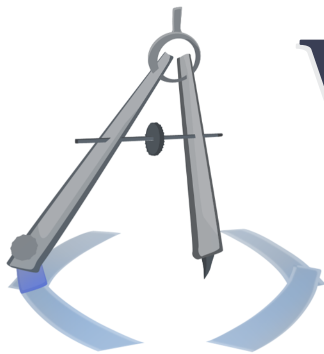

Website Architecture

Lezione 14

SQLite

Introducing the pocket-sized DBMS and its design implications, along with its general-purpose advantages.

Michael Serritella

Summer 2010

© Michael Serritella, 2010 • CIS4930: Website Architecture • The Florida State University

Intro to SQLite

SQLite is a DBMS which understands a large subset of SQL yet operates on a single file. It runs within the calling process instead of running as its own program. And thus it is a uniquely light, flexible product, which has some cool implications in website design.

SQLite is free and open-source, and it has been widely adopted by industry leaders of many fields.

Why it's novel

Until SQLite, DBMSes would run as their own program - a server program - and your PHP program would have to communicate with that program in a remote/long-distance fashion ("IPC": Inter-Process Communication). This has overhead in itself, and it's annoying to depend upon the existence of a DBMS server for even the simplest of tasks. With SQLite, you can design and build the database at home - or on nearly any computer - and then migrate it to the Web server. You can also squeeze a little more performance out of SQLite for simple tasks - like simple `SELECT`s, since you don't need to talk to a server. At the very least, it's a competitive speed and is apparently appropriate for websites with a hundred thousand to a million hits per day, depending on the types of operations used.

At low level, SQLite is actually implemented as a C library that you can *include* into your C program, and from there, you can simply make function calls. PHP is a C program and essentially does this, and this is how the API works.

Example execution

With SQLite, it is remarkably easy to get started. There's no server setup, the creation of DBMS users, Unix users, etc. The command-line program is a simple binary, where you optionally specify the database file path. Execution via an API, like PHP's PDO library, is equally simple.

Standalone console

Via the command-line prompt, simply type `sqlite3`, optionally `sqlite3 <databaseFilesystemPath>`. If you don't specify a path, you start with a **temporary, in-memory database**, which is pretty cool in itself. If you specify simply `sqlite`, then you get version 2, which is completely incompatible with version 3. You generally want version 3, since it is more featureful and is compatible with future versions.

Once you're in the terminal, you can make SQL commands or SQLite-specific commands. The SQLite commands start with a period (.) and do *not* end with a semicolon. For instance, here are some useful commands:

.read <path> Read the contents of an SQL file and apply it to the current database.

.tables Get a list of tables.

.schema [<object>] See the schema of the entire database or of a particular object. This shows you the exact SQL command used to create the object(s).

.headers [on|off Outputs the column names above a query result set (or not).

.quit Eject!

Via an API

Via an API, you would normally need to give connection options to the database, like a port, username, password, etc. Now, you only need the path to the SQLite database file. Or you can usually create an in-memory database through an API.

Here is an example using PHP's PDO:

PHP

```php
<?php
// Use an existing database:
$dbConnection = new PDO("sqlite:../database/theDB.sqlite");
$dbStatement = $dbConnection->prepare("SELECT * FROM Users;");

// Create an in-memory database:
$dbConnection = new PDO("sqlite::memory:");
?>
```

Tips & tricks

Just some pointers. Some of them may even be bad ideas.

Including comments in your schema

When you create a structural object in the database, like a table, view, trigger, etc., you should keep all your comments within the `CREATE` statement, like this:

SQL

```
CREATE VIEW AView AS
-- This is a view which does blah.
  SELECT P.Name AS Name,
         -- Compute the Body Mass Index.
         (P.Weight / P.Height) AS BMI
         FROM Etc;
```

If you do that, then all of the comments will appear when you fetch the schema of this view using `.schema` from the SQLite command line. If you put your main comments above the `CREATE`, those won't be visible.

