CS 130B
SQLite

Douglas Putnam

Week 8

Abstract

In this lecture we will learn about SQLite, a powerful RDBMS bundled with PHP 5. SQLite exists wherever PHP 5 exists. It is fast and furious.

May you do good and not evil.
May you find forgiveness for yourself and forgive others.
May you share freely, never taking more than you give.
— D. Richard Hipp

Contents

1 SQLite

There are times when you have modest data storage needs and a full-fledged database is either not available, or overkill, and feel simple flat file will do the job. You will quickly find that managing data in flat file becomes complicated very quickly. PHP 5 provides a bundled relational database that is perfectly suited for light to moderate database needs: SQLite.

SQLite differs from most databases in that it is not running as a separate process: it is embedded in PHP and runs only when your script is running. SQLite is available to any of your PHP scripts that have permission to read and write from the filesystem. SQLite supports almost all of SQL92, the SQL standard specification, and has some extra features missing from MySQL: transactions, query using subselects, user-defined functions, and triggers.

In this lecture you will learn

- The strong points and weakness of SQlite
- How to decide whether to use SQLite or another database
- How to create and delete SQLite databases
- How to query and fetch from a SQLite database
- How to create robust applications using SQLite

1.1 Strong Points

This section describes SQLite's features as compared to other databases.

1.1.1 Self-Contained

SQLite is self-contained. No server is required. SQLite does not use the client/server model. It is embedded in your application and requires only access to the filesystem. Because SQLite is embedded in PHP, you application has no dependency on external services (databases).

1.1.2 Easy to Get Started

Setting up a new database with SQLite is easy. You do not require the intervention of a database administrator.

1.1.3 Bundled with PHP 5

The entire SQLite engine is contained in PHP 5. There is no need for you, or anyone who uses the application, to have access to a special database.

1.1.4 Lightweight and Fast

SQLite is lean and efficient. It is a new program with little backwards-compatibility baggage. For most queries, SQLite is equal to or exceeds the speed of MySQL.

1.1.5 Procedural and Object-Oriented Interface

PHP's SQLite extension provides both a procedural and OO interface. The OO interface helps reduce the number of lines of code, and in many cases provides better performance than the procedural interface.

1.2 Weak Points

Most of SQLite's weak points are the result of SQLites strong points.

1.2.1 No Server Process

Although this is also one of SQLite's strong points, the fact that SQLite has no server process means that external users cannot connect to the database via sockets. Also, because there is no persistent SQLite process, file locking and concurrency issues arise.

The only way external users can share the database is to share the system with the database file. This way of running remote queries is much slower that using network sockets to send queries and responses, and less reliable.

1.2.2 Not Binary Safe

SQLite does not handle binary data (images, executables, etc) natively: they must be encoded before an INSERT and decoded after a SELECT.

1.2.3 Transactions Lock All Tables

Most databases lock individual tables (or even only rows) during transactions, but because of its design, SQLite locks the *whole* database on inserts, which makes concurrent read/write operations noticeably slow.

1.3 When to use SQLite

SQLite's main virtue is that it is a stand-alone database that is well-suited to building web sites. Because SQLite works on files, there's not need to maintain a set of database credentials; if you can write to a file, you can run SQLite, and you can make changes to the database. Hosting companies need to support the SQLite extensions and programmers can do the rest.

SQLite excels at stand-alone applications where there are many read queries and few write queries. Such an application would be a the code that displays manual pages of the PHP.net, many of which are stored in a SQLite database.

2 The PHP Interface

In this section you will meet most of SQLite's feature set. In the examples that follow, we will use the OO-base API, but we will also mention the procedural equivalents.

2.1 Setting Up SQLite Databases

A SQLite database is a specially formatted file on the system. There is no need to log in to a running daemon process to set up a database with SQLite. To create a database, you simply try to open one; if the database does not exist, SQLite will create it for you. When you create a database, you can specify the file permissions for the resulting file.

