

MOBILE DEVELOPMENT

SQL-DATABASE

A solid green horizontal bar spanning the width of the slide at the bottom.

CONTENTS

USING SQL DATABASES IN ANDROID

TYPE OF SQL COMMANDS

DATABASE VISIBILITY

APPENDIX

USING SQL DATABASES IN ANDROID

Included into the core Android architecture there is a standalone Database Management System (DBMS) called SQLite 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.

USING SQL DATABASES IN ANDROID

(Characteristics of SQLite)

Transactional SQL database engine.

Small footprint (less than 400KBytes)

Typeless

Serverless

Zero-configuration

The source code for SQLite is in the public domain.

According to their website, SQLite is the most widely deployed SQL database engine in the world

.

USING SQL DATABASES IN ANDROID

(Characteristics of SQLite)

1. SQLite implements most of the SQL-92 standard for SQL.
2. It has partial support for triggers and allows complex queries (exceptions include: right/full outer joins, grant/revoke, updatable views).
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. There is no data type checking, therefore it is possible to insert a string into numeric column and so on.

USING SQL DATABASES IN ANDROID

(Creating SQLite database - method 1)

If the database does not exist, then create a new one. Otherwise, open the existing database according to the flags: `OPEN_READWRITE`, `OPEN_READONLY`, `CREATE_IF_NECESSARY`.

```
SQLiteDatabase.openDatabase(myDbPath, null, SQLiteDatabase.CREATE_IF_NECESSARY);
```

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** `SQLException` if the database cannot be opened

USING SQL DATABASES IN ANDROID

(Creating SQLite database - method 1)

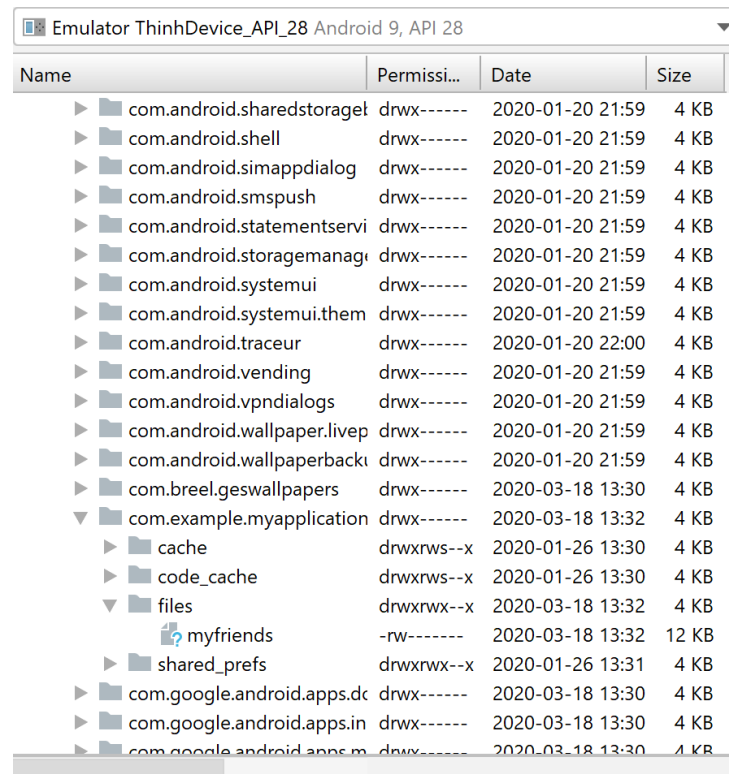
```
public class MainActivity extends Activity {
    SQLiteDatabase db;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState); setContentView(R.layout.activity_main);
        TextView txtMsg = (TextView) findViewById(R.id.txtMsg);
        // path to the external SD card (something like: /storage/sdcard/...)
        // String storagePath = Environment.getExternalStorageDirectory().getPath();
        // path to internal memory file system (data/data/cis470.matos.databases)
        File storagePath = getApplication().getFilesDir();
        String myDbPath = storagePath + "/" + "myfriends";
        txtMsg.setText("DB Path: " + myDbPath);
        try {
            db = SQLiteDatabase.openDatabase(myDbPath, null, SQLiteDatabase.CREATE_IF_NECESSARY);
            // here you do something with your database ...
            db.close();
            txtMsg.append("\nAll done!");
        }
        catch (SQLException e) { txtMsg.append("\nERROR " + e.getMessage()); }
    }
    // onCreate
}
// class
```

USING SQL DATABASES IN ANDROID

(Creating SQLite database - method 1)

SQLite Database is stored using **Internal Memory**

Path: /data/data/com.example.myapplication/, where: com.example.myapplication is package's name



Name	Permissi...	Date	Size
com.android.sharedstorage	drwx-----	2020-01-20 21:59	4 KB
com.android.shell	drwx-----	2020-01-20 21:59	4 KB
com.android.simappdialog	drwx-----	2020-01-20 21:59	4 KB
com.android.smspush	drwx-----	2020-01-20 21:59	4 KB
com.android.statementservi	drwx-----	2020-01-20 21:59	4 KB
com.android.storagemanage	drwx-----	2020-01-20 21:59	4 KB
com.android.systemui	drwx-----	2020-01-20 21:59	4 KB
com.android.systemui.them	drwx-----	2020-01-20 21:59	4 KB
com.android.traceur	drwx-----	2020-01-20 22:00	4 KB
com.android.vending	drwx-----	2020-01-20 21:59	4 KB
com.android.vpndialogs	drwx-----	2020-01-20 21:59	4 KB
com.android.wallpaper.livep	drwx-----	2020-01-20 21:59	4 KB
com.android.wallpaperbacku	drwx-----	2020-01-20 21:59	4 KB
com.breel.geswallpapers	drwx-----	2020-03-18 13:30	4 KB
com.example.myapplication	drwx-----	2020-03-18 13:32	4 KB
cache	drwxrws--x	2020-01-26 13:30	4 KB
code_cache	drwxrws--x	2020-01-26 13:30	4 KB
files	drwxrwx--x	2020-03-18 13:32	4 KB
myfriends	-rw-----	2020-03-18 13:32	12 KB
shared_prefs	drwxrwx--x	2020-01-26 13:31	4 KB
com.google.android.apps.dc	drwx-----	2020-03-18 13:30	4 KB
com.google.android.apps.in	drwx-----	2020-03-18 13:30	4 KB
com.google.android.apps.m	drwx-----	2020-03-18 13:30	4 KB

USING SQL DATABASES IN ANDROID

(Creating SQLite database - method 1)

