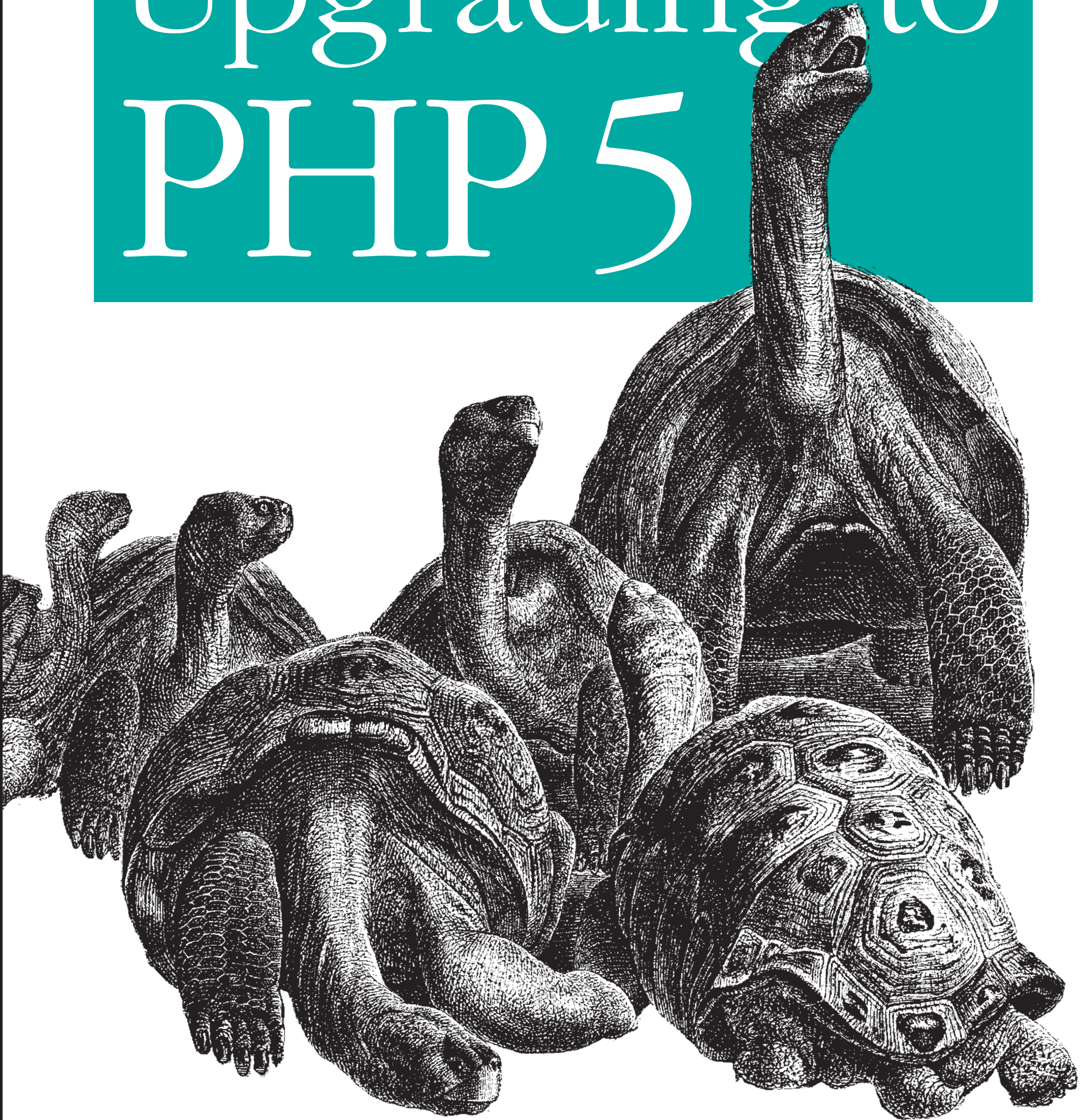
All That's New in PHP 5

Covers MySQL 4.1

Upgrading to PHP 5



O'REILLY®

Adam Trachtenberg

CHAPTER 4

SQLite

Substituting text files for a database is like cutting a fish with a hammer. You might get it to work, but it's going to be a really messy process. When your application needs a server-side storage mechanism but you can't rely upon the presence of a specific database, turn to SQLite. It correctly handles locking and concurrent accesses, the two big headaches with home-brewed flat files.

Since the SQLite database is bundled with PHP 5, now every PHP 5 script can read, write, and search data using SQL. SQLite differs from most databases because it is not a separate application. Instead, SQLite is an extension that reads from and writes to regular files on the hard drive. Any PHP users who have permission to manipulate files can use SQLite to edit a database, just like they can use GD to edit images.

