
2. *Answers* produced by retrieval queries are always held in an *output table.*
3. In order to process the *resulting rows*, the user should provide a **cursor** device. Cursors allow a *row-by-row access* of the records returned by the retrieval queries.

Android offers two mechanisms for phrasing SQL-select statements: ***rawQueries*** and ***simple queries***. Both return a database *cursor*.

1. **Raw queries** take for input a syntactically correct SQL-select statement. The select query could be as complex as needed and involve any number of tables (remember that *outer joins* are not supported).
2. **Simple queries** are compact parametized select statements that operate on a single table (for developers who prefer not to use SQL).

SQL Databases

SQL Select Syntax (see http://www.sqlite.org/lang.html)

SQL-select statements are based on the following components

```
select   field1, field2, … , fieldn
  from   table1, table2,  … , tablen
```

```
 where   ( restriction-join-condition )
order by field n1, …, field nm
group by field m1, … , field mk
  having (group-condition)
```

The first two lines are mandatory, the rest is optional.

SQL Databases

SQL Select Syntax (see http://www.sqlite.org/lang.html)

Examples

```
select    LastName, cellPhone
  from    ClientTable
 where    state = 'Ohio'
order by  LastName
```

```
select    city, count(*) as TotalClients
  from    ClientTable
group by  city
```

SQL Databases

Example1. Using RawQuery (version 1)

Consider the following code fragment

```
Cursor c1 = db.rawQuery(
                "select count(*) as Total from tblAMIGO",
                null);
```

1. The previous *rawQuery* contains a select-statement that counts the rows in the table tblAMIGO.
2. The result of this count is held in a table having only one row and one column. The column is called "**Total**".
3. The cursor **c1** will be used to traverse the rows (one!) of the resulting table.
4. Fetching a row using cursor **c1** requires advancing to the next record in the answer set.
5. Later the (singleton) field **total** must be bound to a local Java variable.
   We soon will show how to do that.

SQL Databases

Example2. Using Parametized RawQuery (version 2)

Using arguments. Assume we want to count how many friends are there whose name is 'BBB' and their recID > 1. We could use the following construction

```
String mySQL = "select count(*) as Total "
             + " from tblAmigo "
             + " where recID > ? "
             + "   and name  = ? ";

   String[] args = {"1", "BBB"};

   Cursor c1 = db.rawQuery(mySQL, args);
```

Parameters

SQL Databases

Example2. Using Parametized RawQuery (version 2)

Using arguments.

After the substitutions are made the resulting SQL statement is:

```
select count(*) as Total
  from tblAmigo
 where recID > 1
   and name = 'BBB'
```

NOTE:

Partial matching using expressions such as: **name like '?%'** are not working now. Wait for an Android fix! (see similar issue: http://code.google.com/p/android/issues/detail?id=2619)
Similarly *String.format(…)* fails to properly work in cases such as: name like **'%s%'**.
note the second **%** is the SQL wild-character symbol, not an invalid string format!

SQL Databases

Example2. Using RawQuery (version 3)

Using arguments. Assume we want to count how many friends are there whose name is 'BBB' and their recID > 1. *We could concatenate pieces of the string. Special care around (single) quoted strings.*

```
String[] args = {"1", "BBB"};

String mySQL = " select count(*) as Total "
             + "  from tblAmigo "
             + " where recID > " + args[0]
             + "   and name  = '" + args[1] + "'";


Cursor c1 = db.rawQuery(mySQL, null);
```

SQL Databases

Simple Queries

Simple queries use a *template* implicitly representing a condensed version of a typical (non-joining) SQL select statement. *No explicit SQL statement is made.*

Simple queries can *only* retrieve data from a *single table*.

The method's signature has a fixed sequence of seven arguments representing:

1. the table name,
2. the columns to be retrieved,
3. the search condition (where-clause),
4. arguments for the where-clause,
5. the group-by clause,
6. having-clause, and
7. the order-by clause.

SQL Databases

Simple Queries

The signature of the Android's simple query method is:

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

SQL Databases

Simple Queries. Example 1

Query the *EmployeeTable*, find the average salary of female employees supervised by 123456789. Report results by *Dno.* List first the highest average, and so on, do not include depts. having less than two employees.

```
String[] columns =
             {"Dno", "Avg(Salary) as AVG"};

String[] conditionArgs =
             {"F", "123456789"};
Cursor c =  db.query(
            "EmployeeTable",
            columns,
            "sex = ? And superSsn = ? " ,
            conditionArgs,
            "Dno",
            "Count(*) > 2",
            "AVG Desc "
);
```

```
← table name
← columns
← condition
← condition args
← group by
← having
← order by
```

SQL Databases

Simple Queries. Example 2

The following query selects from each row of the *tblAMIGO* table the columns: *recID*, *name*, and *phone*. RecID must be greather than 2, and names must begin with 'B' and have three or more letters.

```
String [] columns = {"recID", "name", "phone"};

Cursor c1 = db.query (
      "tblAMIGO",
      columns,
      "recID > 2 and length(name) >= 3 and name like 'B%' ",
      null, null, null,
      "recID" );

int theTotal = c1.getCount();
```

SQL Databases

Simple Queries. Example 2 (cont.)

1. The String array *columns* holds the name of fields to be selected.
2. The retrieval condition is explicitly provided.
3. Several fields are missing in the call including: *selectionArgs, group-by,* and *having-clause*. Instead the **null** value is used to signal their absence.
4. The last argument indicates the result should be sorted on "*recID*" sequence.

```
 String [] columns = {"recID", "name", "phone"};

Cursor c1 = db.query (
       "tblAMIGO",
       columns,
       "recID > 2 and length(name) >= 3 and name like 'B%' ",
       null, null, null,
       "recID" );
```

SQL Databases

Simple Queries. Example 3

In this example we will construct a more complex SQL select statement.

We are interested in tallying how many groups of friends whose recID > 3 have the same name. In addition, we want to see 'name' groups having no more than four people each.

A possible SQL-select statement for this query would be something like:

```
select name, count(*) as TotalSubGroup
  from tblAMIGO
 where recID > 3
 group by name
having count(*) <= 4;
```

SQL Databases

Simple Queries. Example 3 (cont.)

