Mobile Applications (420-P84-AB)

John Abbott College

SQLite

Aakash Malhotra

About SQLite

SQLite is a relational database management system contained in a C programming library. In contrast to many other database management systems, SQLite is not a client–server database engine. Rather, it is embedded into the end program.

SQLite is ACID-compliant and implements most of the SQL standard.

Atomicity

Transactions are often composed of multiple statements. Atomicity guarantees that each transaction is treated as a single "unit", which either succeeds completely, or fails completely: if any of the statements constituting a transaction fails to complete, the entire transaction fails and the database is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors and crashes.

About SQLite

Consistency

Consistency ensures that a transaction can only bring the database from one valid state to another, maintaining database invariants: any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof. This prevents database corruption by an illegal transaction, but does not guarantee that a transaction is correct.

Isolation

Transactions are often executed concurrently (e.g., reading and writing to multiple tables at the same time). Isolation ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially. Isolation is the main goal of concurrency control; depending on the method used, the effects of an incomplete transaction might not even be visible to other transactions.

About SQLite

Durability

Durability guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure (e.g., power outage or crash). This usually means that completed transactions (or their effects) are recorded in non-volatile memory.

SQLite is a popular choice as embedded database software for local/client storage in application software such as web browsers. It is arguably the most widely deployed database engine, as it is used today by several widespread browsers, operating systems, and embedded systems (such as mobile phones), among others. SQLite has bindings to many programming languages.

SQLite with Android

The APIs you'll need to use a database on Android are available in the `android.database.sqlite` package.

Room Persistence Library is the officially supported ORM for SQLite with Android.

Read Write operation

- Create basic insertion and display app structure

Extra:

- Back button on change of activity
- Common menu for more than one activity

Read Write operation

- Create a new MyDbHelperClass that extends SQLiteOpenHelper

```
public class MyDbHelper extends SQLiteOpenHelper {

}
```

- Implement Constructor

```
public MyDbHelper(Context context, String name, SQLiteDatabase.CursorFactory factory, int
version) {

        super(context, name, factory, version);

    }
```

Read Write operation

- Override onCreate and create a new table in it
  - @Override
  - public void onCreate(SQLiteDatabase sqLiteDatabase) {
  - sqLiteDatabase.execSQL("create table user (name text, address text, phone text)");
  - }
- Override onUpgrade. Delete the table if it exists and call onCreate()
  - @Override
  - public void onUpgrade(SQLiteDatabase sqLiteDatabase, int i, int i1) {
  - sqLiteDatabase.execSQL("drop table if exists user");
  - onCreate(sqLiteDatabase);
  - }

Read Write operation

- Create a method in main activity to insert a row in table

```
-     public void saveValuesToDB() {
-         MyDbHelper dbHelper = new MyDbHelper(this, "userdb", null, 1);
-         SQLiteDatabase db = dbHelper.getWritableDatabase();
-
-         ContentValues values = new ContentValues();
-         values.put("name", "john");
-         values.put("address", "Narnia");
-         values.put("phone", "514123456");
-
-         long rowId = db.insert("user", null,values );
-         Log.i("MYTAG","Row number is " + rowId);
-     }
```

onUpgrade

Called when the database needs to be upgraded. The implementation should use this method to drop tables, add tables, or do anything else it needs to upgrade to the new schema version.

If you add new columns you can use ALTER TABLE to insert them into a live table. If you rename or remove columns you can use ALTER TABLE to rename the old table, then create the new table and then populate the new table with the contents of the old table.

This method executes within a transaction. If an exception is thrown, all changes will automatically be rolled back.

In Live App

```
public void onUpgrade(...)
{
  var migrations = getMigrationsToRun(oldVersion, newVersion);
  foreach (migrationFile in migrations)
  {
        // Run all your migrations here
  }
}
```

Define Schema and Contract

One of the main principles of SQL databases is the schema: a formal declaration of how the database is organized. The schema is reflected in the SQL statements that you use to create your database. You may find it helpful to create a companion class, known as a *contract* class, which explicitly specifies the layout of your schema in a systematic and self-documenting way.

A contract class is a container for constants that define names for URIs, tables, and columns. The contract class allows you to use the same constants across all the other classes in the same package. This lets you change a column name in one place and have it propagate throughout your code.