Opening and Closing Databases

`$sqlite = new SQLiteDatabase(...)` Connects the script to a SQLite database, or creates one if none exists yet. Parameters:
* The path and file name (string)
* Permissions in UNIX chmod style (octal number)
* Error message (by-reference, string)

`sqlite_open(...)` Connects the script to a SQLite database, or creates one if none exists yet. Parameters:
* The path and file name (string)
* Permissions in UNIX chmod style (octal number)
* Error message (string)

`sqlite_close(...)` Disconnects the script from the SQLite database connection. The parameter is the SQLite database descriptor (resource)

For example, to create the `vinyl` database

```
<?php
$db = new SQLiteDatabase(”vinyl.db”, 0666, $error) or die(”Failed: $error”);
?>
```

2.1.1 Simple Queries

After the database has been created, we can start executing queries on the database. We must first create tables.

```
<?php
$create_users_tables = ”
CREATE TABLE users(
    users_id,
    name,
    password,
    reminder,
    email,
    last_ip,
    last_host,
    status,
    deleted,
    deleted_date,
    created_date,
    modified_date,
    last_login
    );

CREATE TABLE admin_areas (
    admin_area_id,
    name,
    path
);

CREATE TABLE admin_permissions(
    admin_perm_id,
    admin_area_id,
    users_id,
```

```
    );

CREATE UNIQUE INDEX users_id ON users(users_id);
";

@$db->query($create_users_tables);
?>
```

Notice that the `CREATE TABLE` statement looks quite a bit different from what you would see with other databases: there are no data types in the field definitions. SQLite has only two types: `INTEGER`, which is used to store numbers, and "something else", which is something like a `VARCHAR` except that it can store more than 255 characters, which is sometimes a limitation of this data type in other databases. An `INTEGER` field can become an auto-increment type if it is created with an added "`PRIMARY KEY`" in the field definition. You can do this for one field per table.

The second thing you may have noticed is that we can execute multiple `CREATE TABLE` statement with one function call to the `query()` method. This is often not possible with othere database systems, such as the `mysql` extensions (but `mysqli` can do this).

2.1.2 Inserting Data

Add new rows to the database using `INSERT` and `query()`;

```
$Sqldb->query("INSERT INTO admin_areas values(1, 2, 'users')");
```

To avoid problems with single quotes, SQL injection, and other special characters, we can use `escapeString()`.

```
$safe = $Sqldb->escapeString($_GET['area']);
$Sqldb->query("INSERT INTO admin_areas values(1,2,$safe)")
```

2.1.3 Retrieving Data

To retrieve data, use '`SELECT`' with the `query()` function, the iterate through the result set.

```
$res = $Sqldb->unbufferedQuery('SELECT name from users');
while($aRow = $res->fetch()) {
    // do something
}
```

By default SQLite returns both and indexed and associative array for each row of data.

2.2 Iteration

2.2.1 Object Iterators

SQLite takes advantage of a PHP 5 feature that lets you access rows from your database query as though they're just elements from an array. This feature is called *iteration*.

SQLite does not pre-fetch all rows and store them as a hidden array. Rather, it returns a row from your query as if the row was already present in a results array:

```
// one at a time
$res = $Sqldb->query('SELECT * FROM users');
foreach($res as $aRow) {
    // do something cool here with $aRow
}
```

You can also embed the query inside the `foreach` loop:

```
// one at a time
foreach($Sqldb->unbufferedQuery('SELECT * FROM users') as $aRow) {
    // do something cool here with $aRow
}
```

This syntax works only with `foreach`. When interating over a query in this manner, it is best to use `unbufferedQuery()`, which has some efficiency gains over the more complex `query()` function.

In fact, when converting table data to XML or HTML tables, the `unbufferedQuery()` function allows the data to flow directly from the database into your script with the least amount of overhead.

If you prefer to work with the procedural interface, here is a list of SQLite functions for both the procedural and object-oriented APIs. You will notice that the procedural functions require at least one argument—a SQLite database resource, while the OO interface already contains that resource.

SQLite Functions

Procedural name	Object-oriented name
`$db = sqlite_open($table)`	`$db = new SQLiteDatabase($table)`
`sqlite_close($db)`	`unset($db)`
`$r = sqlite_query($db, $sql)`	`$r = $db->query($sql)`
`$r = sqlite_single_query($db, $sql)`	`$r = $db->singleQuery($db, $sql)`
`$r = sqlite_query_array($db, $sql)`	`$r = $db->arrayQuery($sql)`
`$r = sqlite_query_unbuffered($db, $sql)`	`$r = $db->unbufferedQuery($sql)`
FETCHING FUNCTIONS and METHODS	
`$array = sqlite_fetch_array($r)`	`$array = $r->fetch()`
`$array = sqlite_fetch_all($r [,MODE])`	`$array = $db->fetchAll($sql [,MODE])`
`$scalar = sqlite_fetch_single($r)`	`$scalar = $r->fetchSingle()`
`$safe = sqlite_escape_string($s)`	`$safe = $db->escapeString($s)`
`$id = sqlite_last_insert_rowid($r)`	`$id = $db->lastInsertRowid($r)`

The `MODE` for `fetchAll()` may be `SQLITE_NUM`, `SQLITE_ASSOC`, or `SQLITE_BOTH`.