An Android solution for the problem using a simple template query follows.

```
String [] selectColumns =  {"name", "count(*) as TotalSubGroup"};
String    whereCondition = "recID > ?";
String [] whereConditionArgs = {"3"};
String    groupBy = "name";
String    having =  "count(*) <= 4";
String    orderBy = "name";

Cursor myCur =  db.query (
                  "tblAMIGO",
                  selectColumns,
                  whereCondition, whereConditionArgs,
                  groupBy,
                  having,
                  null  );
```

SQL Databases

Simple Queries. Example 3 (cont.)

Observations

1. The *selectColumns* array indicates two fields *name* which is already part of the table, and *TotalSubGroup* which is to be computed as the count(*) of each name sub-group.

2. The symbol **?** in the *whereCondition* is a *place-marker* for a substitution. The value "**3**" taken from the *whereConditionArgs* is to be injected there.

3. The *groupBy* clause uses '*name*' as a key to create sub-groups of rows with the same *name* value. The *having* clause makes sure we only choose subgroups no larger than four people.

SQL Databases

Cursors

Android cursors are used to gain (random) access to tables produced by SQL *select* statements.

Cursors primarily provide *one row-at-the-time* operations on a table.
Cursors include several types of operator, among them:

1. **Positional awareness operators** (*isFirst(), isLast(), isBeforeFirst(), isAfterLast()*),
2. **Record Navigation** (*moveToFirst(), moveToLast(), moveToNext(), moveToPrevious(), move(n)*)
3. **Field extraction** (*getInt, getString, getFloat, getBlob, getDate, etc.*)
4. **Schema inspection** (getColumnName, getColumnNames, getColumnIndex, getColumnCount, getCount)

SQL Databases

Example 4. Cursors

1. The following example uses a cursor to handle the individual results of a SQL statement.
2. The select-command extracts from the tblAMIGO table the values indicated in the columns array, namely: *recID*, *name*, and *phone*.
3. The *getColumnIndex* method is called to determine the position of chosen columns in the current row.
4. The getters: *getInt*, *getString* commands are used for field extraction.
5. The *moveToNext* command forces the cursor to displace from its before-first position to the first available row.
6. The loop is executed until the cursor cannot be advanced any further.

SQL Databases

Example 4. Cursors

```
String [] columns ={"recID", "name", "phone"};

Cursor myCur = db.query("tblAMIGO", columns,
                        null, null, null, null, "recID");

int idCol = myCur.getColumnIndex("recID");
int nameCol = myCur.getColumnIndex("name");
int phoneCol = myCur.getColumnIndex("phone");

while (myCur.moveToNext()) {
    columns[0] = Integer.toString((myCur.getInt(idCol)));
    columns[1] = myCur.getString(nameCol);
    columns[2] = myCur.getString(phoneCol);

    txtMsg.append("\n" + columns[0] + " "
                       + columns[1] + " "
                       + columns[2] );
}
```

SQL Databases

SQL Action Queries

Action queries are the SQL way of performing maintenance operations on tables and database resources (i.e. *insert, delete, update, create table, drop, …*).

Examples:

```
insert into tblAmigos values ( 'Macarena',  '555-1234' );

update tblAmigos set name = 'Maria Macarena' where phone = '555-1234';

delete from tblAmigos where phone = '555-1234';

create table Temp ( column1 int, column2 text, column3 date );

drop table Temp;
```

SQL Databases

SQL Action Queries

1. Cursors provide **READ_ONLY** access to records.
2. Early versions of the Android SDK included cursor commands to sequentially modify records. Those operators have been deprecated in Release 1.0.
3. Methods such as *cursor.updateInt(...)* and *cursor.deleteRow(...)* are not valid anymore.
4. Instead use an action SQL command in an ***execSQL(...)*** method (explained in the next section).

SQL Databases

ExecSQL – Action Queries

Perhaps the simplest Android way to phrase a SQL action query is to 'stitch' together the pieces of the SQL statement and give it to the ***execSQL*(…)** method.

As an example consider the following case

```
db.execSQL(
      "update tblAMIGO set name = (name || 'XXX') where phone >= '001' ");
```

this statement appends 'XXX' to the name of those whose phone number is equal or greater than '001'.

SQL Databases

ExecSQL – Action Queries (cont.)

Consider the action query:

```
db.execSQL(
      "update tblAMIGO set name = (name || 'XXX') where phone >= '001' ");
```

Alternatively, the SQL statement could be 'pasted' from pieces as follows:

```
String theValue  =  " …";  //some phone value goes here

db.execSQL( "update tblAMIGO set name = (name || 'XXX') " +
                  " where phone >= '" + theValue + "' " );
```

The same strategy could be applied to other SQL statements such as: "*delete from … where…*", "insert into ….", etc.

SQL Databases

Other Android Solutions for Table Maintenance

Although they are not as general as the technique suggested in the previous section, Android provides a number of additional methods to perform *insert, delete, update* operations.

```
public long insert(String table,
                   String nullColumnHack,
                   ContentValues values )

public int update( String table,
                   ContentValues values,
                   String whereClause, String[] whereArgs )

public int delete( String table,
                   String whereClause, String[] whereArgs)
```

SQL Databases

Database insert Operator

public long insert(String table, String nullColumnHack, ContentValues values)

Convenient method for inserting a row into the database.

