SQLite DATABASES
(Chapter 14)

SQLite: Motivation

- An app may need to save its data after it's done running (photos, videos, contacts, etc.)
- We'll refer to this data as the *database* (db) for the app
- Two basic ways of storing/retrieving the db
  - Cloud / online
  - Onboard the device
- In either case, we need a format for storing and retrieving the data

SQLite: Motivation

- **Q**: So why not just use XML or some other standard file format for the db?
- **A**: *Inefficient*: to enter or remove an item from the we'd either have to
  - Read in the entire db, insert / remove the item, and rewrite the db
  - Write our own code for efficiently tracking the location of new / deleted items
- As the db grows in size, these approaches become unscalable (impractical).

SQL: The “Heavyweight” Solution

- **S**tructured **Q**uery **L**anguage**:** An API for database management
- Basic data structure is a **table**
- As with most APIs, there are a handful of operations we use for nearly everything (“80/20 Rule”)
- Let’s look at some ordinary SQL examples, before we dive into Android’s SQLite API ….

SQL: Queries

SQL queries

The SQL queries are the most common and essential SQL operations. Via an SQL query, one can search the database for the information needed. SQL queries are executed with the "SELECT" statement. An SQL query can be more specific, with the help of several clauses:

- FROM - it indicates the table where the search will be made.
- WHERE - it's used to define the rows, in which the search will be carried. All rows, for which the WHERE clause is not true, will be excluded.
- ORDER BY - this is the only way to sort the results in SQL. Otherwise, they will be returned in a random order.

An SQL query example

```
SELECT * FROM
WHERE active
ORDER BY LastName, FirstName
```

https://www.ntchosting.com/encyclopedia/databases/structured-query-language/

SQL: Modifications

An example of an SQL INSERT

```
INSERT INTO phonebook(phone, firstname, lastname, address) VALUES('+1 123 456 7890',
'John', 'Doe', 'North America');
```

With the UPDATE statement, you can easily modify the already existing information in an SQL table.

An example of an SQL UPDATE

```
UPDATE phonebook SET address = 'North America', phone = '+1 123 456 7890' WHERE
firstname = 'John' AND lastname = 'Doe';
```

With the DELETE statement you can remove unneeded rows from a table.

An example of an SQL DELETE

```
DELETE FROM phonebook WHERE WHERE firstname = 'John' AND lastname = 'Doe';
```

https://www.ntchosting.com/encyclopedia/databases/structured-query-language/

SQLite vs. SQL

Situations Where SQLite Works Well

- **Embedded devices and the internet of things**

  Because an SQLite database requires no administration, it works well in devices that must ope consoles, cameras, watches, kitchen appliances, thermostats, automobiles, machine tools, air

  Client/server database engines are designed to live inside a lovingly-attended datacenter at th while providing fast and reliable data services to applications that would otherwise have dodgy

- **Application file format**

  SQLite is often used as the on-disk file format for desktop applications such as version control and so forth. The traditional File/Open operation calls sqlite3_open() to attach to the database superfluous. The File/Save_As menu option can be implemented using the backup API.

  There are many benefits to this approach, including improved application performance, reduce

- **Websites**

  SQLite works great as the database engine for most low to medium traffic websites (which is t its database. Generally speaking, any site that gets fewer than 100K hits/day should work fine demonstrated to work with 10 times that amount of traffic.

  The SQLite website (https://www.sqlite.org/) uses SQLite itself, of course, and as of this writing touching the database. Dynamic content uses about 200 SQL statements per webpage. This s 0.1 most of the time.

- **Data analysis**

Situations Where A Client/Server RDBMS May Work Better

- **Client/Server Applications**

  If there are many client programs sending SQL to the same database over a network, then us latency associated with most network filesystems, performance will not be great. Also, file locl work correctly, two or more clients might try to modify the same part of the same database at implementation, there is nothing SQLite can do to prevent it.

  A good rule of thumb is to avoid using SQLite in situations where the same database will be a network.

- **High-volume Websites**

  SQLite will normally work fine as the database backend to a website. But if the website is writ database engine instead of SQLite.

- **Very large datasets**

  An SQLite database is limited in size to 140 terabytes (2^{47} bytes, 128 tibibytes). And even if it maximum size of files to something less than this. So if you are contemplating databases of th multiple disk files, and perhaps across multiple volumes.

- **High Concurrency**

  SQLite supports an unlimited number of simultaneous readers, but it will only allow one writer database work quickly and moves on, and no lock lasts for more than a few dozen millisecond different solution.

https://sqlite.org/whentouse.html

SQLite for Android

Step 1: Define a schema (table configuration)

_id	UUID	title	date	solved
1	b625df88-bc83-4ff9-99d4-9b46936ac0ba	Stolen yogurt	Fri Nov 11 14:45:21 EST 2016	0
2	d7f2edda-a853-4a39-a898-09c82979d4af	Dirty sink	Mon Nov 14 12:05:15 EST 2016	1

SQLite for Android

Step 2: Create a Java class for the schema

```
public class CrimeDbSchema {
    public static final class CrimeTable {
        public static final String NAME = "crimes";

        public static final class Cols {
            public static final String UUID = "uuid";
            public static final String TITLE = "title";
            public static final String DATE = "date";
            public static final String SOLVED = "solved";
        }
    }
}
```

SQLite for Android

Step 3: Build your initial database:

1) Check whether db already exists.
2) If not, create it and its table(s) and initial data.
3) If exists, open it and see what version of CrimeDbSchema it has.
4) If old version, run code to upgrade to newer.