SQLite Database is stored using **SD card**

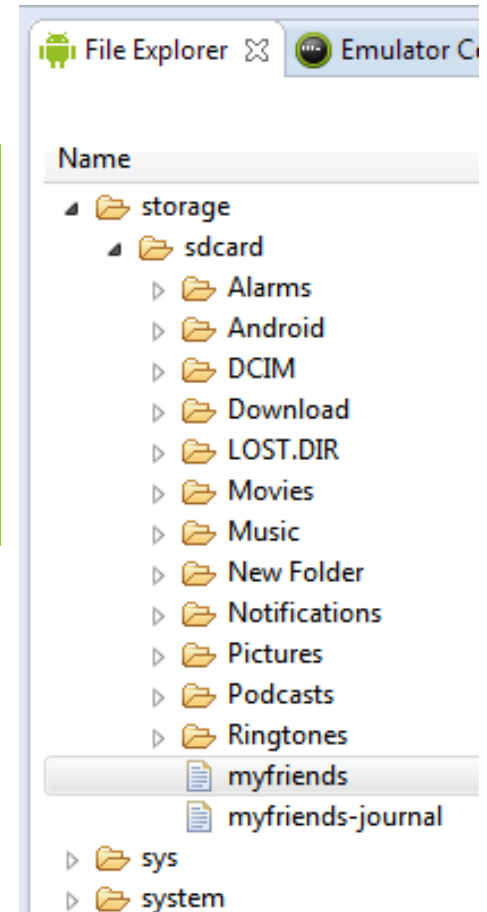
Using:

```
SQLiteDatabase db;
```

```
String SDcardPath = Environment.getExternalStorageDirectory().getPath() + "/myfriends";  
db = SQLiteDatabase.openDatabase(SDcardPath,null, SQLiteDatabase.CREATE_IF_NECESSARY);
```

Manifest must include:

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



USING SQL DATABASES IN ANDROID

(Sharing limitations)

Warning

- Databases created in the internal /data/data/package space are private to that package.
- You cannot access internal databases belonging to other people (instead use Content Providers or external SD resident DBs).
- SD stored databases are public.
- Access to an SD resident database requires the Manifest to include permissions:

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

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

USING SQL DATABASES IN ANDROID

(An alternative method: openOrCreateDatabase)

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

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

Assume this app is made in a namespace called com.example.myapplication, then the full name of the newly created database file will be:


/data/data/com.example.myapplication/myfriendsDB
Internal Memory Package name DB name

- The file can be accessed by all components of the same application.
- Other MODE values: MODE_WORLD_READABLE, and MODE_WORLD_WRITEABLE were deprecated on API Level 17.
- null refers to optional factory class parameter (skip for now)

TYPE OF SQL COMMANDS

(Transaction processing)

Once created, the SQLite 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.

Transactions are desirable because they help maintaining consistent data and prevent unwanted data 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 ‘statement’.

TYPE OF SQL COMMANDS

(Transaction processing)

The typical Android's way of running transactions on a SQLiteDatabase is illustrated by the following code fragment (Assume db is a SQLiteDatabase)

```
db.beginTransaction();
try { //perform your database operations here ...
    db.setTransactionSuccessful(); //commit your changes
}
catch (SQLException 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 operation; consequently the database is reset to the state previous to the beginning of the transaction

TYPE OF SQL COMMANDS

(Create and populate a SQL table)

recID	name	phone
1	AAA	555-1111
2	BBB	555-2222
3	CCC	555-3333

The SQL syntax used for creating and populating a table is illustrated in the following examples

```
create table tblAMIGO (  
    recID integer PRIMARY KEY autoincrement,  
    name text,  
    phone text);  
insert into tblAMIGO(name, phone) values ('AAA', '555-1111');
```

The autoincrement value for recID is NOT supplied in the insert statement as it is internally assigned by the DBMS.

TYPE OF SQL COMMANDS

(Example 2: create and populate a SQL table)

Our Android app will use the `execSQL(...)` method to manipulate SQL action queries. The example below 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 again.

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', '555-1111');");
db.execSQL("insert into tblAMIGO(name, phone) values ('BBB', '555-2222');");
db.execSQL("insert into tblAMIGO(name, phone) values ('CCC', '555-3333');");
```

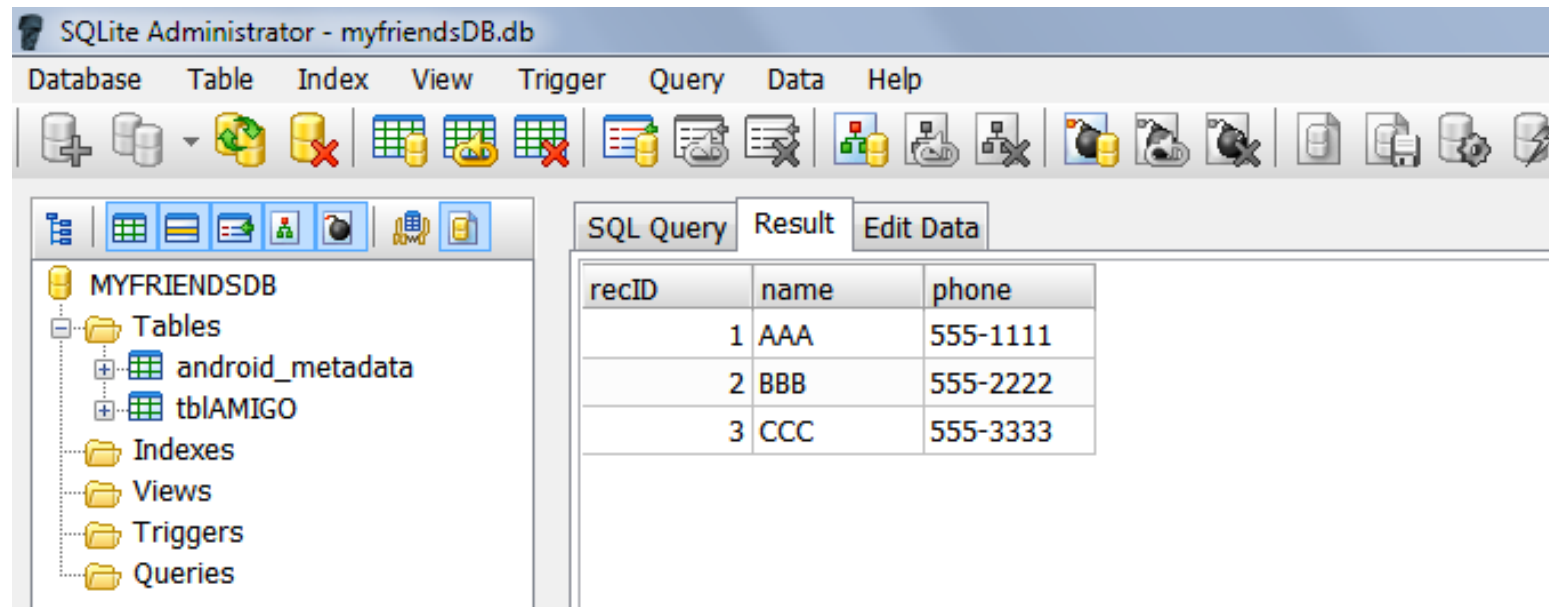
TYPE OF SQL COMMANDS

(Example 2: create and populate a SQL table)

After executing the previous code snippet, we transferred the database to the developer's file system and used the SQL-ADMINISTRATION tool.