Parameters

table	the table to insert the row into
nullColumnHack	SQL doesn't allow inserting a completely empty row, so if argument *values is empty this column* will explicitly be assigned a NULL value.
values	this map (*name, value*) contains the initial column values for the row. The keys should be the column names and the values the column values
Returns	the row ID of the newly inserted row, or -1 if an error occurred

SQL Databases

Example – Database insert Operator

```
1. ContentValues initialValues = new ContentValues();

2. initialValues.put("name", "ABC");
3. initialValues.put("phone", "101");
4. int rowPosition = (int) db.insert("tblAMIGO", null, initialValues);

5. initialValues.put("name", "DEF");
6. initialValues.put("phone", "202");
7. rowPosition = (int) db.insert("tblAMIGO", null, initialValues);

8. initialValues.clear();
9. rowPosition = (int) db.insert("tblAMIGO", null, initialValues);
10.rowPosition = (int) db.insert("tblAMIGO", "name", initialValues);
```

SQL Databases

Example – Database insert Operator – Comments

- **Lines 1-3** define the set of <key, values> called *initialValues to be later inserted in a record of the form <recID, name, phone> . Remember that recID is an autoincremented field. All this work is done to pre-assemble the record < ???, "ABCC", "101">. Here ??? will be the recID field to be determined by the database when the record is accepted.*
- **Line 4** requests the set of <key, values> held in *initialValues to be added to the table tblAMIGO. If the operation fails the insert method returns -1, otherwise the position of the row identifier is returned.*
- **Lines 5-7** define a new set of values to be used as input to the insert operator. The record <???, "DEF", "202"> is placed after the row previously inserted in table tblAMIGO.
- **Line 9** resets the map to empty.
- **Line 10** attempts the insertion of an empty record. SQL rejects the operation and returns -1
- **Line 11** is similar to the code in Line 10, however the presence of a *nullColumnHack variable ("name" in this case) makes SQL change its behavior; the row is generated with null values everywhere except the key autonumber (recID).*

SQL Databases

Database update Operator

public int update (String table,
ContentValues values,
String whereClause, String[] whereArgs)

Convenient method for updating rows in the database.

Parameters

table	the table to update in
values	a map <name,value> from column names to new column values. null is a valid value that will be translated to NULL.
whereClause	the optional WHERE clause to apply when updating. Passing null will update all rows.

Returns *the number of rows affected*

SQL Databases

Example - Database update Operator

We want to use the "update" method to express the SQL statement:

Update tblAmigo set name = 'maria' where (recID > 2 and recID < 7)

Here are the steps to make the call using Android Update Method

```
1. String [] whereArgs = {"2", "7"};

2. ContentValues updValues = new ContentValues();
3. updValues.put("name", "Maria");

4. int recAffected =  db.update( "tblAMIGO",
                      updValues,
                      "recID > ? and recID < ?",
                      whereArgs );
```

SQL Databases

Example /Comments - Database update Operator

Line 1 defines the String array holding the (two) arguments used by the *whereClause*.

Lines 2-3 define and populate a map <*key, value*> to be used by the update operator. The map expresses the idea "*set given column to given value*". In our case the "name" field will acquire the value "maria".

Line 4 invokes the execution of the update operator. After completion it returns the number of records affected by the update (0 If it fails).

In the example a **filter** is given to select the rows to be updated. In this case the condition is "*recID > ? and recID < ?*". The **?** symbols represent placeholders for values supplied by the array *whereArgs.* After the substitutions are made the new filter is: "*recID > 2 and recID < 7*".

SQL Databases

Database delete Operator

public int **delete** (String table, String whereClause, String[] whereArgs)

Convenient method for deleting rows in the database.

Parameters

table	the table to delete from
whereClause	the optional WHERE clause to apply when deleting. Passing null will delete all rows.

Returns	the number of rows affected if a *whereClause* is passed in, 0 otherwise. *To remove all rows and get a count pass "1" as the whereClause.*

SQL Databases

Example - Database delete Operator

Consider the following SQL statement:

Delete from tblAmigo wehere recID > 2 and recID < 7

An equivalent version using the **delete method** follows:

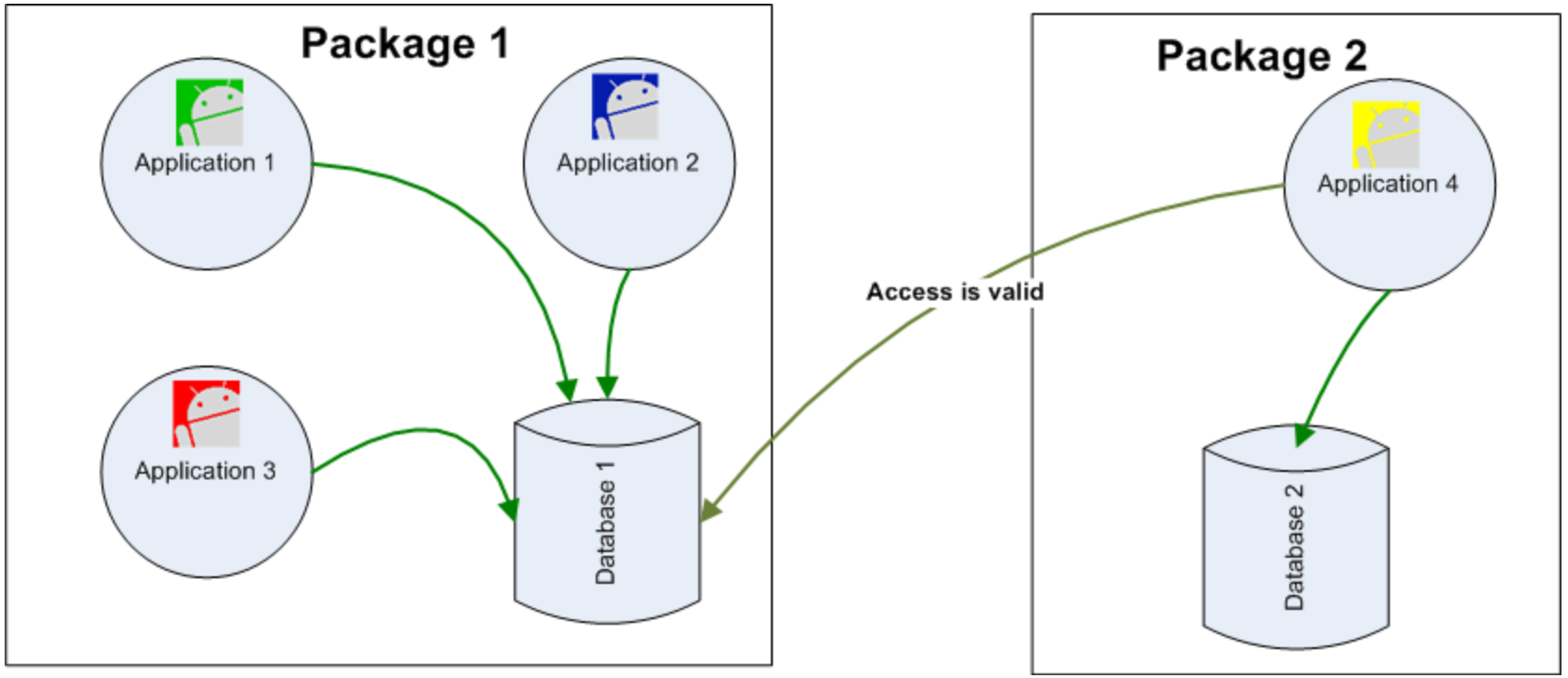
```
1.  String [] whereArgs = {"2", "7"};

2.  recAffected = db.delete("tblAMIGO",
                    "recID > ? and recID < ?",
                    whereArgs);
```