Define Schema and Contract

A good way to organize a contract class is to put definitions that are global to your whole database in the root level of the class. Then create an inner class for each table. Each inner class enumerates the corresponding table's columns.

```java
public final class FeedReaderContract {
    // To prevent someone from accidentally instantiating the contract class,
    // make the constructor private.
    private FeedReaderContract() {}

    /* Inner class that defines the table contents */
    public static class FeedEntry implements BaseColumns {
        public static final String TABLE_NAME = "entry";
        public static final String COLUMN_NAME_TITLE = "title";
        public static final String COLUMN_NAME_SUBTITLE = "subtitle";
    }
}
```

Create a Database using SQL Helper

Once you have defined how your database looks, you should implement methods that create and maintain the database and tables. Here are some typical statements that create and delete a table:

```
private static final String SQL_CREATE_ENTRIES =
    "CREATE TABLE " + FeedEntry.TABLE_NAME + " (" +
    FeedEntry._ID + " INTEGER PRIMARY KEY," +
    FeedEntry.COLUMN_NAME_TITLE + " TEXT," +
    FeedEntry.COLUMN_NAME_SUBTITLE + " TEXT)";

private static final String SQL_DELETE_ENTRIES =
    "DROP TABLE IF EXISTS " + FeedEntry.TABLE_NAME;
```

Create a Database using SQL Helper

Just like files that you save on the device's internal storage, Android stores your database in your app's private folder. Your data is secure, because by default this area is not accessible to other apps or the user.

The `SQLiteOpenHelper` class contains a useful set of APIs for managing your database. When you use this class to obtain references to your database, the system performs the potentially long-running operations of creating and updating the database only when needed and *not during app startup*. All you need to do is call `getWritableDatabase()` or `getReadableDatabase()`.

> **Note:** Because they can be long-running, be sure that you call **`getWritableDatabase()`** or **`getReadableDatabase()`** in a background thread, such as with **`AsyncTask`** or **`IntentService`**.

To use `SQLiteOpenHelper`, create a subclass that overrides the `onCreate()` and `onUpgrade()`callback methods. You may also want to implement the `onDowngrade()` or `onOpen()` methods, but they are not required.

Create a Database using SQL Helper

For example, here's an implementation of SQLiteOpenHelper that uses some of the commands shown above:

```
public class FeedReaderDbHelper extends SQLiteOpenHelper {
    // If you change the database schema, you must increment the database
version.
    public static final int DATABASE_VERSION = 1;
    public static final String DATABASE_NAME = "FeedReader.db";

    public FeedReaderDbHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }
```

Create a Database using SQL Helper

```
public void onCreate(SQLiteDatabase db) {
        db.execSQL(SQL_CREATE_ENTRIES);
    }
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // This database is only a cache for online data, so its upgrade
policy is
        // to simply to discard the data and start over
        db.execSQL(SQL_DELETE_ENTRIES);
        onCreate(db);
    }
```

Create a Database using SQL Helper

```
public void onDowngrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        onUpgrade(db, oldVersion, newVersion);
    }
}
```

Put information in Database

```
// Gets the data repository in write mode
SQLiteDatabase db = mDbHelper.getWritableDatabase();

// Create a new map of values, where column names are the keys
ContentValues values = new ContentValues();
values.put(FeedEntry.COLUMN_NAME_TITLE, title);
values.put(FeedEntry.COLUMN_NAME_SUBTITLE, subtitle);

// Insert the new row, returning the primary key value of the new row
long newRowId = db.insert(FeedEntry.TABLE_NAME, null, values);
```

Put information in Database

The first argument for `insert()` is simply the table name.

The second argument tells the framework what to do in the event that the `ContentValues` is empty (i.e., you did not `put` any values). If you specify the name of a column, the framework inserts a row and sets the value of that column to null. If you specify `null`, like in this code sample, the framework does not insert a row when there are no values.

The `insert()` methods returns the ID for the newly created row, or it will return -1 if there was an error inserting the data. This can happen if you have a conflict with pre-existing data in the database.

Read Information from DB

To read from a database, use the `query()` method, passing it your selection criteria and desired columns. The method combines elements of `insert()` and `update()`, except the column list defines the data you want to fetch (the "projection"), rather than the data to insert. The results of the query are returned to you in a `Cursor` object.