There we submitted the SQL-Query: `select * from tblAmigo.`

Results are shown below.



TYPE OF SQL COMMANDS

(Example 2: create and populate a SQL table)

Comments

1. The field recID is defined as the table's PRIMARY KEY.
2. The “autoincrement” feature guarantees that each new record will be given a unique serial number (0,1,2,...).
3. On par with other SQL systems, SQLite offers the data types: **text, varchar, integer, float, numeric, date, time, timestamp, blob, boolean**
3. In general any well-formed DML SQL action command (insert, delete, update, create, drop, alter, etc.) could be framed inside an `execSQL(. . .)` method call.

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

Caution:

- You should call the `execSQL` method inside of a try-catch-finally block. Be aware of potential `SQLException` conflicts thrown by the method.

TYPE OF SQL COMMANDS

(Example 2: create and populate a SQL 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.

RowID	User defined fields		
0			
1			
2			
3			
4			
5			

TYPE OF SQL COMMANDS

(Asking questions - SQL queries)

1. Retrieval queries are known as SQL-select statements.
2. Answers produced by retrieval queries are always held in a table.
3. In order to process the resulting table rows, the user should provide a cursor device. Cursors allow a row-at-the-time access mechanism on SQL tables.

Android-SQLite offers two strategies for phrasing select statements: rawQueries and simple queries. Both return a database cursor.

1. Raw queries take for input any (syntactically correct) SQL-select statement. The select query could be as complex as needed and involve any number of tables (only a few exceptions such as outer-joins)
2. Simple queries are compact parametrized lookup functions that operate on a single table (for developers who prefer not to use SQL).

TYPE OF SQL COMMANDS

(SQL select statement - syntax)

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

- 1. The select clause indicates the fields to be included in the answer
- 2. The from clause lists the tables used in obtaining the answer
- 3. The where component states the conditions that records must satisfy in order to be included in the output.
- 4. Order by tells the sorted sequence on which output rows will be presented
- 5. Group by is used to partition the tables and create sub-groups
- 6. Having formulates a condition that sub-groups made by partitioning need to satisfy.

```
select  field1 , field2 , ... , fieldn
from    table1 , table2 , ... , tablen
where   (restriction-join-conditions)
order by fieldn1 , ..., fieldnm
group by fieldm1 , ... , fieldmk
having  (group-condition)
```

TYPE OF SQL COMMANDS

(Two examples of SQL-select statements)

Example A:

```
SELECT      LastName, cellPhone  
FROM        ClientTable  
WHERE       state = 'Ohio'  
ORDER BY    LastName
```

Example B:

```
SELECT      city, count(*) as TotalClients  
FROM        ClientTable  
GROUP BY    city
```

TYPE OF SQL COMMANDS

(Example 3: using a **parameterless** RawQuery - version 1)

Consider the following code fragment

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

1. The previous rawQuery contains a select-statement that retrieves all the rows (and all the columns) stored in the table tblAMIGO. The resulting table is wrapped by a Cursor object c1.
2. The 'select *' clause instructs SQL to grab all-columns held in a row.
3. Cursor c1 will be used to traverse the rows of the resulting table.
4. Fetching a row using cursor c1 requires advancing to the next record in the answer set (cursors are explained a little later in this section).
5. Fields provided by SQL must be bound to local Java variables (soon we will see to that).

TYPE OF SQL COMMANDS

(Example 3: using a **parametized** RawQuery - version 2)

Passing arguments

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

```
String mySQL = "select count(*) as Total "  
               + " from tblAmigo "  
               + " where recID > ? "  
               + " and name = ? ";  
String[] args = {"1", "BBB"};  
Cursor c1 = db.rawQuery(mySQL, args);
```

The various symbols '?' in the SQL statement represent positional placeholders. When .rawQuery() is called, the system binds each empty placeholder '?' with the supplied args-value. Here the first '?' will be replaced by "1" and the second by "BBB".

TYPE OF SQL COMMANDS

(Example 3: using a **stitched** RawQuery - version 3)

As in the previous example, assume we want to count how many friends are there whose name is 'BBB' and their recID > 1. We could use the following solution:

```
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);
```

Instead of the symbols '?' acting as placeholder, we conveniently concatenate the necessary data fragments during the assembling of our SQL statement.


TYPE OF SQL COMMANDS (SQL CURSORS)

Cursors are used to gain sequential & random access to tables produced by SQL select statements.

Cursors support one row-at-the-time operations on a table. Although in some DBMS systems cursors can be used to update the underlying dataset, the SQLite version of cursors is read-only.

Cursors include several types of operators, among them:

- 1. Positional awareness: `isFirst()`, `isLast()`, `isBeforeFirst()`, `isAfterLast()`.
- 2. Record navigation: `moveToFirst()`, `moveToLast()`, `moveToNext()`, `moveToPrevious()`, `move(n)`.
- 3. Field extraction: `getInt`, `getString`, `getFloat`, `getBlob`, `getDouble`, etc.
- 4. Schema inspection: `getColumnName()`, `getColumnNames()`, `getColumnIndex()`, `getColumnCount()`, `getCount()`.



TYPE OF SQL COMMANDS

(Example 4A: traversing a cursor—simple case)

1. Prepare a `rawQuery` passing a simple sql statement with no arguments, catch the resulting tuples in cursor `c1`.
2. Move the fetch marker to the absolute position prior to the first row in the file. The valid range of values is $-1 \leq \text{position} \leq \text{count}$.
3. Use **`moveToNext()`** to visit each row in the result set

```
String sql = "select * from tblAmigo";  
1 → Cursor c1 = db.rawQuery(sql, null);  
2 → c1.moveToPosition(-1);  
while( c1.moveToNext() ){  
    int recId = c1.getInt(0);  
    String name = c1.getString(1);  
    String phone = c1.getString(c1.getColumnIndex("phone"));  
    // do something with the record here...  
}
```