Although the name SQLite hints at a less than full-featured product, SQLite actually supports almost all of SQL92, the SQL standard specification. Besides the usual `INSERT`s and `SELECT`s, with SQLite you can also use transactions, query using subselects, define your own functions, and invoke triggers.

SQLite actually performs most actions more quickly than many other popular databases. In particular, SQLite excels at `SELECT`ing data. If your application does an initial (or periodic) data `INSERT` and then reads many times from the database, SQLite is an excellent choice. The PHP web site uses SQLite to handle some forms of searches.

Unfortunately, SQLite has some downsides. Specifically, when you update the database by adding new data, SQLite must lock the entire file until the alteration completes. Therefore, it does not make sense in an environment where your data is constantly changing. SQLite does not have any replication support, because there's no master program to handle the communication between the master database and its slaves.

Additionally, SQLite has no concept of access control, so the `GRANT` and `REVOKE` keywords are not implemented. This means you cannot create a protected table that only certain users are allowed to access. Instead, you must implement access control by using the read and write permissions of your filesystem.

SQLite is not for sites that are flooded with heavy traffic or that require access permissions on their data. But for low-volume personal web sites and small business intranet applications, SQLite lets you do away with the burden of database administration. SQLite is also perfect for log file analysis scripts and other applications that benefit from a database but whose authors don't want to require the user to install one. SQLite is bundled with PHP 5, so unless it has been specifically omitted, it's part of every PHP 5 installation.

The SQLite home page (*http://www.sqlite.org/*) has more details about SQLite's features, limitations, and internals. A list of PHP's SQLite functions is online at *http://www.php.net/sqlite*.

This chapter starts off with SQLite basics: creating databases, passing SQL queries to SQLite, and retrieving results—everything you need to start using SQLite. It then moves on to alternative SQLite retrieval functions and interfaces, including a nifty object-oriented interface. After covering how to talk with SQLite, this chapter shows how to improve SQLite performance with indexes and how to gracefully handle errors. It closes with a few advanced features: transactions and user-defined functions, which help keep your data consistent and extend SQLite, respectively.

SQLite Basics

It's easy to get up and running with SQLite. Its design eliminates the need for any configuration variables, such as a database server name or a database username and password. All you need is the name of a file where the data is stored:

```
$db = sqlite_open('/www/support/users.db');
sqlite_query($db, 'CREATE TABLE users(username VARCHAR(100),
                                      password VARCHAR(100))');
```

This creates a `users` table stored in the database file located at */www/support/users.db*. When you try to open a database file that doesn't already exist, SQLite automatically creates it for you; you don't need to execute a special command to initialize a new database.

If you cannot seem to get SQLite to work, make sure you have both read and write permission for the location on the filesystem where you're trying to create the database.

SQLite has even fewer data types than PHP—everything's a string. While you *can* define a column as INTEGER, SQLite won't complain if you then INSERT the string PHP into that column. This feature (the SQLite manual declares this a feature, not a bug) is unusual in a database, but PHP programmers frequently use this to their advantage in their scripts, so it's not a completely crazy idea. A column's type matters only when SQLite sorts its records (what comes first: 2 or 10?) and when you enforce UNIQUEness (0 and 0.0 are different strings, but the same integer).

The table created in this example has two columns: username and password. The columns' fields are all declared as VARCHARs because they're supposed to hold text. Although it doesn't really matter what type you declare your fields, it can be easier to remember what they're supposed to hold if you give them explicit types.

Inserting Data

Add new rows to the database using INSERT and sqlite_db_query( ):

```
$username = sqlite_escape_string($username);
$password = sqlite_escape_string($password);

sqlite_query($db, "INSERT INTO users VALUES ('$username', '$password')");
```

You must call sqlite_escape_string( ) to avoid the usual set of problems with single quotes and other special characters. Otherwise, a password of abc'123 will cause a parser error. Don't use addslashes( ) instead of sqlite_escape_string( ), because the two functions are not equivalent.

Retrieving Data

To retrieve data from an SQLite database, call sqlite_query( ) with your SELECT statement and iterate through the results:

```
$r = sqlite_query($db, 'SELECT username FROM users');
while ($row = sqlite_fetch_array($r)) {
    // do something with $row
}
```

By default, sqlite_fetch_array( ) returns an array with the fields indexed as both a numeric array and an associative array. For example, if this query returned one row with a username of rasmus, the preceding code would print:

```
Array (
    [0] => rasmus
    [username] => rasmus
)
```

As you can see, sqlite_fetch_array( ) works like mysqli_fetch_array( ).