Line 2 requests the deleting from the tblAMIGO of those records whose recID is in between the values 2, and 7. The actual values are taken from the *whereArgs* array shown in **Line 1**. The method returns the number of rows deleted after executing the command.

SQL Databases

Database Visibility

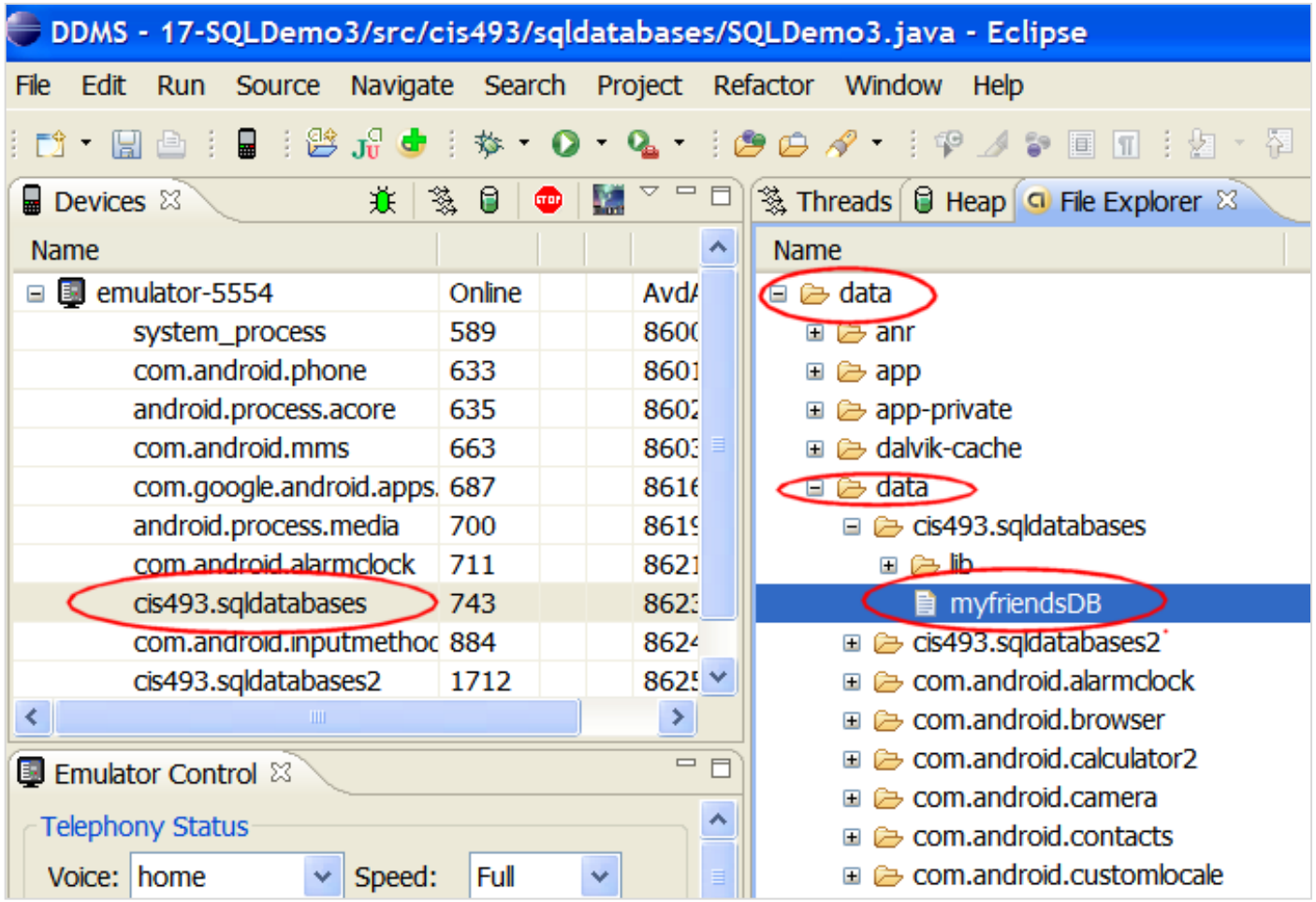
Any Application can access an SD stored database. All it's needed is knowledge of the path where the database file is located. *Other (ContentProvider) ways of sharing will be explored later.*



SQL Databases

Database Location

Emulator's *File Explorer* showing the placement of the database



SQL Databases

Using SQLITE Command Line

The Android SDK contains a command line interface to SQLITE databases.
To open/Create a database use the command

```
C:> sqlite3 myNewDatabase
```

You may directly reach the Emulator's data folder and operate on existing databases.

Assume an emulator is running.