TYPE OF SQL COMMANDS

(Example 4B: traversing a cursor-enhanced navigation)

```
1 → private String showCursor(Cursor cursor) {  
    // reset cursor's top (before first row)  
    cursor.moveToPosition(-1);  
    String cursorData = "\nCursor: [";  
2 →    try { // get SCHEMA (column names & types)  
        String[] colName = cursor.getColumnNames();  
        for(int i=0; i<colName.length; i++){  
            String dataType = getColumnType(cursor, i);  
            cursorData += colName[i] + dataType;  
            if (i<colName.length-1) { cursorData+= ","; }  
        }  
    }  
    catch (Exception e) { Log.e("<<SCHEMA>>", e.getMessage() ); }  
    cursorData += "];"  
    // now get the rows  
    cursor.moveToPosition(-1); //reset cursor's top  
3 →    while (cursor.moveToNext()) {  
        String cursorRow = "\n[";  
4 →        for (int i = 0; i < cursor.getColumnCount(); i++) {  
            cursorRow += cursor.getString(i);  
            if (i<cursor.getColumnCount()-1) cursorRow += ",";  
        }  
        cursorData += cursorRow + "];"  
    }  
    return cursorData + "\n"; }  
}
```

```
5 → private String getColumnType(Cursor cursor, int i) {  
    try {  
        //peek at a row holding valid data  
        cursor.moveToFirst();  
        int result = cursor.getType(i);  
        String[] types = {":NULL", ":INT", ":FLOAT", ":STR", ":BLOB", ":UNK" };  
        //backtrack - reset cursor's top  
        cursor.moveToPosition(-1);  
        return types[result];  
    }  
    catch (Exception e) { return " "; }  
}
```

TYPE OF SQL COMMANDS

(Example 4B: traversing a cursor-enhanced navigation)

Comments:

1. The method: `showCursor(Cursor cursor)` implements the process of visiting individual rows retrieved by a SQL statement. The argument `cursor`, is a wrapper around the SQL resultset. For example, you may assume `cursor` was created using a statement such as:

```
Cursor cursor = db.rawQuery("select * from tblAMIGO", null);
```

2. The database schema for `tblAmigo` consists of the attributes: `recID`, `name`, and `phone`. The method `getColumnNames()` provides the schema.

3. The method `moveToNext` forces the cursor to travel from its current position to the next available row.

4. The accessor `.getString` is used as a convenient way of extracting SQL fields without paying much attention to the actual data type of the fields.

5. The function `.getColumnType()` provides the data type of the current field (0:null, 1:int, 2:float, 3:string, 4:blob)

TYPE OF SQL COMMANDS

(SQLite simple queries - template based queries)

Simple SQLite queries use a template-oriented schema whose goal is to 'help' non-SQL developers in their process of querying a database.

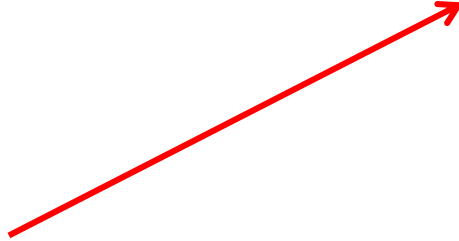
This template exposes all the components of a basic SQL-select statement.

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.

The signature of the SQLite simple .query method is:



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

TYPE OF SQL COMMANDS

(Example 5: SQLite simple queries)

Assume we need to consult an EmployeeTable (see next Figure) and find the average salary of female employees supervised by emp. 123456789. Each output row consists of Dept. No, and ladies-average-salary value. Our output should list the highest average first, then the second, and so on. Do not include depts. having less than two employees.

<pre>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");</pre>	<ul style="list-style-type: none">← table name← output columns← condition← condition-args← group by← having← order by
---	---

EMPLOYEE	
	FNAME
	MINIT
	LNAME
🔑	SSN
	BDATE
	ADDRESS
	SEX
	SALARY
	SUPERSSN
	DNO

This is a representation of the EmployeeTable used in this example.

It contains: first name, initial, last name, SSN, birthdate, address, sex, salary, supervisor's SSN, and department number.

TYPE OF SQL COMMANDS

(Example 6: SQLite simple queries)

In this example we use the tblAmigo table. We are interested in selecting the columns: recID, name, and phone. The condition to be met is that RecID must be greater 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 recRetrieved = c1.getCount();
```

We enter null in each component not supplied to the method. For instance, in this example select-args, having, and group-by are not used.

TYPE OF SQL COMMANDS

(Example 7: SQLite simple queries)

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;
```

An equivalent Android-SQLite solution using a simple template query follows.

```
1 → String [] selectColumns = {"name", "count(*) as TotalSubGroup"};  
2 → String whereCondition = "recID > ?";  
   String [] whereConditionArgs = {"3"};  
3 → String groupBy = "name";  
   String having = "count(*) <= 4";  
   String orderBy = "name";  
   Cursor cursor = db.query ("tblAMIGO", selectColumns, whereCondition, whereConditionArgs, groupBy, having, orderBy );
```


TYPE OF SQL COMMANDS

(Example 7: SQLite simple queries)

Observations

1. The **selectColumns** string array contains the output fields. One of them (name) is already part of the table, while TotalSubGroup is an alias for the computed 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.

TYPE OF SQL COMMANDS

(SQL action queries)

Action queries are the SQL way of performing maintenance operations on tables and database resources. Example of action-queries include insert, delete, update, create table, drop, etc.

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;
```

TYPE OF SQL COMMANDS

(SQL action queries: using ExecSQL)

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 easy to use –but rather limited- ***execSQL(...)*** method.

Unfortunately SQLite **execSQL** **does NOT return any data**. Therefore knowing how many records were affected by the action is not possible with this operator. Instead you should use the Android versions describe in the next section.

```
db.execSQL("update tblAMIGO set name = (name || 'XXX') where phone >= '555-1111' ");
```

This statement appends 'XXX' to the name of those whose phone number is equal or greater than '555-1111'.

Note: the symbol `||` is the SQL concatenate operator

TYPE OF SQL COMMANDS

(SQL action queries: using ExecSQL)

Alternatively, the SQL action-statement used in ExecSQL could be 'pasted' from pieces as follows:

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

The same strategy could be applied to other SQL action-statements such as:

- "delete from ... where...",
- "insert into ...values...", etc.

TYPE OF SQL COMMANDS

(Android's INSERT, DELETE, UPDATE operators)

Android provides a number of additional methods to perform insert, delete, update operations.

They all return some feedback data such as the record ID of a recently inserted row, or number of records affected by the action. This format is recommended as a better alternative than `execSQL`.

```
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)
```



TYPE OF SQL COMMANDS

(ContentValues class)

This class is used to store a set of [name, value] pairs (functionally equivalent to Bundles).

When used in combination with SQLite, a ContentValues object is just a convenient way of passing a variable number of parameters to the SQLite action functions.

Like bundles, this class supports a group of put/get methods to move data in/out of the container.