When you're using user-entered data in a WHERE clause, in addition to calling sqlite_escape_string( ), you must filter out SQL wildcard characters. The easiest way to do this is with strtr( ):

```
$username = sqlite_escape_string($_GET['username']);
$username = strtr($username, array('_' => '\_', '%' => '\%'));

$r = sqlite_query($db,
                  "SELECT * FROM users WHERE username LIKE '$username'");
```

Use sqlite_num_rows( ) to find the total number of rows returned by your query without iterating through the results and counting them yourself:

```
$count = sqlite_num_rows($r);
```

You can call sqlite_num_rows( ) without retrieving the results from SQLite. Remember, this function takes the query result handle, like sqlite_fetch_array( ).

If speed is a concern, use sqlite_array_query( ). This retrieves all the data and puts it into an array in a single request:

```
$r = sqlite_array_query($db, 'SELECT * FROM users');
foreach ($r as $row) {
    // do something with $row
}
```

However, if you have more than 50 rows and only need sequential access to the data, use sqlite_unbuffered_query( ):

```
$r = sqlite_unbuffered_query($db, 'SELECT * FROM users');
while ($row = sqlite_fetch_array($r)) {
    // do something with $row
}
```

This is the most efficient way to print items in an XML feed or rows in an HTML table because the data flows directly from SQLite to your PHP script without any overhead tracking behind the scenes. However, you can't use it with sqlite_num_row( ) or any function that needs to know the "current" location within the result set.

When you are done with the connection, call sqlite_close( ) to clean up:

```
sqlite_close($db);
```

Technically, this is not necessary, since PHP will clean up when your script finishes. However, if you open many SQLite connections, calling sqlite_close( ) when you're finished reduces memory usage.

SQLite Versus MySQL

The SQLite function names are similar to the MySQL functions, but not identical. Table 4-1 provides a side-by-side comparison of the two.

Table 4-1. Comparison of major MySQL and SQLite function names

MySQL	SQLite
`mysqli_connect( )`	`sqlite_connect( )`
`mysqli_close( )`	`sqlite_close( )`
`mysqli_query( )`	`sqlite_query( )`
`mysqli_fetch_row( )`	`sqlite_fetch_array( )`
`mysqli_fetch_assoc( )`	`sqlite_fetch_array( )`
`mysqli_num_rows( )`	`sqlite_num_rows( )`
`mysqli_insert_id( )`	`sqlite_last_insert_rowid( )`
`mysqli_real_escape_string( )`	`sqlite_escape_string( )`

Alternate SQLite Result Types

SQLite has many different functions for retrieving data. The ones you've already seen are not the only ones at your disposal, and you can control whether `sqlite_fetch_array( )` returns numeric arrays, associative arrays, or both.

By default, when `sqlite_fetch_array( )` returns data, it provides you with an array containing numeric *and* associative keys. This is a good thing, because it lets you refer to a column either by its position in the `SELECT` or by its name:

```
$r = sqlite_query($db, 'SELECT username FROM users');
while ($row = sqlite_fetch_array($r)) {
    print "user: $row[username]\n";  // this line and...
    print "user: $row[0]\n";         // this line are equivalent
}
```

This is also a bad thing because it can catch you unawares. For example:

```
$r = sqlite_query($db, 'SELECT * FROM users');
while ($row = sqlite_fetch_array($r)) {
    foreach ($row as $column) {
        print "$column\n";           // print each retrieved column
    }
}
```

This actually displays every column *twice*! First it prints the value stored in `$row[0]`, and then it prints the same value referenced by its column name. If you have a generalized table-printing routine where you don't know the number of fields in advance, you might fall prey to this bug.

Additionally, if you retrieve a large dataset from SQLite, such as an entire web page or an image, then each result takes up twice as much memory because there are two copies stashed in the array.

Therefore, SQLite query functions take an optional parameter that controls the results. Pass SQLITE_ASSOC for only column names, SQLITE_NUM for only column positions, and SQLITE_BOTH for the combination. These arguments are constants, not strings, so you do not place them in quotation marks. For example:

```
// numeric
$row = sqlite_fetch_array($r, SQLITE_NUM);

// associative
$row = sqlite_fetch_array($r, SQLITE_ASSOC);

// both (the default value)
$row = sqlite_fetch_array($r, SQLITE_BOTH);
```

SQLite returns column names in the same mixed case as you CREATEd them. This is not true of all databases. Some like to use all uppercase letters; others turn everything into lowercase. When porting applications from one of these databases to SQLite, use the sqlite.assoc_case configuration parameter to maintain compatibility without rewriting your code. The default value is 0, for mixed case; changing it to 1 turns the strings in your associative arrays to uppercase, whereas 2 sets them to lowercase. Modifying the column names slows down SQLite slightly, but PHP's strtolower( ) is significantly worse in this regard.

Object-Oriented Interface

The SQLite extension allows you to interact with SQLite in an object-oriented manner. SQLite's OO interface turns your database connection into an object and lets you call methods on it. When using this interface, there's no need to pass in a database handle to any SQLite functions, because the object knows what database connection it should use.

