SQLite

Session 6

Objective

What is SQLite?

SQLite Datatype

Database – Creation

Database - Helper class

Select, Insert, Update, Delete in SQLite

What is SQLite

- SQLite is a opensource SQL database that stores data to a text file on a device. Android comes in with built in SQLite database implementation.
- SQLite supports all the relational database features. In order to access this database, you don't need to establish any kind of connections for it like JDBC,ODBC e.t.c

SQLite Datatype

NULL – null value

INTEGER - signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value

REAL - a floating point value, 8-byte IEEE floating point number.

TEXT - text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE).

BLOB. The value is a blob of data, stored exactly as it was input.

SQLite Datatype

This is quite different than the normal SQL data types so please read:

http://www.sqlite.org/datatype3.html

android.database.sqlite

-Contains the SQLite database management classes that an application would use to manage its own private database.

- Create a database you just need to call this method openOrCreateDatabase with your database name and mode as a parameter.

```
SQLiteDatabase mydatabase = openOrCreateDatabase("your database name",MODE_PRIVATE,null);
```

- SQLiteDatabase - Exposes methods to manage a SQLite database.
- SQLiteOpenHelper - A helper class to manage database creation and version management.

Contains the methods for: creating, opening, closing, inserting, updating, deleting and quering an SQLite database

Create a SQLite database

This method will open an existing database or create one in the application data area

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

```
import android.database.sqlite.SQLiteDatabase;

SQLiteDatabase  myDatabase;

myDatabase = openOrCreateDatabase ("my_sqlite_database.db" ,
                    SQLiteDatabase.CREATE_IF_NECESSARY , null);
```

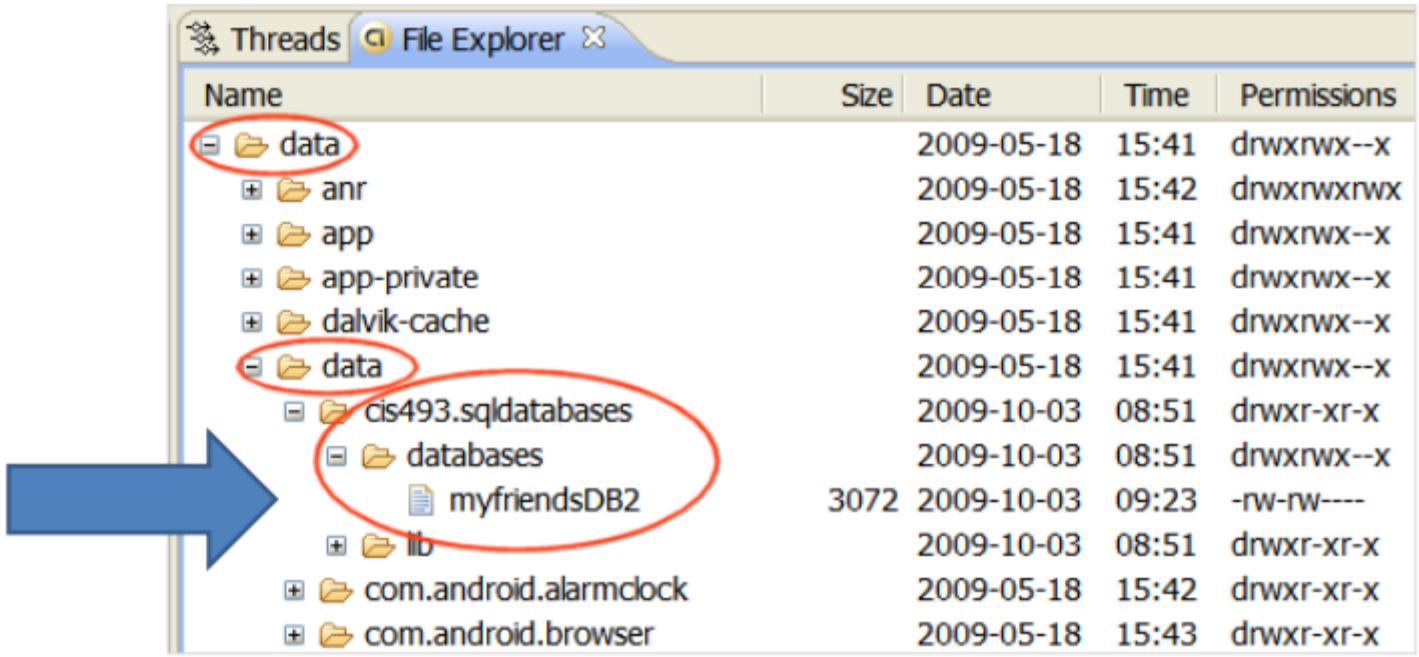
Create a SQLite database

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".

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

Create a SQLite database

Database is saved in the device's memory




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

Creating Tables

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 );
```

Create a static string containing the SQLite CREATE statement, use the execSQL( ) method to execute it.

```
String  createAuthor = "CREAT TABLE   authors (
                              id  INTEGER PRIMARY KEY
AUTOINCREMENT,
                              fname  TEXT,
                               lname  TEXT);

myDatabase.execSQL(createAuthor);
```

Creating Tables

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.

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

```
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' );"  );
```

Insert


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

Convenient method for inserting a row into the database

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

Insert


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

Convenient method for inserting a row into the database

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);
```

Update


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

table the table to update

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

Update

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 );
```

Delete


```
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.