The naming scheme for the procedural functions prepends each function with 'sqlite_' and uses the '_' character between words. The OO interface does not use the 'sqlite_' prefix, and the methods are named using camel-back notation.

SQLite versus MySQL

SQLite Function	MySQL equivalent function
`sqlite_open()`	`mysqli_connect()`
`sqlite_close()`	`mysqli_close()`
`sqlite_query()`	`mysql_query()`
`sqlite_fetch_array()`	`mysqli_fetch_row()`
`sqlite_fetch_array()`	`mysqli_fetch_assoc()`
`sqlite_num_rows()`	`mysqli_num_rows()`
`sqlite_last_insert_rowid()`	`mysqli_insert_id()`
`sqlite_escape_string()`	`mysqli_real_escape_string()`

3 Robust SQLite Applications

Now that we have covered the basics, we can look at all of the features we need to create robust applications using SQLite—such features as creating primary and other keys, using in-memory tables, transactions, and error-handling. All of these are needed to keep a site up and out of trouble.

3.1 Indexes

Adding an *index*, also called a *key*, is the easiest way to improve database performance. If SQLite performs a search without keys, it must search every row of the table looking for matches. However, if an index has been applied to a table, SQLite can use a specially constructed lookup table to speed searches dramatically.

If you know in the planning stages that you will be performing searches based on a particular unique field in a table, you should declare the field UNIQUE in the creation of the table. SQLite will create a key on this field and create an index. When a field is marked as `UNIQUE`, the database will prevent duplicate entries for this field.

```
CREATE TABLE users (user_email TEXT UNIQUE, password TEXT);
```

To add an index to an existing SQLite table, use the `CREATE INDEX` statement:

```
CREATE INDEX users_name_idx on users(name);
```

An index can have any name, but practical names are informative, indicating the table and field the index applies to. A simple index does not require a field to be unique. However, some fields—a date field, for example—may be used for quick searches.

You can add a `UNIQUE` key to an existing table:

```
CREATE UNIQUE INDEX users_email_idx on users(email);
```

To remove and index, you can use the `DROP` command:

```
DROP INDEX indexname;
```

Indexes make your database larger, but there is no harm in keeping an index around, even if it is not used.

3.2 Primary Keys

A *primary key* is a special kind of index. When a column is defined as primary key, it becomes a unique identifier for a row. If you need to fetch the information for a row, use the primary key in your query.

In SQLite, primary keys must be fields defined as integers. This field will be incremented automatically, much the same as a MySQL auto_increment field. SQLite will insert a '1' for the first row, a '2' for the second, and so on. If you delete a record, SQLite will preserve that hole in the sequence and add any new records to the end of the table instead of filling empty rows.

To create a primary key, define a column as an `INTEGER PRIMARY KEY`.

```
CREATE TABLE users ( userid INTEGER PRIMARY KEY,
                     user_name TEXT UNIQUE,
                     password TEXT );
```

When you add a row to a table, pass NULL as the value of the primary key:

```
INSERT INTO users VALUES (NULL, 'Harry', 'yrraH');
```

To assign a specific number to a primary key field, pass that number instead of `NULL`.

```
INSERT INTO users VALUES (999999, 'root', 'drowssap');
```

`lastInsertRowId()` and `sqlite_last_insert_rowid()` To find the value of the primary key of the last row inserted, use `lastInsertRowId()` (if using the object-oriented interface) or `sqlite_last_insert_rowid()` if using the procedural interface.

```
$db = new SQLiteDatabase('/tmp/users.db');
$sql = "INSERT INTO user VALUES(NULL,'$username','$password')";
$db->query($sql);
$rowid = $db->lastInsertRowId();
```

Using the `lastInsertRowId()` is better than writing a query that looks for the largest value in the primary key column because it is quite possible that another user may alter the table.

3.3 Error Handling

One of the fundamental problems of every application is how to handle errors gracefully. SQLite does not have slick error handling. SQLite's error handling can produce inappropriate error message because each of the query functions can throw warnings. It is important to stifle SQLite a little by prepending the *shut up* operator (`@`) to all of the query functions.