Additionally, the SQLite OO interface lets you iterate directly over queries inside a foreach without needing to call fetch_array( ). PHP will automagically request the appropriate row from SQLite and then stop the loop when you've read all the rows.

Using the SQLiteDatabase Object

To use the OO interface, instantiate a new SQLiteDatabase object and call methods on it. Example 4-1 uses this interface to connect to the database */www/support/users.db* and SELECT all the rows from the users table.

Example 4-1. Using the SQLite object-oriented interface

```
$db = new SQLiteDatabase('/www/support/users.db');

// one at a time
$r = $db->query('SELECT * FROM users');
while ($row = $r->fetch( )) {
    // do something with $row
}

// all at once
$r = $db->arrayQuery('SELECT * FROM users');
foreach ($r as $row) {
    // do something with $row
}

unset($db);
```

All procedural SQLite functions are available under the object-oriented interface, but their names are not identical. For one, you must remove the leading `sqlite_` from the function name. Also, names use studlyCaps instead of underscores.

Additionally, you don't pass in the database link identifier, since that's stored in the object. So, `sqlite_query($db, $sql)` becomes `$db->query($sql)`, and so forth.

The major exception to these rules is `sqlite_close( )`. To end the connection when using the OO interface, delete the object by using `unset( )`.

Table 4-2 contains a list of frequently used SQLite functions and their object equivalents.

Table 4-2. 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)`
`$r = sqlite_query_array($db, $sql)`	`$r = $db->arrayQuery($sql)`
`$r = sqlite_query_unbuffered($db, $sql)`	`$r = $db->unbufferedQuery($sql)`
`sqlite_fetch_array($r)`	`$r->fetch( )`
`sqlite_fetch_single($r)`	`$r->fetchSingle( )`
`$safe = sqlite_escape_string($s)`	`$safe = $db->escapeString($s)`
`$id = sqlite_last_insert_rowid($r)`	`$id = $db->lastInsertRowid($r)`

Object Iterators

SQLite takes advantage of a new 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 and is the subject of Chapter 6.

Don't confuse this with `sqlite_array_query( )`. SQLite is not prefetching all the rows and storing them as keys inside an array; instead, upon each loop iteration, it returns a new row as if the row already lived in your results array:

```
// one at a time
$r = $db->query('SELECT * FROM users');
foreach ($r as $row) {
    // do something with $row
}
```

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

```
// one at a time
foreach ($db->query('SELECT * FROM users') as $row) {
    // do something with $row
}
```

While this interface hides many of the messy details of database result retrieval, SQLite must still make the requests and transfer the data from the database. Therefore, this syntax works only in `foreach`. You cannot use a `for` loop or pass `$db->query( )` into other array functions, such as `array_map( )`.

When iterating over an SQLite result, it's usually best to use the `unbuffered_query( )` function or `unbufferedQuery( )` method instead of the simple `query( )` method. Since you rarely take advantage of the additional benefits provided by `query( )`, `unbuffered_query( )` gives you an efficiency gain at no cost.

```
// one at a time
$r = $db->unbufferedQuery('SELECT * FROM users');
foreach ($r as $row) {
    // do something with $row
}
```

Indexes, Error Handling, and In-Memory Tables

Now that you know the basics, it's time to cover the features needed to create robust applications using SQLite. Features such as creating primary and other keys, using in-memory tables, and error handling are all necessary to keep your site up and running in a responsive manner.

Indexes

Adding an *index*, also called a *key*, is the easiest way to improve application performance. When SQLite searches a database without keys, it needs to look at every single row in the table to check for matches. However, after you apply an index to the search fields, SQLite can often avoid this time-consuming process. Instead, it consults a specially constructed record that allows SQLite to quickly look up a field's location within the table.

If you know ahead of time that the data in a particular field in your database is going to be unique (i.e., each value will appear in only one record for that field), then you should declare the field `UNIQUE` in your `CREATE TABLE` SQL statement. SQLite automatically indexes `UNIQUE` fields.

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

In a web application, a `username` field is often unique, whereas a `password` field is not. When `username` is `UNIQUE`, SQLite creates a key, since SQLite needs to scan the column to protect against duplicate entries every time you insert a record. This prevents the database from having two users named `rasmus`. Also, you often query against fields that are important enough to be `UNIQUE`:

```
SELECT * FROM users WHERE username LIKE 'rasmus';
```

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

```
CREATE INDEX indexname ON tablename(fieldname);
```

Here, `indexname` is the name of the index. It can be anything, but *`tablename_fieldname`*`_index` is a good way to protect against reusing the same index name. For example:

```
CREATE INDEX users_username_index ON users(username);
```

Creating a plain-vanilla `INDEX` imposes no `UNIQUE`ness constraints on your data. This is helpful because there are non-unique fields, such as locations or dates, where you still need quick search capabilities:

```
SELECT * FROM stores WHERE state LIKE 'New York';

SELECT * FROM purchases WHERE date LIKE '2004-07-22';
```

You can add a `UNIQUE` key to a pre-existing table:

```
CREATE UNIQUE INDEX indexname ON tablename(fieldname);
```

To remove an index, issue the `DROP INDEX` command:

```
DROP INDEX indexname;
```

Indexes make your database files larger. Other than that, there's usually no harm in keeping an index around, even if you're not using it.