```
ContentValues myArgs = new ContentValues();  
myArgs.put("name", "ABC");  
myArgs.put("phone", "555-7777");
```

myArgs

Key	Value
name	ABC
phone	555-7777

TYPE OF SQL COMMANDS (Android's INSERT operation) +

The method tries to insert a row in a table. The row's column-values are supplied in the map called values. If successful, the method returns the rowID given to the new record, otherwise -1 is sent back.

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

Parameters

table	the table on which data is to be inserted
nullColumnHack	Empty and Null are different things. For instance, values could be defined but empty. If the row to be inserted is empty (as in our next example) this column will explicitly be assigned a NULL value (which is OK for the insertion to proceed).
values	Similar to a bundle (<i>name, value</i>) containing the column values for the row that is to be inserted.

TYPE OF SQL COMMANDS (Android's INSERT operation) +

- 1 → ContentValues rowValues= new ContentValues();
rowValues.put("name", "ABC");
rowValues.put("phone", "555-1010");
- 2 → long rowPosition = db.insert("tblAMIGO", null, rowValues);
- 3 → rowValues.put("name", "DEF");
rowValues.put("phone", "555-2020");
rowPosition = db.insert("tblAMIGO", null, rowValues);
- 4 → rowValues.clear();
- 5 → rowPosition = db.insert("tblAMIGO", null, rowValues);
- 6 → rowPosition = db.insert("tblAMIGO", "name", rowValues);

TYPE OF SQL COMMANDS (Android's INSERT operation) +

Comments

1. A set of <key, values> called rowValues is created and supplied to the insert() method to be added to tblAmigo. Each tblAmigo row consists of the columns: recID, name, phone. Remember that recID is an auto-incremented field, its actual value is to be determined later by the database when the record is accepted.
2. The newly inserted record returns its rowID (4 in this example)
3. A second record is assembled and sent to the insert() method for insertion in tblAmigo. After it is collocated, it returns its rowID (5 in this example).
4. The rowValues map is reset, therefore rowValues which is not null becomes empty.
5. SQLite rejects attempts to insert an empty record returning rowID -1.
6. The second argument identifies a column in the database that allows NULL values (NAME in this case). Now SQL purposely inserts a NULL value on that column (as well as in other fields, except the key RecID) and the insertion successfully completes.

TYPE OF SQL COMMANDS

(Android's UPDATE operation)

The method tries to update row(s) in a table. The SQL **set column = newvalue** clause is supplied in the values map in the form of [key,value] pairs. The method returns the number of records affected by the action.

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

Parameters:

table	the table on which data is to be updated
values	Similar to a bundle (name, value) containing the columnName and NewValue for the fields in a row that need to be updated.
whereClause	This is the condition identifying the rows to be updated. For instance "name = ?" where ? Is a placeholder. Passing null updates the entire table.
whereArgs	Data to replace ? placeholders defined in the whereClause.

TYPE OF SQL COMMANDS (Android's UPDATE operation)

Example: we want to use the .update() method to express the following SQL statement

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

Here are the steps to make the call using Android's equivalent Update Method

- 1 → String [] whereArgs = {"2", "7"};
ContentValues updValues = new ContentValues();
- 2 → updValues.put("name", "Maria");
- 3 → int recAffected = db.update("tblAMIGO", updValues, "recID > ? and recID < ?", whereArgs);

Comments

- 1. Our whereArgs is an array of arguments. Those actual values will replace the placeholders '?' set in the whereClause.
- 2. The map updValues is defined and populated. In our case, once a record is selected for modifications, its "name" field will be changed to the new value "maria".
- 3. The db.update() method attempts to update all records in the given table that satisfy the filtering condition set by the whereClause. After completion it returns the number of records affected by the update (0 if it fails).
- 4. The update filter verifies that "recID > ? and recID < ? ". After the args substitutions are made the new filter becomes: "recID > 2 and recID < 7".

TYPE OF SQL COMMANDS

(Android's DELETE operation)

The method is called to delete rows in a table. A filtering condition and its arguments are supplied in the call. The condition identifies the rows to be deleted. The method returns the number of records affected by the action.

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

table	the table on which data is to be deleted
whereClause	This is the condition identifying the rows to be deleted. For instance "name = ? " where ? is a placeholder. Passing null deletes the entire table.
whereArgs	Data to replace ? placeholders defined in the whereClause.

TYPE OF SQL COMMANDS

(Android's DELETE operation)

Consider the following SQL statement: `delete from tblAmigo where recID > 2 and recID < 7`

An equivalent implementation using the Android's delete method follows:

```
String [] whereArgs = {"2", "7"};  
int recAffected = db.delete("tblAMIGO", "recID > ? and recID < ?", whereArgs);
```

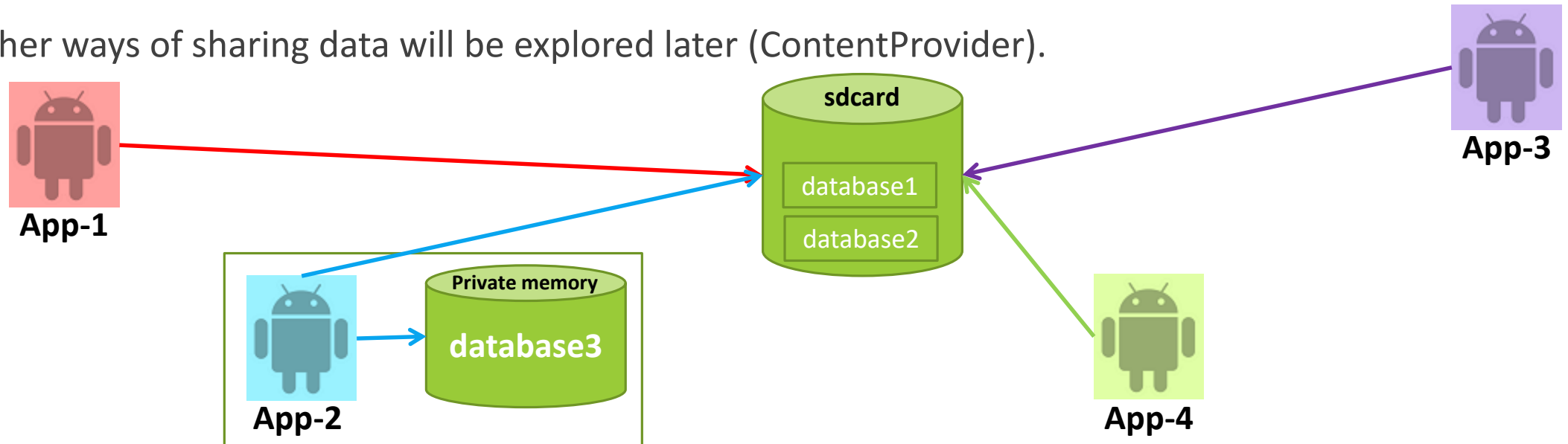
A record should be deleted if its recID is in between the values 2, and 7. The actual values are taken from the whereArgs array. The method returns the number of rows removed after executing the command (or 0 if none).

DATA VISIBILITY 👁️👁️

Any Application can access a database externally stored in the device's SD. All it's needed is knowledge of the path where the database file is located (arguable, this is an opened door to security problems).

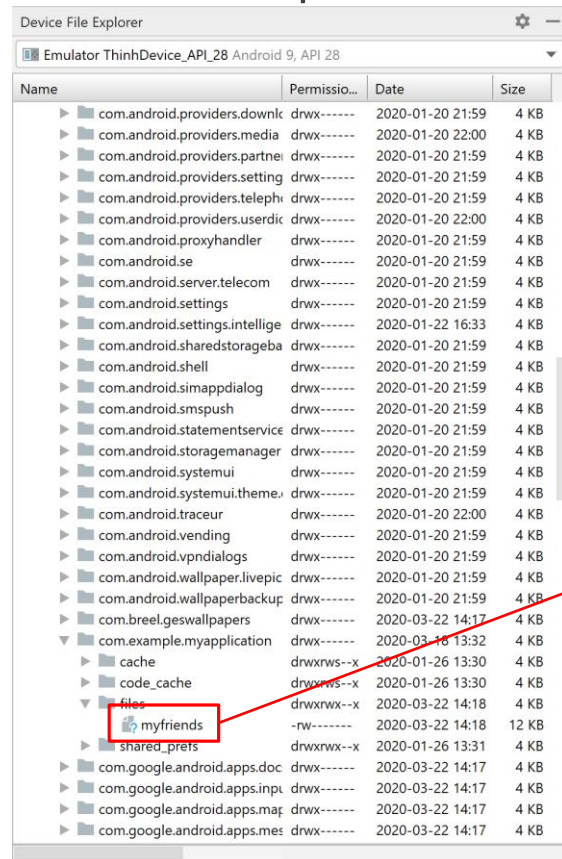
Databases created privately inside the application's process space cannot be shared (however they consume precious memory resources)

Other ways of sharing data will be explored later (ContentProvider).



DATA VISIBILITY

Emulator's File Explorer showing the location of a private database



The path to the private memory database is:
`data/data/com.example.databases/myfriends`

DATA VISIBILITY 👁👁

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:

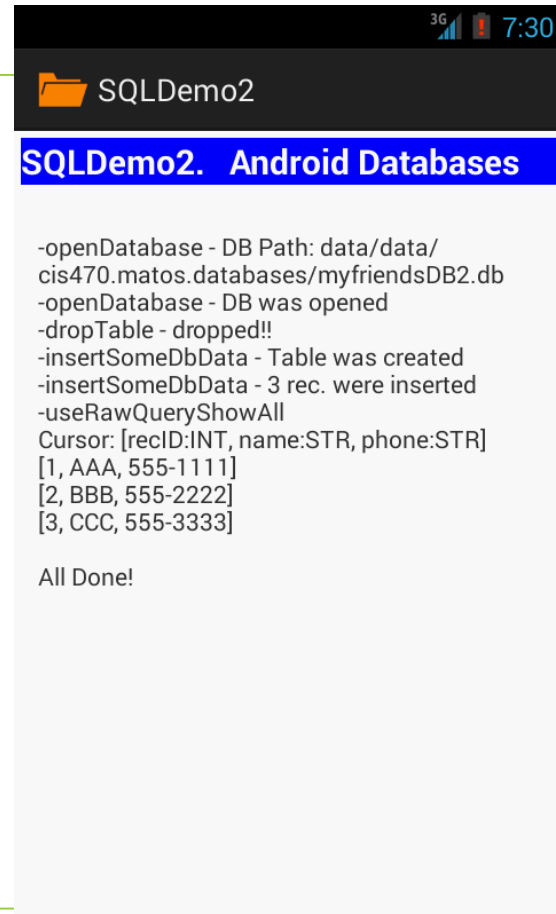
- SQLite Administrator: (<http://sqliteadmin.orbmu2k.de>)
- SQLite Manager (Firefox adds-on)



DATA VISIBILITY (EXAMPLE)

Layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:padding="4dp" android:orientation="vertical" >
    <TextView
        android:id="@+id/txtCaption" android:layout_width="match_parent"
        android:layout_height="wrap_content" android:background="#ff0000ff"
        android:text="SQLDemo2. Android Databases" android:textColor="#ffffff"
        android:textSize="20dp" android:textStyle="bold" />
    <ScrollView
        android:id="@+id/ScrollView01" android:layout_width="match_parent"
        android:layout_height="match_parent" android:padding="10dp" >
        <TextView
            android:id="@+id/txtMsg" android:layout_width="match_parent"
            android:layout_height="wrap_content" android:text="" />
    </ScrollView>
</LinearLayout>
```



DATA VISIBILITY (EXAMPLE)

```
public class SQLDemo2 extends Activity {
    SQLiteDatabase db; TextView txtMsg;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState); setContentView(R.layout.activity_main);
        txtMsg = (TextView) findViewById(R.id.txtMsg);
        try {
            openDatabase(); // open (create if needed) database
            dropTable(); // if needed drop table tblAmigos
            insertSomeDbData(); // create-populate tblAmigos
            useRawQueryShowAll(); // display all records
            useRawQuery1(); // fixed SQL with no arguments
            useRawQuery2(); // parameter substitution
            useRawQuery3(); //manual string concatenation
            useSimpleQuery1(); //simple (parametric) query
            useSimpleQuery2(); //nontrivial 'simple query'
            showTable("tblAmigo"); //retrieve all 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
            txtMsg.append("\nAll Done!");
        }
        catch (Exception e) {
            txtMsg.append("\nError onCreate: " + e.getMessage());
            finish(); } } // onCreate
```

```
private void openDatabase() {
    try {
        // path to the external SD card (something like: /storage/sdcard/...)
        // String storagePath = Environment.getExternalStorageDirectory().getPath();
        // path to internal memory file system (data/data/cis470.matos.databases)
        File storagePath = getApplication().getFilesDir();
        String myDbPath = storagePath + "/" + "myfriends";
        txtMsg.setText("DB Path: " + myDbPath);
        db = SQLiteDatabase.openDatabase(myDbPath, null, SQLiteDatabase.CREATE_IF_NECESSARY);
        txtMsg.append("\n-openDatabase - DB was opened");
    }
    catch (SQLException e) {
        txtMsg.append("\nError openDatabase: " + e.getMessage());
        finish();
    }
} // openDatabase
```