Read Information from DB

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// Define a projection that specifies which columns from the database
// you will actually use after this query.
String[] projection = {
    BaseColumns._ID,
    FeedEntry.COLUMN_NAME_TITLE,
    FeedEntry.COLUMN_NAME_SUBTITLE
    };
```

Read Information from DB

```
// Filter results WHERE "title" = 'My Title'
String selection = FeedEntry.COLUMN_NAME_TITLE + " = ?";
String[] selectionArgs = { "My Title" };

// How you want the results sorted in the resulting Cursor
String sortOrder =
    FeedEntry.COLUMN_NAME_SUBTITLE + " DESC";
```

Read Information from DB

```
Cursor cursor = db.query(
    FeedEntry.TABLE_NAME,   // The table to query
    projection,             // The array of columns to return (pass null to
get all)
    selection,              // The columns for the WHERE clause
    selectionArgs,          // The values for the WHERE clause
    null,                   // don't group the rows
    null,                   // don't filter by row groups
    sortOrder               // The sort order
    );
```

Read Information from DB

The third and fourth arguments (`selection` and `selectionArgs`) are combined to create a WHERE clause. Because the arguments are provided separately from the selection query, they are escaped before being combined. This makes your selection statements immune to SQL injection. For more detail about all arguments, see the `query()` reference.

To look at a row in the cursor, use one of the `Cursor` move methods, which you must always call before you begin reading values. Since the cursor starts at position -1, calling `moveToNext()` places the "read position" on the first entry in the results and returns whether or not the cursor is already past the last entry in the result set. For each row, you can read a column's value by calling one of the `Cursor`get methods, such as `getString()` or `getLong()`. For each of the get methods, you must pass the index position of the column you desire, which you can get by calling `getColumnIndex()` or `getColumnIndexOrThrow()`. When finished iterating through results, call `close()` on the cursor to release its resources.

Delete information from a database

To delete rows from a table, you need to provide selection criteria that identify the rows to the `delete()` method. The mechanism works the same as the selection arguments to the `query()`method. It divides the selection specification into a selection clause and selection arguments. The clause defines the columns to look at, and also allows you to combine column tests. The arguments are values to test against that are bound into the clause. Because the result isn't handled the same as a regular SQL statement, it is immune to SQL injection.

Delete information from a database

```java
// Define 'where' part of query.
String selection = FeedEntry.COLUMN_NAME_TITLE + " LIKE ?";
// Specify arguments in placeholder order.
String[] selectionArgs = { "MyTitle" };
// Issue SQL statement.
int deletedRows = db.delete(FeedEntry.TABLE_NAME, selection, selectionArgs);
```

The return value for the `delete()` method indicates the number of rows that were deleted from the database.

Update a database

When you need to modify a subset of your database values, use the `update()` method.

Updating the table combines the `ContentValues` syntax of `insert()` with the `WHERE` syntax of `delete()`.

Update a database

```
SQLiteDatabase db = mDbHelper.getWritableDatabase();
// New value for one column
String title = "MyNewTitle";
ContentValues values = new ContentValues();
values.put(FeedEntry.COLUMN_NAME_TITLE, title);
// Which row to update, based on the title
String selection = FeedEntry.COLUMN_NAME_TITLE + " LIKE ?";
String[] selectionArgs = { "MyOldTitle" };
int count = db.update(
    FeedReaderDbHelper.FeedEntry.TABLE_NAME,
    values,
    selection,
    selectionArgs);
```

Update a database

The return value of the `update()` method is the number of rows affected in the database.

Persisting database connection

The return value of the `update()` method is the number of rows affected in the database.

Since `getWritableDatabase()` and `getReadableDatabase()` are expensive to call when the database is closed, you should leave your database connection open for as long as you possibly need to access it. Typically, it is optimal to close the database in the `onDestroy()` of the calling Activity.

```
@Override
protected void onDestroy() {
    mDbHelper.close();
    super.onDestroy();
}
```

Misc Topics

Stetho by FB

For visual representation of sqlite db.

Specifying Parent Activity

```
<activity
...
android:parentActivityName ="com.example.app_name.A"  >
 ...
</activity>
```

References:

- https://en.wikipedia.org/wiki/SQLite
- https://developer.android.com/training/data-storage/sqlite
- https://en.wikipedia.org/wiki/ACID_(computer_science)