Primary Keys

A *primary key* is a special kind of index. When you place primary key status upon a column in your table, the field serves as a unique identifier for a row. Therefore, if you're interested in gathering information about a specific row in the database, the best way to retrieve it is by using its primary key.

A field with primary key status must be an integer. SQLite assigns the number `1` to the first row put into the table, `2` to the second, and so on. If you delete a line from the table, SQLite preserves the hole in the database and places any new records at the end instead of filling up the empty row.

To get SQLite to automatically create this strictly increasing set of values, first define a column in your table as an `INTEGER PRIMARY KEY`. Extending the previous example:

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

Then, when you add a new row to the table, pass `NULL` as the value of the primary key:

```
INSERT INTO users VALUES (NULL, 'rasmus', 'z.8cMpdFbNAPw');
```

If you want to assign a specific number to a row as its primary key, pass that number instead of `NULL`. To find the value of the primary key of the last added row, call `sqlite_last_insert_rowid( )` (or `lastInsertRowid( )` when using the OO interface). For example:

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

The `$rowid` variable holds the primary key assigned to your `INSERT`.

This method is better than writing a query that retrieves the largest valued key; it's possible that between inserting your initial row and making this request, another user has altered the table.

Error Handling

Just like the `mysqli` extension, SQLite error handling differs depending on whether you use the procedural or object-oriented interface. With the procedural interface, you must check the return value of each SQLite call and then consult the message in a special SQLite error variable. Alternatively, the SQLite object tosses an `SQLiteException` whenever it encounters dragons.

Procedural error handling

Here is a good way to structure your procedural code to check for SQLite errors:

```
$db = sqlite_open($database, 0666, $sqlite_error) or die ($sqlite_error);

if ($r = sqlite_query($db, $sql)) {
    // row iteration code here
} else {
    die (sqlite_error_string(sqlite_last_error($db)));
}
```

There are three different ways to access SQLite errors. When you initially try to connect to an SQLite database, the third parameter to `sqlite_open( )` (in this case `$sqlite_error`) is a variable passed by reference. If SQLite cannot open the database, it will return `false` and store the error message in `$sqlite_error`.

The second parameter to `sqlite_open( )` is the *mode*, which is an octal number that describes the file permissions SQLite uses when creating a new database. Currently, the SQLite extension always uses a mode of `0666`, regardless of what's passed in during `sqlite_open( )`. In other words, this value is ignored completely; however, it may be respected in future versions of the extension. This mode means the database is readable and writable by all users, including the web server.

Once your connection is established, SQLite still returns `false` upon errors, but it no longer uses `$sqlite_error`. Instead, it has a pair of error-reporting functions: `sqlite_last_error( )` and `sqlite_error_string( )`.

The first function, `sqlite_last_error( )`, returns the SQLite error code for the most recent error. Since the error code is a number, it's not very helpful for humans. To convert the number to an actual error message, pass it to `sqlite_error_string( )`.

In the previous example, any error triggers a `die( )`. More user-friendly applications require gentler error handling. Using `error_log( )` in combination with a polite, generic message to users may be the best solution.

You cannot "save up" error checking while you complete a series of queries. SQLite resets the value returned by `sqlite_last_error( )` after every query, so old error messages will be removed before you view them. SQLite even resets the message after an error-free query, so a query with an error followed by valid query leaves you with an error message of `not an error`.

Object-oriented error handling

When you use the object-oriented interface to the SQLite extension, you need to process exceptions or risk a fatal error. Exceptions are a method of

error processing that eliminates the need to check return values of functions. They're described in more detail in Chapter 7.

SQLite doesn't always throw exceptions instead of returning `NULL`. In fact, the opposite it true: it throws exceptions only from its constructor. Therefore, while you need to catch that single exception, you still need to rely on traditional error handling for other errors.

Example 4-2 demonstrates this.

Example 4-2. Catching SQLite exceptions

```
$database = 'sqlite.db';
$sql      = 'INVALID SQL';
try {
    $db = new SQLiteDatabase($database);
} catch (SQLiteException $error) {
    print "Message: ".$error->getMessage( )."\n";
    print "File:".$error->getFile( )."\n"; die;
}

if ($r = $db->query($sql)) {
    // row iteration code here
} else {
    die (sqlite_error_string($db->lastError()));
}
```

When SQLite has an error, it throws an `SQLiteException`. After you catch the exception, learn more about the specific error by calling `getMessage( )` and find out which file caused the error with `getFile( )`.