DATA VISIBILITY (EXAMPLE)

```
private void insertSomeDbData() {
    // create table: tblAmigo
    db.beginTransaction();
    try { // create table
        db.execSQL("create table tblAMIGO (recID integer PRIMARY KEY autoincrement, name text, phone text );");
        // commit your changes
        db.setTransactionSuccessful();
        txtMsg.append("\n-insertSomeDbData - Table was created");
    }
    catch (SQLException e1) {
        txtMsg.append("\nError insertSomeDbData: " + e1.getMessage());
        finish();
    }
    finally { db.endTransaction(); }
    // populate table: tblAmigo
    db.beginTransaction();
    try { // insert rows
        db.execSQL("insert into tblAMIGO(name, phone) " + " values ('AAA', '555-1111' );");
        db.execSQL("insert into tblAMIGO(name, phone) " + " values ('BBB', '555-2222' );");
        db.execSQL("insert into tblAMIGO(name, phone) " + " values ('CCC', '555-3333' );");
        // commit your changes
        db.setTransactionSuccessful();
        txtMsg.append("\n-insertSomeDbData - 3 rec. were inserted");
    }
    catch (SQLiteException e2) { txtMsg.append("\nError insertSomeDbData: " + e2.getMessage()); }
    finally { db.endTransaction(); }
} // insertSomeData
```

```
private void useRawQueryShowAll() {
    try { // hard-coded SQL select with no arguments
        Cursor c1 = db.rawQuery("select * from tblAMIGO", null);
        txtMsg.append("\n-useRawQueryShowAll" + showCursor(c1) );
    }
    catch (Exception e) { txtMsg.append("\nError useRawQuery1: " + e.getMessage()); }
} // useRawQuery1
private String showCursor( Cursor cursor) { //show SCHEMA (column names & types)
    cursor.moveToPosition(-1); //reset cursor's top
    String cursorData = "\nCursor: [";
    try { // get column names
        String[] colName = cursor.getColumnNames();
        for(int i=0; i<colName.length; i++){
            cursorData += colName[i] + getColumnType(cursor, i);
            if (i<colName.length-1){ cursorData += ", "; }
        }
    }
    catch (Exception e) { Log.e( "<<SCHEMA>>", e.getMessage() ); }
    cursorData += "]";
    // now get the rows
    cursor.moveToPosition(-1); //reset cursor's top
    while (cursor.moveToNext()) {
        String cursorRow = "\n[";
        for (int i = 0; i < cursor.getColumnCount(); i++) {
            cursorRow += cursor.getString(i);
            if (i<cursor.getColumnCount()-1) cursorRow += ", ";
        }
        cursorData += cursorRow + "]";
    }
    return cursorData + "\n";
}
```

DATA VISIBILITY (EXAMPLE)

```
private void useRawQuery1() {
    try { // hard-coded SQL select with no arguments
        String mySQL = "select * from tblAMIGO";
        Cursor c1 = db.rawQuery(mySQL, null);
        // get the first recID
        c1.moveToFirst();
        int index = c1.getColumnIndex("recID"), theRecID = c1.getInt(index);
        txtMsg.append("\n-useRawQuery1 - first recID " + theRecID);
        txtMsg.append("\n-useRawQuery1" + showCursor(c1));
    }
    catch (Exception e) { txtMsg.append("\nError useRawQuery1: " + e.getMessage()); }
} // useRawQuery1

private void useRawQuery2() {
    try { // use: ? as argument's placeholder
        String mySQL = " select recID, name, phone from tblAmigo "
            + " where recID > ? " + " and name = ? ";
        String[] args = { "1", "BBB" };
        Cursor c1 = db.rawQuery(mySQL, args);
        // pick NAME from first returned row
        c1.moveToFirst();
        int index = c1.getColumnIndex("name");
        String theName = c1.getString(index);
        txtMsg.append("\n-useRawQuery2 Retrieved name: " + theName);
        txtMsg.append("\n-useRawQuery2 " + showCursor(c1));
    }
    catch (Exception e) { txtMsg.append("\nError useRawQuery2: " + e.getMessage()); }
} // useRawQuery2
```

```
private String getColumnType(Cursor cursor, int i) {
    try {
        //peek at a row holding valid data
        cursor.moveToFirst();
        int result = cursor.getType(i);
        String[] types = { ":NULL", ":INT", ":FLOAT", ":STR", ":BLOB", ":UNK" };
        //backtrack - reset cursor's top
        cursor.moveToPosition(-1);
        return types[result];
    }
    catch (Exception e) {
        return "";
    }
}
```

DATA VISIBILITY (EXAMPLE)

```
private void useRawQuery3() {
    try {
        // arguments injected by manual string concatenation
        String[] args = { "1", "BBB" };
        String mySQL = " select recID, name, phone"
                        + " from tblAmigo "
                        + " where recID > " + args[0]
                        + " and name = " + args[1] + """";

        Cursor c1 = db.rawQuery(mySQL, null);
        // pick PHONE from first returned row
        int index = c1.getColumnIndex("phone"); //case sensitive
        c1.moveToNext();
        String thePhone = c1.getString(index);
        txtMsg.append("\n-useRawQuery3 - Phone: " + thePhone);
        txtMsg.append("\n-useRawQuery3 " + showCursor(c1) );
    }
    catch (Exception e) {
        txtMsg.append("\nError useRawQuery3: " + e.getMessage());
    }
} // useRawQuery3
```

```
private void useSimpleQuery1() {
    try {
        // simple-parametric query on one table.
        // arguments: tableName, columns, condition, cond-args,
        // groupByCol, havingCond, orderBy
        // the next parametric query is equivalent to SQL stmt:
        // select recID, name, phone from tblAmigo
        // where recID > 1 and length(name) >= 3
        // order by recID
        Cursor c1 = db.query("tblAMIGO", new String[] { "recID", "name", "phone" }, "recID > 1 and length(name) >= 3 ", null, null, null, "recID");
        // get NAME from first data row
        int index = c1.getColumnIndex("phone");
        c1.moveToFirst();
        String theName = c1.getString(index);
        txtMsg.append("\n-useSimpleQuery1 - Total rec " + theName);
        txtMsg.append("\n-useSimpleQuery1 " + showCursor(c1) );
    }
    catch (Exception e) {
        txtMsg.append("\nError useSimpleQuery1: " + e.getMessage());
    }
} // useSimpleQuery1
```

DATA VISIBILITY (EXAMPLE)

```
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 c1 = db.query("tblAMIGO", selectColumns, whereCondition, whereConditionArgs, groupBy, having, orderBy);
        int theTotalRows = c1.getCount();
        txtMsg.append("\n-useSimpleQuery2 - Total rec: " + theTotalRows);
        txtMsg.append("\n-useSimpleQuery2 " + showCursor(c1));
    }
    catch (Exception e) { txtMsg.append("\nError useSimpleQuery2: " + e.getMessage()); }
} // useSimpleQuery2

private void showTable(String tableName) {
    try {
        String sql = "select * from " + tableName;
        Cursor c = db.rawQuery(sql, null);
        txtMsg.append("\n-showTable: " + tableName + showCursor(c));
    }
    catch (Exception e) { txtMsg.append("\nError showTable: " + e.getMessage()); }
} // useCursor1
```

```
private void useCursor1() {
    try { // this is similar to showCursor(...)
        // obtain a list of records[recId, name, phone] from DB
        String[] columns = { "recID", "name", "phone" };
        // using simple parametric cursor
        Cursor c = db.query("tblAMIGO", columns, null, null, null, null, "recID");
        int theTotal = c.getCount();
        txtMsg.append("\n-useCursor1 - Total rec " + theTotal);
        txtMsg.append("\n");
        int idCol = c.getColumnIndex("recID");
        int nameCol = c.getColumnIndex("name");
        int phoneCol = c.getColumnIndex("phone");
        c.moveToPosition(-1);
        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) {
        txtMsg.append("\nError useCursor1: " + e.getMessage());
        finish();
    }
} // useCursor1
```