We will use **adb shell** to tap in the emulator's internal memory

SQL Databases

Using SQLITE Command Line

After opening the DOS command window type the following commands:

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

E:\Android> adb shell

# sqlite3 /data/data/matos.sql1/databases/myfriendsDB

sqlite3 /data/data/matos.sql1/databases/myfriendsDB
SQLite version 3.5.9
Enter ".help" for instructions
```

SQL Databases

Using SQLITE Command Line

After opening the DOS command window type the following commands:

```
sqlite> .tables
.tables
android_metadata  tblAMIGO

sqlite> select * from tblAMIGO;

1|AAAXXX|555
2|BBBXXX|777
3|Maria|999
4|Maria|000
5|Maria|001

sqlite> .exit
#
```

SQL Databases

Summary of SQLITE3 commands

sqlite3> **.help**

.bail ON|OFF	Stop after hitting an error. Default OFF
.databases	List names and files of attached databases
.dump ?TABLE? ...	Dump the database in an SQL text format
.echo ON|OFF	Turn command echo on or off
.exit	Exit this program
.explain ON|OFF	Turn output mode suitable for EXPLAIN on or off.
.header(s) ON|OFF	Turn display of headers on or off
.help	Show this message
.import FILE TABLE	Import data from FILE into TABLE
.indices TABLE	Show names of all indices on TABLE
.load FILE ?ENTRY?	Load an extension library

SQL Databases

Summary of SQLITE3 commands

.mode MODE ?TABLE?		Set output mode where MODE is one of:
	csv	Comma-separated values
	column	Left-aligned columns. (See .width)
	html	HTML <table> code
	insert	SQL insert statements for TABLE
	line	One value per line
	list	Values delimited by .separator string
	tabs	Tab-separated values
	tcl	TCL list elements
.nullvalue STRING		Print STRING in place of NULL values
.output FILENAME		Send output to FILENAME
.output stdout		Send output to the screen
.prompt MAIN CONTINUE		Replace the standard prompts

SQL Databases

Summary of SQLITE3 commands

.quit	Exit this program
.read FILENAME	Execute SQL in FILENAME
.schema ?TABLE?	Show the CREATE statements
.separator STRING	Change separator used by output mode and .import
.show	Show the current values for various settings
.tables ?PATTERN?	List names of tables matching a LIKE pattern
.timeout MS	Try opening locked tables for MS milliseconds
.width NUM NUM ...	Set column widths for "column" mode

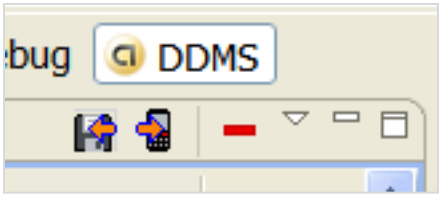
SQL Databases

Using GUI Tools for SQLITE

In order to move a copy of the database in and out of the Emulator's storage space and either receive or send the file into/from the local computer's file system you may use the commands:

adb pull <*full_path_to_database*> and
adb push <*full_path_to_database*>.

You may also use the Eclipse's *DDMS Perspective* to push/pull files in/out the emulator's file system.



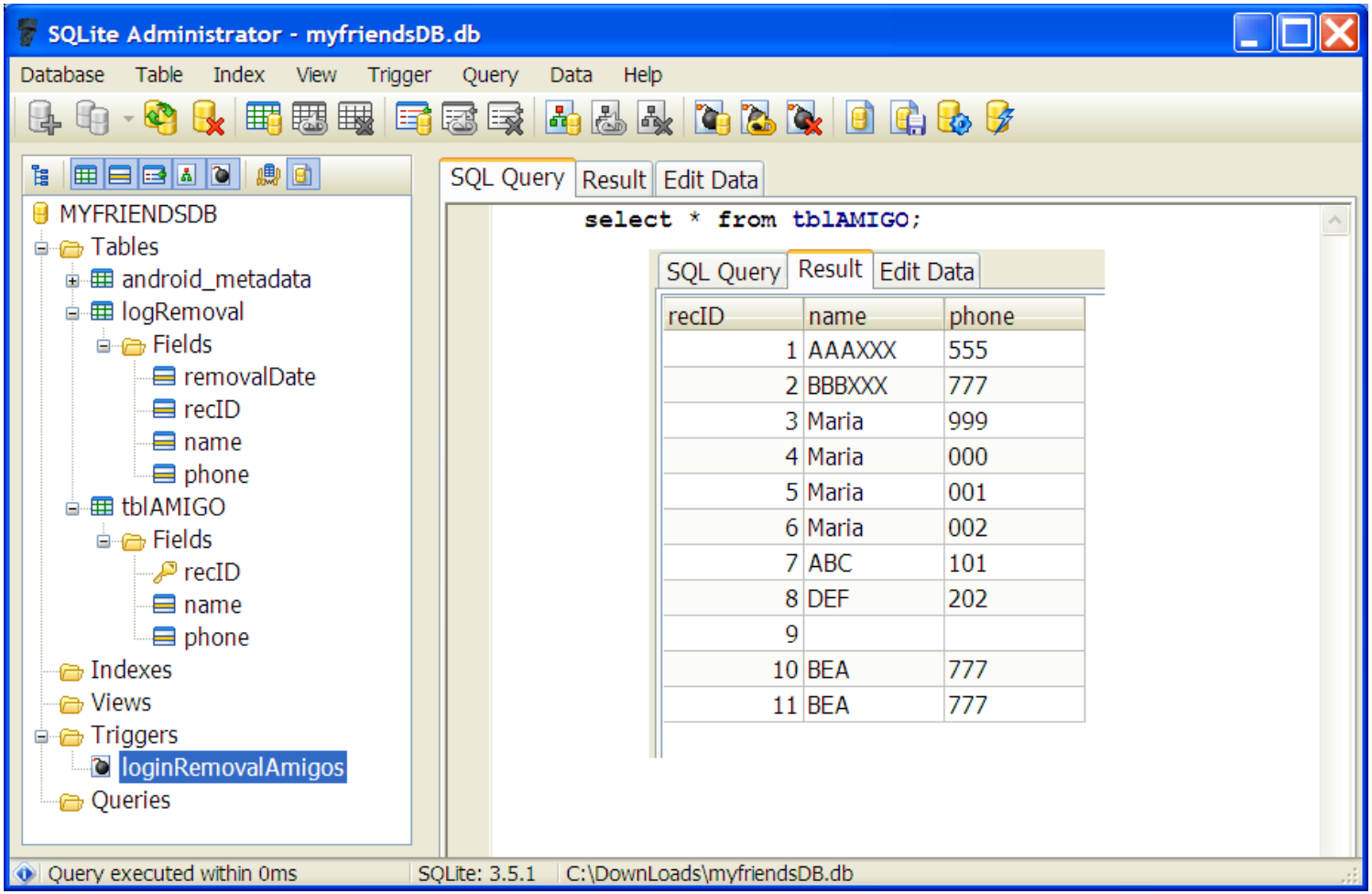
Once the database is in your computer's disk you may manipulate the database using a 'user-friendly' tool such as:

- **SQLite Manager** (Firefox adds-on)
- **SQLite Administrator** (http://sqliteadmin.orbmu2k.de)

In this example we use SQLite Administrator.

SQL Databases

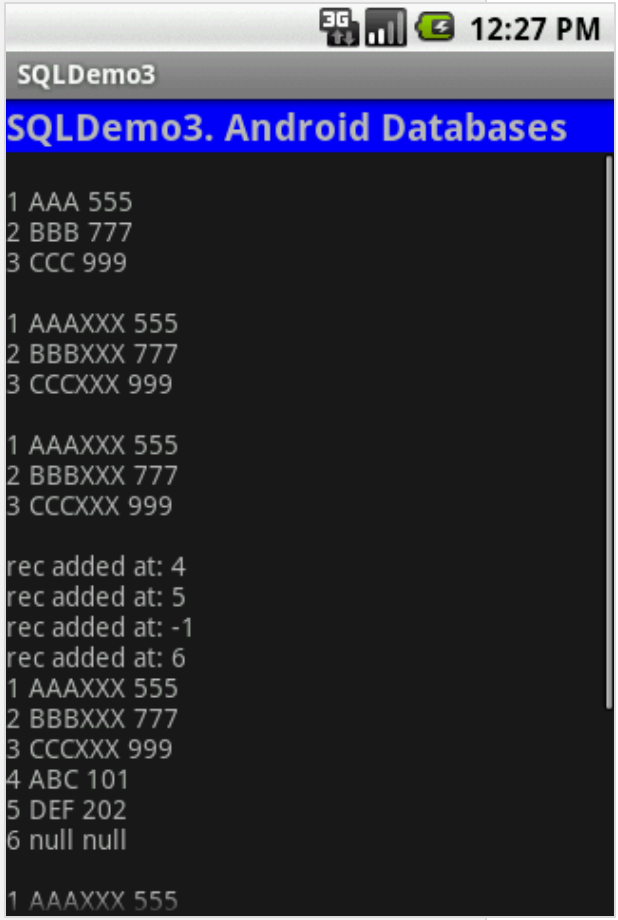
Using *SQLite Administrator*



SQL Databases

Example: Complete Listing for Previous Fragments