Although SQLite does throw exceptions in the object-oriented constructor, we must otherwise check the result of each query against `FALSE` to see if the query succeeded. If the query fails, we need to use the `lastError()` and `errorString()` methods to retrive a textual description of the error. These error messages are quite terse and uninformative.

SQLite's constructor might also throw an SQLiteException, which you need to handle yourself with a `try..catch` block. SQLite's error handling will be improving, most likely, in future versions of PHP.

3.4 Transactions

Transactions are part of designing robust database applications. Transactions can be started by executing a query with a simple 'BEGIN' or a 'BEGIN TRANSACTION'. If an error occurs during a sequence of operations, SQLite transactions allow us to back out of all the changes that have been made from the `BEGIN` statement by using a 'ROLLBACK' query, or we may `'COMMIT'` the changes to the database if the query is successful.

Yet another advantage of transactions is that multiple operations can be performed more efficiently in a single transaction. The more operations SQLite handles per request, the greater its efficiency. The fact is that SQLite is dismally slow when making inserts.

In this script, for example, 880 inserts take about 1:20 minutes when inserts are done sequentially. But the same number of inserts made within the framework of a transaction takes only 3 seconds, more than 30 times faster a series of normal queries.

As always there is a disadvantage for every advantage. In the case of transactions, the price is that transactions, by design, lock the entire database when making changes to a row or field. This is necessary to prevent processes from changing the database simultaneously and corrupting data. When using long running transactions, your users may experience a sluggishness.

Transaction Example In the following example, we will load data from a MySQL database into the SQLite `vinyl` database. This example uses the object-oriented interface.

```
<?php

try {
    $Sqldb = new SQLiteDatabase("vinyl.db", 0666, $error);
}
catch(SQLiteException $e) {
    if($e) {
        die ('Could not connect to database: '.$e->getMessage());
    }
}
```

First we try to open the vinyl.db SQLite database. Because SQLite throws exceptions when we use the OO interface, we can put our connection code into a `try/catch` block.

```
// Use DB to connect to MySQL
include 'DB.php';
$db = DB::connect('mysqli://newguy:NEwGUy@localhost/newguy');
if(PEAR::isError($db)) {
    die("Could not connect: ". $db-getMessage());
}
```

Our next step is to open the MySQL database use DB.

Once we have this database open, we can get all of the records from the `newguy.users` table using the DB `getAll()` function, which returns all of the rows in a result set to an array. We will use this array to generate the `INSERT` statements that will put the MySQL data into the SQLite `vinyl.db`.

```
// use the DB getAll() method to place the result set into an array
$aAll = $db->getAll('select * from newguy.users');

// start the transaction (for speed and file locking);
try {
    $Sqldb->query('BEGIN TRANSACTION');
    print 'Inserting User Data'."\n";
    print 'Inserting User Data'."\n";
    foreach($aAll as $k => $v) {
        // escape any problematic characters
        foreach($v as $kk => $vv) { $v[$kk] = sqlite_escape_string($vv); }
        $insert = "INSERT INTO users values('" . join("','",$v) . "');";
        $res = $Sqldb->query($insert);
        if(!$res) {
            throw new Exception("\n".'Could not insert into admin_permissions table: '.
            sqlite_error_string($Sqldb->lastError()));
        }
        // show progress
        print ".";
    }
```

Notice the use of the `escapeString()` function to properly quote all of the entries into the database. You may be tempted to use `addslashes()`, as you might do with MySQL, but this function does not properly quote data for SQLite.

```
    $aAll = $db->getAll('select * from newguy.admin_areas');
    print 'Inserting Admin Area Data'."\n";
    foreach($aAll as $k => $v) {
        // escape any problematic characters
        foreach($v as $kk => $vv) { $v[$kk] = sqlite_escape_string($vv); }
        $insert = "INSERT INTO admin_areas values('" . join("','",$v) . "');";
        $res = $Sqldb->query($insert);
        if(!$res) {
            throw new Exception("\n".'Could not insert into admin_permissions table: '.
            sqlite_error_string($Sqldb->lastError()));
        }
    print ".";
    }
    $aAll = $db->getAll('select * from newguy.admin_permissions');
    print 'Inserting Admin Permission Data'."\n";
    foreach($aAll as $k => $v) {
        // escape any problematic characters
        foreach($v as $kk => $vv) { $v[$kk] = sqlite_escape_string($vv); }
        $insert = "INSERT INTO admin_permissions values('" . join("','",$v) . "');";
        $res = $Sqldb->query($insert);
        if(!$res) {
            throw new Exception("\n".'Could not insert into admin_permissions table: '.
            sqlite_error_string($Sqldb->lastError()));
        }
        print ".";
```

```
    }
}
catch (Exception $e) {
    print $e->getMessage();
    print "\n".'ROLLING BACK';
    $Sqldb->query('ROLLBACK');
    exit;
}
```

This program end with the `catch` block. SQLite, when running in the OO mode, will throw a `SQLiteException` whenever an error occurs in the constructor! This means that we must handle all other errors the old-fashioned way: by checking the return result.