For example, if you try to create an SQLite database file in a location where you do not have write permission, the code inside the `catch` block prints:

```
Message: sqlite_factory( ): unable to open database: /sbin/sqlite.db
File: /www/docroot/sqlite.php
```

When you detect an error outside of the constructor, as in the `query( )` method in Example 4-2, use the `lastError( )` method to retrieve the error code. To convert this number to a human-understandable message, use `sqlite_error_string( )`. The function `sqlite_error_string( )` is not an object method, because it is static and does not vary between database instances.

In-Memory Tables

For extra-fast access, SQLite supports storing tables in RAM instead of on disk. Unfortunately, these tables do not persist across requests, so you cannot create them once and then refer to them again and again; instead, they need to be created each time a page loads. Therefore, these tables are best

used in applications that load in lots of data up front and then make a series of requests, such as a report generation script.

To tell SQLite to use an in-memory database, pass the token :memory: as your database name:

```
sqlite_open(':memory:');
sqlite_query('CREATE TABLE...');
```

Besides the special database name, there's no difference between using in-memory and on-disk tables. You interact with them using the same set of PHP and SQL commands.

Transactions

SQLite supports database transactions. Transactions are good for ensuring database consistency, but they serve a second purpose in SQLite: speed. When a set of queries are grouped together inside a transaction, SQLite executes them significantly faster than if they were performed individually. The more queries you throw at SQLite simultaneously, the larger the percentage increase in speed.

When SQLite creates a connection or makes a query, it does a certain amount of setup; likewise, when it closes a connection or completes a query, it again must perform a sequence of housecleaning tasks. These duties are relatively expensive, but SQLite needs to do this only once per transaction, regardless of how many queries are inside the transaction. This translates into a performance improvement.

However, there's a downside to using transactions in SQLite: when you wrap all your calls into a transaction, SQLite locks the entire database file, and the locked file cannot be accessed by other users. (More finely grained locking capabilities are a benefit of using a "real" database instead of SQLite.) If you're more concerned about overall system responsiveness than with optimizing for a specific action, benchmark your site to evaluate whether using transactions in this manner is appropriate in your script.

To signal to SQLite that you want to begin a transaction, use the keyword BEGIN; to end a transaction, use COMMIT. In PHP, pass these keywords as part of your SQL inside of sqlite_query( ):

```
$users = array(array('rasmus', 'z.8cMpdFbNAPw'),
               array('zeev'  , 'asd34.23NNDeq'));

$sql = 'BEGIN;';
foreach ($users as $user) {
    $sql .= "INSERT INTO users
                 VALUES('${user[0]}', '${user[1]}');";
}
```

```
$sql .= 'COMMIT;';

sqlite_query($db, $sql);
```

The SQL opens with `BEGIN`, and then PHP iterates through an array, appending a series of `INSERT`s to `$sql`. When the loop is done, `COMMIT` is appended. SQL statements are separated by semicolons (;). This lets SQLite know to move from one statement to another. Unlike the MySQL extensions, it is always acceptable to combine multiple SQL statements in a line, even if you're not within a transaction.

You can also spread out a transaction over multiple calls to `sqlite_query( )` like this:

```
$users = array(array('rasmus', 'z.8cMpdFbNAPw'),
               array('zeev'  , 'asd34.23NNDeq'));

sqlite_query($db, 'BEGIN;');

foreach ($users as $user) {
        // Assume data is already escaped
    $sql = "INSERT INTO users
                VALUES('${user[0]}', '${user[1]}');";
    sqlite_query($db, $sql);
}

sqlite_query($db, 'COMMIT;');
```

It is more efficient to make just a single query; however, spreading your queries out gives you the opportunity to undo, or *roll back*, a transaction.

For instance, here's a modification of the previous example that aborts the transaction if an error is found:

```
function add_users($db, $users) {
    $error = false;

    // Start transaction
    sqlite_query($db, 'BEGIN;');

    // Add each new user one-by-one
    foreach ($users as $user) {
        $sql = "INSERT INTO users
                    VALUES('${user[0]}', '${user[1]}');";
        sqlite_query($db, $sql);

        // Abort if there's an error
        if (sqlite_last_error($db)) {
            $error = true;
            break;
        }
    }
```

```
        // Revert previous commits on error; otherwise, save
        if ($error) {
            sqlite_query($db, 'ROLLBACK;');
        } else {
            sqlite_query($db, 'COMMIT;');
        }

        return !$error;
    }
```

This function does the same loop through `$users`, but now it checks `sqlite_last_error( )` after every `INSERT`. If there's an error, the function returns a true value, so `$error` gets set and you break out of the loop. When there are no errors, `sqlite_last_error( )` returns `0`.