```
<?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"
   >
        <TextView
        android:id="@+id/txtCaption"
        android:text="SQLDemo3. Android Databases"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:background="#ff0000ff"
        android:textSize="20px"
        android:textStyle="bold"/>

        <ScrollView
        android:id="@+id/ScrollView01"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
                <TextView
                android:id="@+id/txtMsg"
                android:text=""
                android:layout_width="fill_parent"
                android:layout_height="wrap_content" />
        </ScrollView>
</LinearLayout>
```



SQL Databases

Example: Complete Listing for Previous Fragments

```
//USING ANDROID-SQLITE DATABASES
package cis493.sqldatabases;

import android.app.Activity;
import android.content.ContentValues;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteException;
import android.os.Bundle;
import android.widget.TextView;
import android.widget.Toast;

public class SQLDemo3 extends Activity {
SQLiteDatabase db;
TextView txtMsg;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        txtMsg = (TextView) findViewById(R.id.txtMsg);
```

SQL Databases

Example: Complete Listing for Previous Fragments

```
    try {
        openDatabase();     //open (create if needed) database
        dropTable();        //if needed drop table tblAmigos
        insertSomeDbData(); //create-populate tblAmigos
        useRawQuery1();     //fixed SQL with no arguments
        useRawQuery2();     //parameter substitution
        useRawQuery3();     //manual string concatenation
        useSimpleQuery1();  //simple query
        useSimpleQuery2();  //nontrivial 'simple query'
        useCursor1();       //retrieve rows from a table
        updateDB();         //use execSQL to update
        useInsertMethod();  //use insert method
        useUpdateMethod();  //use update method
        useDeleteMethod();  //use delete method

        db.close();//make sure to release the DB
        Toast.makeText(this,"All done!",1).show();
    } catch (Exception e) {
        Toast.makeText(this,e.getMessage(),1).show();
    }
}// onCreate
```

SQL Databases

Example: Complete Listing for Previous Fragments [**openDatabase**]

```
private void openDatabase() {
        try {
            db = SQLiteDatabase.openDatabase(
                    "data/data/cis493.sqldatabases/myfriendsDB",
                    //"sdcard/myfriendsDB",
                    null,
                    SQLiteDatabase.CREATE_IF_NECESSARY) ;

            Toast.makeText(this, "DB was opened!", 1).show();
        }
        catch (SQLiteException e) {
            Toast.makeText(this, e.getMessage(), 1).show();
        }
    }//createDatabase
```

SQL Databases

Example: Complete Listing for Previous Fragments [**insertSomeDbData**]

```
private void insertSomeDbData() {
    //create table: tblAmigo
    db.beginTransaction();
   try {
        db.execSQL("create table tblAMIGO ("
                  + " recID integer PRIMARY KEY autoincrement, "
                  + " name  text, "
                  + " phone text );  ");
        //commit your changes
        db.setTransactionSuccessful();

        Toast.makeText(this, "Table was created",1).show();
   } catch (SQLException e1) {
        Toast.makeText(this, e1.getMessage(),1).show();
   }
   finally {
        //finish transaction processing
        db.endTransaction();
   }
```

SQL Databases

Example: Complete Listing for Previous Fragments [**insertSomeDbData**]

```
   // populate table: tblAmigo
    db.beginTransaction();
    try {
        //insert rows
       db.execSQL( "insert into tblAMIGO(name, phone) "
                  + " values ('AAA', '555' );" );
      db.execSQL("insert into tblAMIGO(name, phone) "
                  + " values ('BBB', '777' );" );
      db.execSQL("insert into tblAMIGO(name, phone) "
                  + " values ('CCC', '999' );" );