4 In-Memory Tables

For extremely fast access, SQLite supports storing table in RA instead of on the disk. These tables do not persist between requests, so you cannot create them and refer to them again and again. These tables need to be created each time the page loads, so they are best suited to scripts that load large amounts of data into the program at the beginning and make a series of requests. Such a program would be one that generates a number of reports from a large data set.

To use in-memory tables, pass the token `:memory:` as the database name:

```
$db = new SQLiteDatabase(':memory:');
$db->query('CREATE TABLE ...');
```

5 User-Defined Functions

In addition to the SQL standard functions such as `lower()` and `upper()`, SQLite allows users to include their own functions written in PHP. The functions are known as *user-defined functions*, or *UDFs*. When you create a UDF, you embed your logic into SQLite and avoid the need to perform the task in you PHP script. The advantage of UDFs is that you can take advantage of all the features in the database, such as sorting and finding data.

There are two kinds of UDFs: standard and aggregate functions.

5.1 Standard UDFs

Standard UDFs are one-to-one functions: when given a single row of data, they return a single result. UDFs can change case, create cryptographic hashes, compute sales tax on items in a shopping cart, and so on.

```
<?php
$db = new SQLiteDatabase('vinyl.db');

// function to turn data rows into HTML rows
function data2cell($sVal)
{
    return '<td>'.$sVal.'</td>';
}

// functions to create HTML row tag
function tr()
{
    return '<tr>';
```

```
}
function tr_end()
{
    return '</tr>';
}

// add functions to SQLite
$db->createFunction('td','data2row',1);
$db->createFunction('tr','tr',0);
$db->createFunction('endtr','endtr',0);
$row = $db->query("SELECT tr() || td(name) || td(email)
                  || tr_end() from users WHERE name like 'a%' limit 10");

if(!$row) {
    die(sqlite_error_string($db->lastError()));
}
foreach($row as $aRow) {
    print $aRow[0] . "\n";
}
?>
```

5.2 Aggregate functions

Aggregate UDFs, by contrast, are many-to-one: they receive multiple rows of data and return a single value.

SQLite provides many built-in aggregate functions, most of which deal with statistics such as the count (total) rows, max value, min value, mean value, and so on. Here's an example of an aggregate function taken from the PHP manual)

Example 1. max_length aggregation function example

```
<?php
$data = array(
   'one',
   'two',
   'three',
   'four',
   'five',
   'six',
   'seven',
   'eight',
   'nine',
   'ten',
   );
$dbhandle = sqlite_open(':memory:');
sqlite_query($dbhandle, "CREATE TABLE strings(a)");
foreach ($data as $str) {
    $str = sqlite_escape_string($str);
    sqlite_query($dbhandle, "INSERT INTO strings VALUES ('$str')");
}
function max_len_step(&$context, $string)
{
    if (strlen($string) > $context) {
        $context = strlen($string);
```

```
    }
}
function max_len_finalize(&$context)
{
    return $context;
}
sqlite_create_aggregate($dbhandle, 'max_len', 'max_len_step', 'max_len_finalize');
var_dump(sqlite_array_query($dbhandle, 'SELECT max_len(a) from strings'));
?>
```

6 Triggers

SQLite has some advanced features, including triggers. Triggers can be set to act whenever specific queries modify the database. Triggers are set using this syntax:

```
$trigger_query = "CREATE TRIGGER users_trigger
AFTER INSERT ON admin_permissions
BEGIN
UPDATE users SET modified_date = now()
WHERE user_id = $user_id
END;";
$db->query($trigger_query);
```

7 Views

The last thing to discuss are **views**, a SQL feature that simplifies user queries. For example, we can create a view of the users table that restricts the data the user see to the user name and email address.

```
$view = "CREATE VIEW users_name_email AS
SELECT name, email FROM users";
$db->query($view);
$aAll = $db->fetchAll('SELECT * from users_name_email limit 10', SQLITE_NUM);
```