Instead of automatically committing the transaction, check `$error`. If an error is found, reverse the transaction by executing the `ROLLBACK` command. Issuing a `ROLLBACK` instructs SQLite to revert the status of the database to its condition before `BEGIN` was sent.

Here is an example that triggers a rollback:

```
    $db = sqlite_open('/www/support/users.db');

    $users = array(array('rasmus', 'z.8cMpdFbNAPw'),
                   array('zeev'  , 'asd34.23NNDeq'),
                   array('rasmus', 'z.8cMpdFbNAPw'));

    add_users($db, $users);
```

Assume the users table requires that each `username` entry be `UNIQUE`. Since there are two entries in the array with a `username` of `rasmus`, SQLite issues an error when you attempt to enter the second `rasmus` into the table.

You could ignore the error and proceed, but as things currently stand, the entire set of users is skipped. A more sophisticated example would examine the specific value returned by `sqlite_last_error( )` and take different actions on a case-by-case basis. This would let you skip over a minor error like this but also let you revert the transaction if a more drastic error occurred.

User-Defined Functions

In addition to all the built-in SQL functions, such as `lower( )` and `upper( )`, you can extend SQLite to include functions of your own written in PHP. These are known as *user-defined functions*, or *UDFs* for short. With a UDF, you embed logic into SQLite and avoid doing it yourself in PHP. This way, you take advantage of all the features inherent in a database, such as sorting and finding distinct entries.

There are two types of UDFs: standard and aggregate. Standard UDFs are one-to-one: when given a single row of data, they return a single result. Functions that change case, calculate cryptographic hashes, and compute the sales tax on an item in a shopping cart are all standard functions. In contrast, aggregate functions are many-to-one: when using an aggregate function, SQLite passes it multiple rows and receives only a single value.

Although it is not a UDF, the most popular aggregate function is `count( )`, which returns the number of rows passed to it. Besides `count( )`, most aggregate functions are related to statistics: finding the average, standard deviation, or the maximum or minimum value of a set of data points.

Standard Functions

UDFs are good for chopping up strings so you can perform nonstandard collations and groupings. For example, you want to sort through a list of URLs, maybe from a referrer log file, and create a list of unique hostnames sorted alphabetically. So, *http://www.example.com/directory/index.html* and *http://www.example.com/page.html* would both map to one entry: *http://www.example.com*.

To do this in PHP, you need to retrieve all the URLs, process them inside your script, and then sort them. Plus, somewhere in all that, you need to do the deduping. However, if it weren't for that pesky URL-conversion process, this could all be done in SQL using the `DISTINCT` and `ORDER BY` keywords.

With a UDF like the one shown in Example 4-3, you foist all that hard work back onto SQLite where it belongs.

Example 4-3. Retrieving unique hostnames using an SQLite UDF

```
// CREATE table and INSERT URLs
$db = sqlite_open('/www/support/log.db');
$sql = 'CREATE TABLE access_log(url);';

$urls = array('http://www.example.com/directory/index.html',
              'http://www.example.com/page.html');

foreach ($urls as $url) {
    $sql .= "INSERT INTO access_log VALUES('$url');";
}
sqlite_query($db, $sql);


// UDF written in PHP
function url2host($url) {
    $parts = parse_url($url);
    return "$parts[scheme]://$parts[host]";
}
```

Example 4-3. Retrieving unique hostnames using an SQLite UDF (continued)

```
// Tell SQLite to associate PHP function url2host( ) with the
// SQL function host( ). Say that host( ) will only take 1 argument.
sqlite_create_function($db, 'host', 'url2host', 1);

// Do the query
$r = sqlite_query($db, 'SELECT DISTINCT host(lower(url)) AS clean_host
                        FROM access_log ORDER BY clean_host;');

// Loop through results
while ($row = sqlite_fetch_array($r)) {
    print "$row[clean_host]\n";
}
```

```
http://www.example.com
```

To use a UDF, you first write a regular function in PHP. The function's arguments are what you want to pass in during the `SELECT`, and the function should `return` a single value. The `url2host( )` function takes a URL; calls the built-in PHP function `parse_url( )` to break the URL into its component parts; and returns a string containing the scheme, `://`, and the host. So, `http://www.example.com/directory/index.html` gets broken apart into many pieces. `http` is stored into `$parts['scheme']` and `www.example.com` goes into `$parts['host']`.* This creates a return value of `http://www.example.com`.

The next step is to register `url2host( )` with SQLite using `sqlite_create_function( )`. This function takes four parameters: the database handle, the name you want the function to be called inside SQLite, the name of your function written in PHP, and the number of arguments your function expects. The last parameter is optional, but if you know for certain that your function accepts only a specific number of parameters, providing this information helps SQLite optimize things behind the scenes. In this example, the SQL function is `host( )`, while the PHP function is `url2host( )`. These names can be the same; they're different here to make the distinction between them clear.