Use "calculator" relations

Say you have a database of student grades, and you have the letter grades stored as 'A', 'A-', 'B+', etc. And you want to know the impact on a student's GPA, like 4.0, 3.75, or whatever. Storing this extra data would be redundant and only leaves open the chance of error. It's similar enough to our BodyMassIndex example, so let's incoroporate it into a view.

Well, we could lump it into a `CASE` command, which is like a switch statement. But let's assume that this mapping of ('A' ⇒ 4.0) is very large and even potentially variable. It would probably be obnoxious and unmaintainable to incorporate it into the definition of the view.

So let's build a new relation that, well, relates these two data sets. And if you can avoid it, why bother storing the value 4 in a table? It seems wasteful of disk space. Let's create a relation like this:

SQL

```sql
CREATE VIEW GradeGPAPointsCalculator AS
  SELECT 'A' AS LetterGrade, 4.0 AS GPAPoints
  UNION ALL
  SELECT 'A-' AS LetterGrade, 3.75 AS GPAPoints
  UNION ALL
  SELECT 'B+' AS LetterGrade, 3.25 AS GPAPoints
-- etc.
```

Depending on your tradeoffs - or your style - you may want to use this kind of thing. Let's call this beast a "calculator" relation, since it performs a kind of computation by relating or mapping values. Then, we can do this:

SQL

```sql
CREATE VIEW StudentGrades AS
  SELECT S.Name,
         S.Grade,
         -- Retrieve the value from the calculator.
         GGPC.GPAPoints AS GPAPoints
      FROM Students S
        LEFT JOIN GradeGPAPointsCalculator GGPC
          -- Ensure the proper calculator row is matched.
          ON S.Grade = GGPC.LetterGrade;
```

The literal encoding of these values in a view may or may not be a good idea, but you should generally use calculator relations in some way, even if they are tables. SQLite lacks procedural SQL functions, which would make this more straightforward.

Probably use the trigger/relational model

As we'll see, SQLite can't handle super-complex queries - they just won't work very quickly. You should profile your application first and see how it performs, but you should probably use the trigger-enhanced relational model to facilitate easy `SELECT`s when your website is running.

Feature set

SQLite offers a strong feature set, especially in terms of linguistic expressiveness; most of what you would want of SQL is supported.

What it does have

SQLite has all the usual, basic commands featured in the SQL lesson, like `CREATE TABLE`, `SELECT`, etc. It also has views, triggers, indices, user-defined functions (bound), at least two types of joins, support for correlated subqueries, (like with the `SELECT ... WHERE EXISTS` example) and more miscellany.

Untyped columns

One "feature" (they're really claiming this) is that you can store any type in any column. You can encode the columns as typed when you create the table, but you can insert anything in there. SQLite ostensibly stores most all data as a string (e.g. "45" instead of 45). This is obviously a double-edged sword - or at least a single-edged sword. So watch out.

Limited joins

SQLite does have joins, but you *may* want a little more than it has. Honestly, you probably won't. But it's possible. SQLite supports left joins and the most basic case of cross/inner joins. You can fake a right join by reordering a left join, and you can get a full join from unioning a left and right join. So, no big loss, but you may want to think in terms of left joins.

Binding user-defined functions

SQLite cannot define new SQL functions in the sense of defining new blocks of SQL code. SQLite can let you bind a function in your PHP program, or your C program, etc., to a new (or old) SQL function name. Your function can be a simple/scalar function or an aggregate function. See PHP's **documentation** for details.

Disadvantages

Of course, there are some downsides, namely due to the nature of SQLite executing within several potentially-concurrent programs and executing anew every time your program runs.

Untyped columns

SQLite has untyped columns - or at least any type may get thrown in there. So when you query from SQLite from within PHP, you should be careful that you only use data once it has been guaranteed to meet your type expectations. You should sanitize the data coming from a SQLite database or (somehow/magically) strictly control what goes into it.

Lack of persistent configuration

Since SQLite does not persistently run in the background like a normal DBMS, it has to recreate its working environment every time. And since it can't take too long to start up, it can't have *too* complex a working environment. Thus, you should probably avoid using components that require significant ini-

tialization before they are effective, like a large set of indices or the binding of a user-defined function (unless it's necessary).