DATA VISIBILITY (EXAMPLE)

```
private void updateDB() {
    // action query performed using execSQL
    // add 'XXX' to the name of person whose phone is 555-1111
    txtMsg.append("\n-updateDB");
    try {
        String thePhoneNo = "555-1111";
        db.execSQL(" update tblAMIGO set name = (name || 'XXX') "
                    + " where phone = '" + thePhoneNo + "' ");
        showTable("tblAmigo");
    }
    catch (Exception e) { txtMsg.append("\nError updateDB: " + e.getMessage()); }
    useCursor1();
}

private void dropTable() {
    // (clean start) action query to drop table
    try {
        db.execSQL(" drop table tblAmigo;");
        // >>Toast.makeText(this, "Table dropped", 1).show();
        txtMsg.append("\n-dropTable - dropped!!");
    }
    catch (Exception e) {
        txtMsg.append("\nError dropTable: " + e.getMessage());
        finish();
    }
}
```

```
public void useInsertMethod() {
    // an alternative to SQL "insert into table values(...). ContentValues is an Android dynamic row-like container
    try {
        ContentValues initialValues = new ContentValues();
        initialValues.put("name", "ABC");
        initialValues.put("phone", "555-4444");
        int rowPosition = (int) db.insert("tblAMIGO", null, initialValues);
        txtMsg.append("\n-useInsertMethod rec added at: " + rowPosition);
        showTable("tblAmigo");
    }
    catch (Exception e) { txtMsg.append("\n-useInsertMethod - Error: " + e.getMessage()); }
} // useInsertMethod

private void useUpdateMethod() {
    try { // using the 'update' method to change name of selected friend
        String[] whereArgs = { "1" };
        ContentValues updValues = new ContentValues();
        updValues.put("name", "Maria");
        int recAffected = db.update("tblAMIGO", updValues, "recID = ?", whereArgs);
        txtMsg.append("\n-useUpdateMethod - Rec Affected " + recAffected);
        showTable("tblAmigo");
    }
    catch (Exception e) { txtMsg.append("\n-useUpdateMethod - Error: " + e.getMessage()); }
}
```

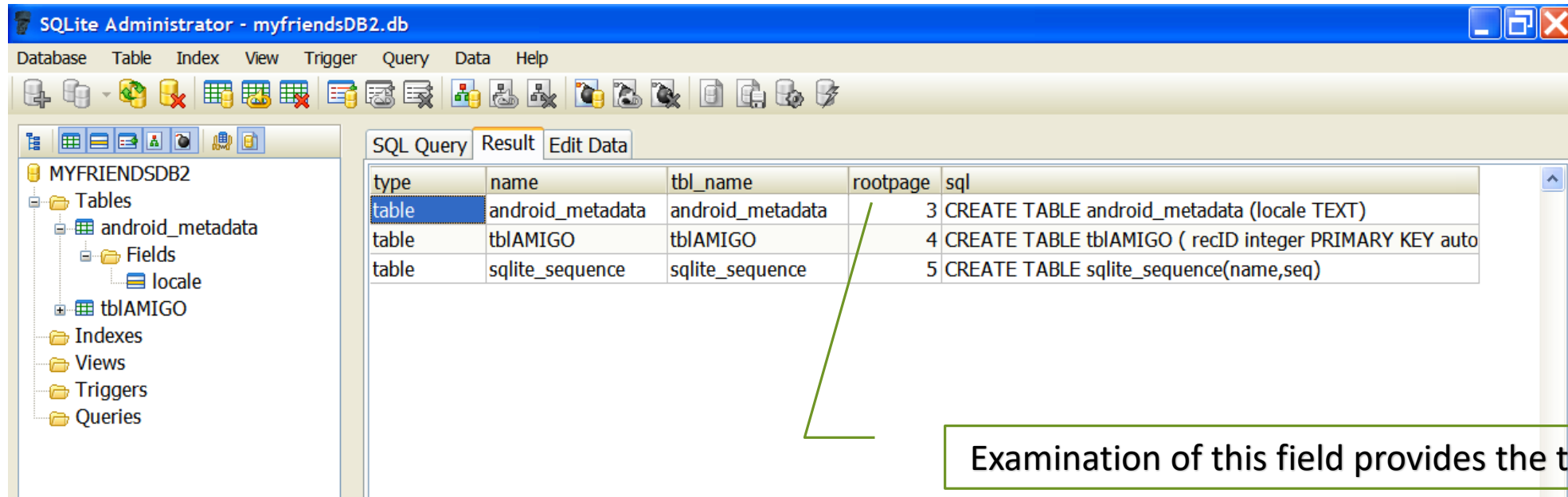
DATA VISIBILITY (EXAMPLE)

```
private void useDeleteMethod() {  
    // using the 'delete' method to remove a group of friends whose id# is between 2 and 7  
    try {  
        String[] whereArgs = { "2" };  
        int recAffected = db.delete("tblAMIGO", "recID = ?", whereArgs);  
        txtMsg.append("\n-useDeleteMethod - Rec affected " + recAffected);  
        showTable("tblAmigo");  
    }  
    catch (Exception e) { txtMsg.append("\n-useDeleteMethod - Error: " + e.getMessage()); }  
}  
} // class
```


APPENDIX

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

Example: select * from sqlite_master;



The screenshot shows the SQLite Administrator interface for a database named 'myfriendsDB2.db'. The 'Result' tab is active, displaying the output of the query 'select * from sqlite_master;'. The results are shown in a table with five columns: type, name, tbl_name, rootpage, and sql. The first row is highlighted, showing 'table' as the type, 'android_metadata' as the name and tbl_name, and '3' as the rootpage. A green line points from the 'rootpage' column to a text box that says 'Examination of this field provides the table schema'.

type	name	tbl_name	rootpage	sql
table	android_metadata	android_metadata	3	CREATE TABLE android_metadata (locale TEXT)
table	tblAMIGO	tblAMIGO	4	CREATE TABLE tblAMIGO (recID integer PRIMARY KEY auto
table	sqlite_sequence	sqlite_sequence	5	CREATE TABLE sqlite_sequence(name,seq)

Examination of this field provides the table schema

APPENDIX

In Java code you may formulate 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;  
}
```

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

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