      //commit your changes
      db.setTransactionSuccessful();
      Toast.makeText(this, " 3 records were inserted",1).show();
      }
     catch (SQLiteException e2) {
      //report problem
      Toast.makeText(this, e2.getMessage(),1).show();
     }
     finally {
      db.endTransaction();
     }
}//insertSomeData
```

SQL Databases

Example: Complete Listing for Previous Fragments [**useRawQuery1**]

```
private void useRawQuery1() {
    try {
        //hard-coded SQL-select command with no arguments
        String mySQL ="select count(*) as Total from tblAMIGO";

        Cursor c1 = db.rawQuery(mySQL, null);

        int index = c1.getColumnIndex("Total");
        //advance to the next record (first rec. if necessary)
        c1.moveToNext();
        int theTotal = c1.getInt(index);
        Toast.makeText(this, "Total1: " + theTotal, 1).show();

    } catch (Exception e) {
        Toast.makeText(this, e.getMessage(), 1).show();
    }
}//useRawQuery1
```

SQL Databases

Example: Complete Listing for Previous Fragments [**useRawQuery2**]

```
private void useRawQuery2() {
    try {
        // ? arguments provided for automatic replacement
        String mySQL = " select count(*) as Total "
                     + " from tblAmigo "
                     + " where recID > ? "
                     + "   and name  = ? ";
        String[] args = {"1", "BBB"};
        Cursor c1 = db.rawQuery(mySQL, args);

        int index = c1.getColumnIndex("Total");
        //advance to the next record (first rec. if necessary)
        c1.moveToNext();
        int theTotal = c1.getInt(index);
        Toast.makeText(this, "Total2: " + theTotal, 1).show();
    } catch (Exception e) {
        Toast.makeText(this, e.getMessage(), 1).show();
  }
}//useRawQuery2
```

SQL Databases

Example: Complete Listing for Previous Fragments [**useRawQuery3**]

```
private void useRawQuery3() {
    try {
        //arguments injected by manual string concatenation
        String[] args = {"1", "BBB"};

        String mySQL = " select count(*) as Total "
            + "  from tblAmigo "
              + " where recID > "  + args[0]
              + "   and name  = '" + args[1] + "'";

        Cursor c1 = db.rawQuery(mySQL, null);
        int index = c1.getColumnIndex("Total");
        //advance to the next record (first rec. if necessary)
        c1.moveToNext();
        int theTotal = c1.getInt(index);
        Toast.makeText(this, "Total3: " + theTotal, 1).show();
    } catch (Exception e) {
        Toast.makeText(this, e.getMessage(), 1).show();
    }
}//useRawQuery3
```

SQL Databases

Example: Complete Listing for Previous Fragments [**simpleQuery1**]

```
  private void useSimpleQuery1() {
    try {
        //simple (implicit) query on one table
        String [] columns = {"recID", "name", "phone"};

        Cursor c1 = db.query (
            "tblAMIGO",
            columns,
            "recID > 2 and length(name) >= 3 and name like 'B%' ",
            null, null, null,
            "recID" );

        int theTotal = c1.getCount();
        Toast.makeText(this, "Total4: " + theTotal, 1).show();
     } catch (Exception e) {
        Toast.makeText(this, e.getMessage(), 1).show();
     }
}//useSimpleQuery1
```

SQL Databases

Example: Complete Listing for Previous Fragments [**simpleQuery2**]

```
private void useSimpleQuery2() {
    try {
        //nontrivial 'simple query' on one table
        String [] selectColumns =  {"name", "count(*) as TotalSubGroup"};
        String    whereCondition = "recID >= ?";
        String [] whereConditionArgs = {"1"};
        String    groupBy = "name";
        String    having =  "count(*) <= 4";
        String    orderBy = "name";

        Cursor c =  db.query ( "tblAMIGO",
                               selectColumns,
                               whereCondition, whereConditionArgs,
                               groupBy,
                               having,
                               orderBy  );

        int theTotal = c.getCount();
        Toast.makeText(this, "Total5: " + theTotal, 1).show();
     } catch (Exception e) {
        Toast.makeText(this, e.getMessage(), 1).show();
     }
}//useSimpleQuery2
```

SQL Databases

Example: Complete Listing for Previous Fragments [**useCursor1**]

```
private void useCursor1() {
   try {
        txtMsg.append("\n");
        // obtain a list of <recId, name, phone> from DB
        String[] columns = { "recID", "name", "phone" };
        Cursor c = db.query("tblAMIGO", columns,
                            null, null, null, null, "recID");
        int theTotal = c.getCount();
        Toast.makeText(this, "Total6: " + theTotal, 1).show();
        int idCol = c.getColumnIndex("recID");
        int nameCol = c.getColumnIndex("name");
        int phoneCol = c.getColumnIndex("phone");
        while (c.moveToNext()) {
             columns[0] = Integer.toString((c.getInt(idCol)));
             columns[1] = c.getString(nameCol);
             columns[2] = c.getString(phoneCol);
             txtMsg.append( columns[0] + " " + columns[1] + " "
                          + columns[2] + "\n" );
        }
        } catch (Exception e) {
          Toast.makeText(this, e.getMessage(), 1).show();
        }
   }//useCursor1
```

SQL Databases

Example: Complete Listing for Previous Fragments [**updateDB**]

```
private void updateDB(){
    //action query using execSQL
    String theValue;
    try {
        theValue = "222";

        db.execSQL(  " update tblAMIGO "
                   + " set name = (name || 'XXX') "
                   + " where phone >= '" + theValue + "' " );

        useCursor1();
    } catch (Exception e) {
      Toast.makeText(this,"updateDB " + e.getMessage(), 1).show();
    }
    useCursor1();
}
```

SQL Databases

Example: Complete Listing for Previous Fragments [**dropTable**]

```
private void dropTable(){
  //(clean start) action query to drop table

  try {
       db.execSQL( " drop table tblAmigo; ");

       Toast.makeText(this, "Table dropped", 1).show();

  } catch (Exception e) {
       Toast.makeText(this,
                      "dropTable()\n" + e.getMessage(), 1).show();
  }
}// dropTable
```

SQL Databases

Example: Complete Listing for Previous Fragments [**useInsertMethod**]

