SQLite

Nikola Ralev, Ph.D.
nralev@setelis.com
Setelis Labs LLC

SQLite

- SQLite is a small footprint database
- management system that provides access via standard SQL commands, i.e. SELECT, INSERT, UPDATE, DELETE.
- supports transactions and is very robust in terms of database consistency.
- The main difference compared to a standard SQL system, e.g. mySQL, is that even though the columns in a SQLite database are given datatypes, they are not enforced. Thus, unlike a mySQL database, even if a column is defined to store INTEGER data, any type of data can actually be stored in that field. Thus it is the responsibility of the application code, i.e. ORM methods, to interpret the data appropriately.

Create database

- SQLite database file can be included in the assets directory of the project
- Will be embedded in the .apk file
- simply copied onto the local database file

Create database

- Create database using **SQLiteOpenHelper**
  - **Constructor -** which calls the base class constructor.
  - **onCreate() -** which can be used to build an initial database, e.g. from string constants stored in string.xml
  - **onUpgrade() -** which can be used to modify a database based on a version number allowing migration from one schema to another (or possibly just deleting all previous tables and creating new ones)

Create database sample

```
INVENTORY_TABLE = "CREATE TABLE IF NOT EXISTS items

(

id INTEGER PRIMARY KEY AUTOINCREMENT,
name VARCHAR(80) NOT NULL,
quantity INTEGER NOT NULL,
vendor_name VARCHAR(80) NOT NULL,
vendor_phone_num VARCHAR(12) NOT NULL

)";
```

Accessing database

- **query()**, **insert()**, **update()** and **delete()** methods in the SQLiteDatabase class to perform transactions on the database
- use the execSQL() method to execute a single SQL statement against the database

Insert()

- long insert(String table, String nullColumnHack, ContentValues values)
  - returns the id of the inserted row (or -1 if an error occurs
  - table - the name of the table to run the query against
  - nullColumnHack - prevents an empty row from being inserted by specifying a column name to insert a null into (usually set to null)
  - values - a ContentValues map that specifies the column name/value pairs to be inserted into the row

Insert sample

insert() for an item with name "Oranges" and quantity 10 might be:

```
ContentValues vals = new ContentValues();
vals.put("name","Oranges");
vals.put("quantity",10);
long row = db.insert(INVENTORY_TABLE, null, vals);
```

Query()

- **Cursor query(String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy**)
  - Returns cursor
  - table - name of the table to run the query against
  - columns - list of columns to return the values of in the Cursor
  - selection - filter for which rows to return, i.e. the WHERE clause without the WHERE keyword (set to null to return all rows)
  - selectionArgs - a list of values to substitute for ? placeholders in the selection
  - groupBy - filter corresponding to the GROUP BY SQL clause (null for no grouping)
  - having - filter corresponding to the HAVING clause (null for no filter)
  - orderBy - filter corresponding to the ORDER BY clause (null for default sort order)

Query sample

query() for an item with name "Apples" might be

```
Cursor c = db.query(INVENTORY_TABLE, null, "name=?",
new String{}{"Apples"},null,null,null);
```

Update()

- int update(String table, ContentValues values, String whereClause, String[] whereArgs)
  - returns an integer indicating the number of rows that were modified
  - table - the name of the table to run the query against
  - values - a ContentValues map that specifies column names/new values to be updated
  - whereClause - filter to specify matching criteria to only update particular matching rows (null updates all rows)
  - whereArgs - a list of values to substitute for any placeholders in the whereClause

Update sample

update() to change the quantity of "Oranges" to 42 might be:

```
ContentValues vals = new ContentValues();
vals.put("quantity",42);
int numRows = db.update(INVENTORY_TABLE, vals, "name=?", "Oranges");
```

Delete()

- int delete(String table, String whereClause, String[] whereArgs)
  - returns an integer indicating the number of rows that were removed (if a whereClause is specified, 0 otherwise) and has parameters:
  -
  - table - the name of the table to run the query against
  - whereClause - filter to specify matching criteria to only delete particular matching rows (null deletes all rows)
  - whereArgs - a list of values to substitute for any placeholders in the whereClause

Delete sample

delete() to remove "Oranges" might be:

```
int delRows = db.delete(INVENTORY_TABLE, "name=?",
"Oranges");
```

ExecSQL()

- void execSQL(String sql, Object[] bindArgs)
  - The method cannot return a value and thus cannot be used with SELECT statements. It has a single parameter:
  - sql - which is a string representing the SQL statement to execute
  - bindArgs - a list of values to substitute for any placeholders

ExecSQL sample

execSql to remove "Oranges" might be:

```
String delSql = "DELETE FROM " + INVENTORY_TABLE +
 "WHERE name = ?";
Object[] bindArgs = new Object[]{"Oranges"};
db.execSQL(delSql, bindArgs);
```

Cursor

Cursor objects have several commonly used methods:

- GetCount() - number of rows returned by the query
- moveToFirst() - reset the cursor to the first returned row
- moveToPosition(int i) - sets the cursor's position to the ith row
- close() - clean up the cursor resources
- getColumnIndexOrThrow(String name) - retrieves the column index for a given column name
- getInt(int idx), getString(int idx), etc. - gets the value

Cursor sample

```
Cursor ic;

// Obtain cursor via database query

// Iterate over cursor rows
for (int i = 0; i < ic.getCount(); i++) {
        // Get next row
        ic.moveToPosition(i);

        // Extract fields from row
        int id = ic.getInt(ic.getColumnIndexOrThrow("_id");
        String name = ic.getString(ic.getColumnIndexOrThrow("name");
        int quantity = ic.getInt(ic.getColumnIndexOrThrow("quantity");

        // Construct new model object
        Item item = new Item(id, name, quantity);
}

ic.close();
```

Performance tips

- Use multiple databases to avoid locking
- Use multiple databases to allow multi-threaded access
- Avoid multiple open and close statements
- Synchronous access takes time – only use on tables when needed
- Use Indexes
- Use transactions for larger groups of data manipulation
- Use Indexes when creating table (PRIMARY and UNIQUE) or by CREATE INDEX
- Split tables

Performance tips

- Compact database - Use VACUUM – manual and auto (or on demand based on pagecount and freelist count) – do in thread, may take time do after big delete
- For complex nested queries, break up and use temporary tables to stire results
- Store Compressed data
- Use LIMIT=1 when checking for something existing
- Avoid GLOB and LIKE where possible. Avoid LENGTH if long strings
- Use IN instead of OR when comparing single variable to multiple values - can sue index
- Use SQL statements rath er then code

Any questions ?