Dealing with complex queries

Traditional DBMSes - or at least good ones - have a very complicated module for computing the best execution plan of a given query. SQLite is rather simple, and so it will often miss opportunities for optimization, some of which you might think are obvious. If you use a 6 10-layer structure of views that simply operate upon the same table but get more and more selective every time, other DBMSes might see through this depth, recognize that you're only operating on one table, and structure the query execution in a prudent way. SQLite can be overwhelmed by such depthwise optimizations, and so you probably shouldn't have a super-abstract schema which requires them.

Concurrency challenges

When concurrent programs want to access a file, if they are only reading, then it's OK. If one or more of them is writing, the operating system has to impose some exclusivity and order on the

requested operations. The solution is generally called "locking", where one process "has a lock" on a file. SQLite operates upon a simple file, where other DBMSes may lump everything into a handful of files and then manage locking internally. SQLite must go through the operating system to obtain a lock *on the entire file*. If multiple instances of programs using SQLite want to read from & write to a file, you may start feeling some slow-down.

General-purpose use

Outside the area of websites, SQLite is very useful for general-purpose applications.

For large groups of users

If your program manages other, smaller programs or manages a bunch of users, each of which needs its own database, you might want to simply give each user or program its own SQLite database. For instance, if a program manages plug-ins, like Firefox does, and its own plug-in API guarantees exclusive access to a database, it should probably have a SQLite database for each plug-in.

This may be quite advanced for us now, but in a website environment, you may have users who need to manage their own data, whether or not they access it as a database or just a "virtual folder" of files. For instance, if each user has a dropbox of files, you may want to manage their file metadata in a SQLite database rather than creating a username & password with a large DBMS server, which could hinder all your core operations.

For ad-hoc statistical analysis

If you have a data set in some funky, ad-hoc format, and you just want to make some queries on it, you can hardly do better than throwing it in a SQLite database - perhaps an in-memory database.

As a custom file type

What's perhaps best about SQLite for general purpose is that each application can have a SQLite database instead of a custom file format. SQLite databases are very compact, they take the correctness problem out of the equation, and they offer more expressive lookups and storage than a custom/hack file type. *And* they're cross-platform.

Website architecture implications

SQLite databases have a nice place in websites, though they aren't always appropriate. Here are generally the advantages of SQLite in a situation in which it is appropriate.

Simple backups

It is quite simple to back up a SQLite database. They are so small and well-contained, you can do a few cute things with them.. You can store them as entries in a larger database dedicated to backups - perhaps an SQLite database - or you can just email them to yourself once a day. Backups don't get easier than with SQLite.

Possibly simple collaboration

It always depends on your interpersonal dynamic, but it may be very easy for you to collaborate with people by exchanging copies of the database. This is related to its strength in backups; you can get easy version control, which facilitates rolling back the entire state of the database when one of your

team members screws it up. That's much easier than saying, "What did you do to the database?"/"Oh, I did some queries last night."

Economy of simple read operations

If your website simply reads from the database on every page request - which you should be doing 96% of the time - then SQLite is very fast, since it doesn't have the overhead of talking to a database server. At the very least, it is quite smooth for situations that simply read.

As a workaround for server congestion

With queries *and their entire result sets* being trafficked around your various servers - your Web server(s) and your database server(s) - the congestion of your local network can become a concern. Since SQLite executes in the Web server process, that communication is avoided and the overall traffic is that much lighter.

Furthermore, since SQLite executes in the Web server process, the database server will simply have less work to do.

Problem: RAM usage

But.. since SQLite executes in the Web server process, you may have a problem with RAM consumption. Web servers typically don't allocate a lot of RAM to each request, and if SQLite is making a RAM-intensive query, it may break that threshold. This can be frustrating when you only need to execute a particular query once - not even for every request - and you still can't do it. If your queries take a couple minutes to execute (perhaps for maintenance; it's possible), then it's a good sign that it's teetering on the edge of your RAM allowance. This is a very application- and server-configuration-specific problem, but know that it is possible.