Delete

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);
```

Queries

SQL-select statements are based on the following components

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

```
 where   ( restriction-join-conditions )
order by  fieldn1, …, fieldnm
group by  fieldm1, … , fieldmk
 having   (group-condition)
```

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

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

Using RawQuery

```
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.

Using Parametized RawQuery

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

After the substitutions are made the resulting SQL statement is:

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

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 )
```

Queries

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",          ← table name
           columns,                  ← columns
           "sex = ? And superSsn = ? " ,  ← condition
           conditionArgs,            ← condition args
           "Dno",                    ← group by
           "Count(*) > 2",           ← having
           "AVG Desc "               ← order by
);
```

Queries

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();
```

Queries

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  );
```

Queries


```
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  );
```

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.

Cursors

Android cursors are used to gain (sequential & 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 operators, 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.*)

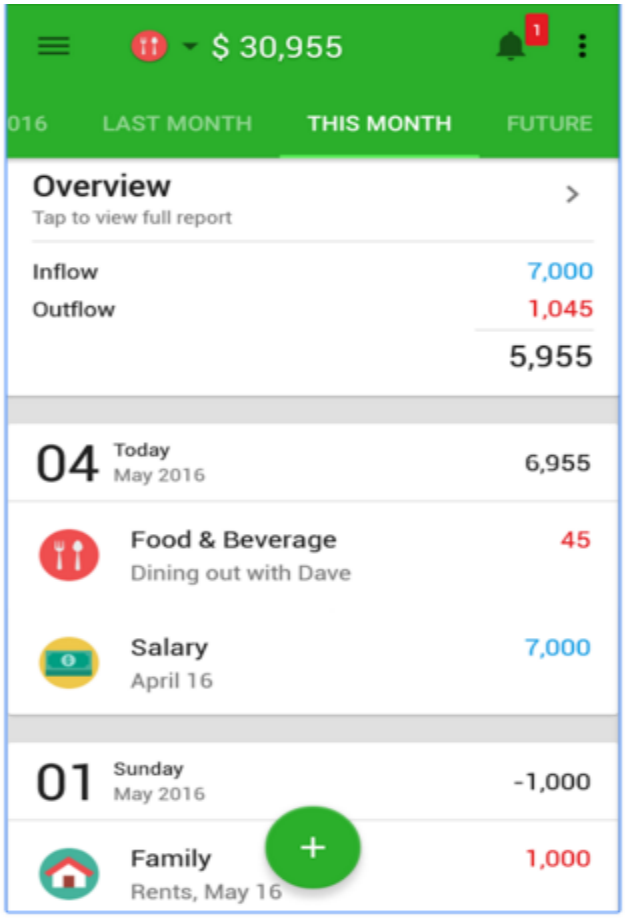
4. Schema inspection (getColumnName, getColumnNames, getColumnIndex, getColumnCount, getCount)

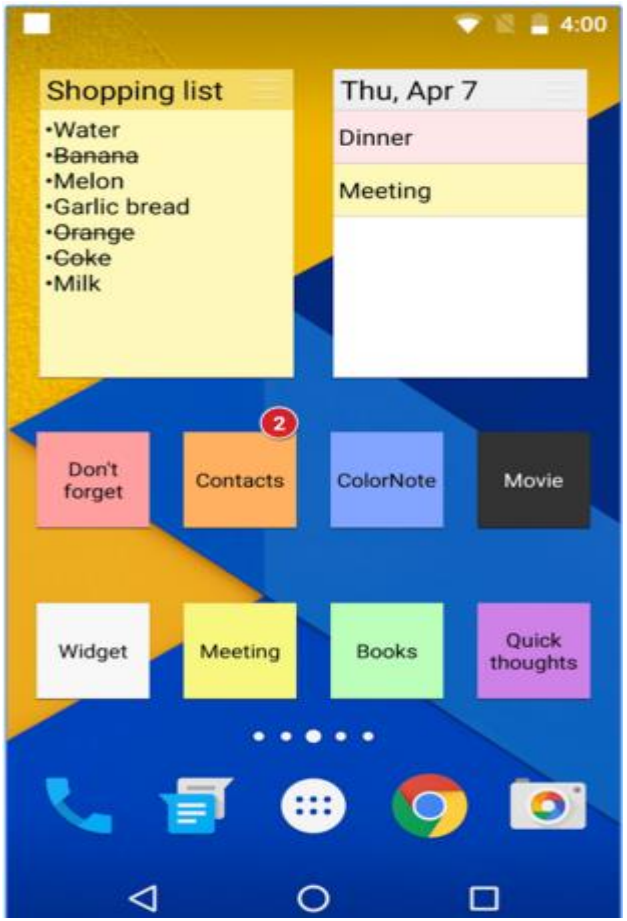
Cursors


```
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] );
}
```

Conclude

Using SQLite you can make Notes app, saving marker news, saving high score in game, expenditure management app, …etc.





Summary

In this session, we learnt:

- What is SQLite?
- SQLite Datatype
- Database – Creation
- Database - Helper class
- Select, Insert, Update, Delete in SQLite