Subclass the SQLiteOpenHelper

```
public class CrimeBaseHelper extends SQLiteOpenHelper {
     private static final int VERSION = 1;
     private static final String DATABASE_NAME = “crimeBase.db”;

     public CrimeBaseHelper(Context context) {
          super(context, DATABASE_NAME, null, VERSION);
     }

     @Override
     public void onCreate(SQLiteDatabase db) {
          // left blank for now
     }

     @Override
     public void onUpgrade(SQLiteDatabase db, int oldVersion, int new Version) {
     }
}
```

Modify CrimeLab to use CrimeBaseHelper

```
public class CrimeLab {
     private static CrimeLab sCrimeLab;

     private List<Crime> mCrimes;
     private Context mContext;
     private SQLiteDatabase mDatabase;
     …
     private CrimeLab(Context context) {
          mContext = context.getApplicationContext();
          mDatabase = new CrimeBaseHelper(mContext).getWritableDatabase();
     }
```

Now you get the desired behavior automatically:

1) Attempt to open (private) file `/data/data/edu.wlu.cs.levy.android.criminalintent/databases/crimeBase.db`, creating it if it doesn't exist.
2) If it's being created, call `CrimeBaseHelper.onCreate(SQListDatabase)`, saving out the latest version number.
3) Otherwise, check the version number in the database. If the version number in `CrimeOpenHelper` is higher, call `CrimeBaseHelper.onUpgrade(SQListDatabase, int, int)`.

Fleshing out

CrimeBaseHelper.onCreate

```
@Override
public void onCreate(SQLiteDatabase db) {
    db.execSQL("create table " + CrimeTable.NAME + "(" +
            " _id integer primary key autoincrement, " +
            CrimeTable.Cols.UUID + ", " +
            CrimeTable.Cols.TITLE + ", " +
            CrimeTable.Cols.DATE + ", " +
            CrimeTable.Cols.SOLVED +
            ")"
    );
}
```

Writing to the Database

- Naturally, we will use yet another class, **`ContentValues`**, to do this
- Add to `CrimeLab.java` :

```
public class CrimeLab {
    …
    private static ContentValues getContentValues(Crime crime) {
        ContentValues values = new ContentValues();
        values.put(CrimeTable.Cols.UUID, crime.getId().toString());
        values.put(CrimeTable.Cols.TITLE, crime.getTitle());
        values.put(CrimeTable.Cols.DATE, crime.getDate.getTime());
        values.put(CrimeTable.Cols.SOLVED, crime.isSolved() ? 0 : 1);
        return values;
    }
```

Inserting and Updating Rows

```
public void addCrime(Crime c) {
    ContentValues values = getContentValues(c);

    mDatabase.insert(CrimeTable.NAME, null, values);
}
```

optimization parameter, rarely used

```
public void updateCrime(Crime crime) {
    String uuidString = crime.getId().toString();
    ContentValues values = getContentValues(crime);

    mDatabase.update(CrimeTable.NAME, values,
            CrimeTable.Cols.UUID + " = ?",
            new String[] { uuidString });
}
```

SQL code to avoid “injection” attacks

Pushing Updates in CrimeFragment

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    UUID crimeId = (UUID)
getArguments().getSerializable(ARG_CRIME_ID);
    mCrime = CrimeLab.get(getActivity()).getCrime(crimeId);
}

@Override
public void onPause() {
    super.onPause();

    CrimeLab.get(getActivity())
            .updateCrime(mCrime);
}
```

Reading from the Database

- Recall the SQL **SELECT** command

An SQL query example

```
SELECT * FROM
WHERE active
ORDER BY LastName, FirstName
```

- In the SQLite Java API, this is implemented as a (wait for it!) another class, called **`Cursor`** ...

Reading from the Database

`Cursor` API:

```
public Cursor query(
    String table,
    String[] columns,
    String where,
    String[] whereArgs,
    String groupBy,
    String having,
    String orderBy,
    String limit)
```

Reading from the Database

```
private Cursor queryCrimes(String whereClause, String[]
whereArgs) {
    Cursor cursor = mDatabase.query(
            CrimeTable.NAME,
            null, // Columns - null selects all columns
            whereClause,
            whereArgs,
            null, // groupBy
            null, // having
            null  // orderBy
    );

    return cursor;
}
```

Using a `CursorWrapper` for DRY (**D**o **N**ot **R**epeat)

`CursorWrapper` provides a convenient way of packing up the following code:

```
    String uuidString =
getString(getColumnIndex(CrimeTable.Cols.UUID));
    String title =
getString(getColumnIndex(CrimeTable.Cols.TITLE));
    long date =
getLong(getColumnIndex(CrimeTable.Cols.DATE));
    int isSolved =
```

In a new class **CrimeCursorWrapper.java:**

```
public class CrimeCursorWrapper extends CursorWrapper {

    public CrimeCursorWrapper(Cursor cursor) {
        super(cursor);
    }

    public Crime getCrime() {
        String uuidString = getString(getColumnIndex(CrimeTable.Cols.UUID));
        String title = getString(getColumnIndex(CrimeTable.Cols.TITLE));
        long date = getLong(getColumnIndex(CrimeTable.Cols.DATE));
        int isSolved = getInt(getColumnIndex(CrimeTable.Cols.SOLVED));

        Crime crime = new Crime(UUID.fromString(uuidString));
        crime.setTitle(title);
        crime.setDate(new Date(date));
        crime.setSolved(isSolved != 0);
        return crime;
    }
}
```