```
public void useInsertMethod() {
    ContentValues initialValues = new ContentValues();

    initialValues.put("name", "ABC");
    initialValues.put("phone", "101");
    int rowPosition = (int) db.insert("tblAMIGO", null, initialValues);
    txtMsg.append("\nrec added at: " + rowPosition);

    initialValues.put("name", "DEF");
    initialValues.put("phone", "202");
    rowPosition = (int) db.insert("tblAMIGO", null, initialValues);
    txtMsg.append("\nrec added at: " + rowPosition);

    initialValues.clear();
    rowPosition = (int) db.insert("tblAMIGO", null, initialValues);
    txtMsg.append("\nrec added at: " + rowPosition);
    rowPosition = (int) db.insert("tblAMIGO", "name", initialValues);
    txtMsg.append("\nrec added at: " + rowPosition);

    useCursor1();

}// useInsertMethod
```

SQL Databases

Example: Complete Listing for Previous Fragments [**useUpdateMethod**]

```
private void useUpdateMethod() {
    //using the update method to change name of selected friend
    String [] whereArgs = {"2", "7"};

    ContentValues updValues = new ContentValues();
    updValues.put("name", "Maria");

    int recAffected =db.update("tblAMIGO",
                               updValues,
                               "recID > ? and recID < ?",
                               whereArgs );

    Toast.makeText(this, "Total7: " + recAffected, 1).show();
    useCursor1();
}
```

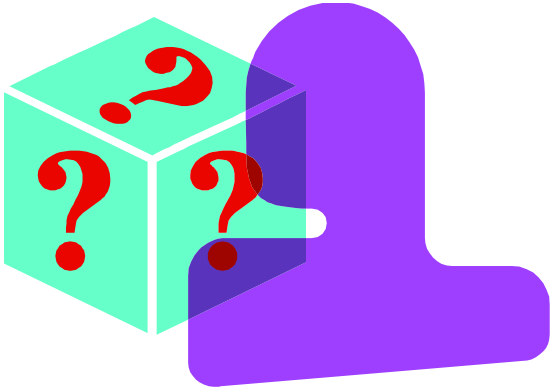
SQL Databases

Example: Complete Listing for Previous Fragments [**useDeleteMethod**]

```
    private void useDeleteMethod() {
        //using the delete method to remove a group of friends
        //whose id# is between 2 and 7
        String [] whereArgs = {"2", "7"};

        int recAffected = db.delete("tblAMIGO",
                                    "recID > ? and recID < ?",
                                    whereArgs);

        Toast.makeText(this, "Total8: " + recAffected, 1).show();
        useCursor1();

    }// useDeleteMethod

}//class
```

SQL Databases

Questions



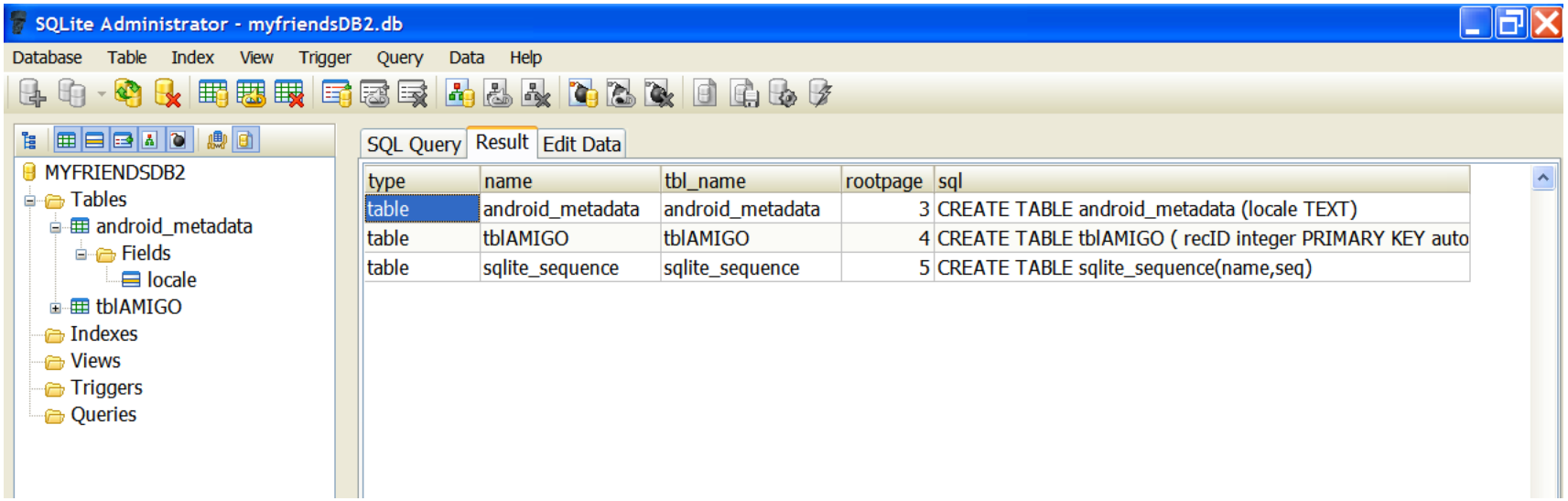
SQL Databases

Appendix 1: Database Dictionary - SQLITE Master Table

You may query the SQLITE master table (named: ***sqlite_master***) looking for a table, index, or other database object.

Example

```
select * from sqlite_master;
```



SQL Databases

Appendix 1: Database Dictionary - SQLITE Master Table

In Java code you may phrase the test for existence of a database object using something similar to the following fragment

```
public boolean tableExists(SQLiteDatabase db, String tableName)
{
   //true if table exists, false otherwise
   String mySql = " SELECT name FROM sqlite_master "
                + " WHERE type='table'                "
                + "   AND name='" + tableName  + "'";

   int resultSize = db.rawQuery(mySql, null).getCount();

   if (resultSize ==0) {
        return true;
   } else
        return false;
}
```

SQL Databases

Appendix 2: Convenient Database Command

In Java code you may phrase the request for “CREATE or REPLACE” a table using the following safe construct:

```
db.execSQL("DROP TABLE IF EXISTS tableXYZ");
```