Now you can use `host( )` inside any SQL calls using that database connection. The SQL in Example 4-3 `SELECT`s `host(lower(url)) AS clean_host`. This takes the URL stored in the `url` column, converts it to lowercase, and calls the UDF `host( )`.

The function is not permanently registered with the database, and goes away when you close the database. If you want to use it when you reopen the database, you must reregister it. Also, the function is registered only for that database; if you open up a new database using `sqlite_connect( )`, you need to call `sqlite_create_function( )` again.

* The other portions of the URL are stored in different variables.

The returned string is then named `AS clean_host`; this lets you refer to the results later on in the SQL query and also access the value in PHP using that name. Since you're still in SQLite, you can take advantage of this to sort the list using `ORDER BY host`. This sorts the results in alphabetical order, starting at `a`.

Now that's cool, but it's not *that* cool. What *is* cool is SQLite's ability to call UDFs in the `ORDER BY` clause. If you use the default alphabetical sort, *http://php.example.org* and *http://www.example.org* won't be near each other, because "p" and "w" aren't next to each other in the alphabet. Yet both hosts are located under the `example.org` domain, so it makes sense that they should be listed together. Not surprisingly, another UDF saves the day.

```
function reverse_host($url) {
    list ($scheme, $host) = explode('://', $url);
    return join('.',array_reverse(explode('.',$host)));
}

sqlite_create_function($db, 'reverse', 'reverse_host', 1);
```

The `reverse_host( )` function takes a URL and chops it into two bits, the scheme and host, by `explode( )`ing on `://`. You can do this because the previous UDF, `host( )`, has specifically created strings in this manner. Next, `$host` is passed through a series of three functions that splits it up into its component parts, reverses those parts, and then glues them back together. This flips around the pieces of the host separated by periods, but doesn't actually reverse the text. So, `www.example.org` becomes `org.example.www` and not `gro.elpmaxe.www` or `www.elpmaxe.gro`.

This reversed hostname is perfect for sorting. When you alphabetize `org.example.www`, it nicely sits next to all its brethren in the `.org` top-level domain, then sorts by the other hosts inside `example.org`, and finally orders the remaining subdomains. And that's exactly what you want.

You then register `reverse_host( )` in the exact same way you registered `url2string( )`, using `sqlite_create_function( )`.

Once that's done, you can call `reverse( )` inside your SQL query:

```
$r = sqlite_query($db, 'SELECT DISTINCT host(lower(url)) AS clean_host
                        FROM access_log ORDER BY reverse(clean_host);');
```

Given the following list of URLs as input:

```
http://www.example.com
http://php.example.org
http://www.example.org
```

you get the following as output:

```
http://www.example.com
http://php.example.org
http://www.example.org
```

The URL containing php.example.com has filtered down in the list below www.example.com, even though php comes before www in the alphabet.

In contrast, Example 4-4 shows what you need to do to implement this in PHP without UDFs.

Example 4-4. Sorting unique hostnames without using SQLite UDFs

```
function url2host($url) {
    $parts = parse_url($url);
    return "$parts[scheme]://$parts[host]";
}

function reverse_host($url) {
    list ($scheme, $host) = explode('://', $url);
    return join('.',array_reverse(explode('.',$host)));
}

function host_sort($a, $b) {
    $count_a = $GLOBALS['hosts'][$a];
    $count_b = $GLOBALS['hosts'][$b];

    if ($count_a < $count_b) { return  1; }
    if ($count_a > $count_b) { return -1; }

    return strcasecmp(reverse_host($a), reverse_host($b));
}

$hosts = array( );

$r = sqlite_unbuffered_query($db, 'SELECT url FROM access_log');
while ($url = sqlite_fetch_single($r)) {
    $host = url2host($url);
    $hosts[$host]++ ;
}

uksort($hosts, 'host_sort');
```

This process breaks down into many steps:

1. Make a database query for urls.
2. Retrieve url into $url using sqlite_fetch_single( ).
3. Convert $url into a host and store it in $host.
4. Place $url as a new element in the $hosts array and increment that element by 1. This tracks the number of times each URL has appeared.
5. Perform a user-defined key sort on the $hosts array.

The sqlite_fetch_single( ) function returns the first (and in this case only) column from the result as a string. This allows you to skip the step of saving

the result as an array and then extracting the element, either by using `list` or as the `0`th index.

Doing `$hosts[$host]++` is a old trick that allows you to easily count the number of times each key appears in a list.

Since `uksort( )` only passes array keys to the sorting function, `host_host( )` is not very elegant, because it requires using a global variable to determine the number of hits for each element.

Overall, compared to a UDF, this method requires more memory, execution time, and lines of code, because you're replicating database functionality inside PHP.

User-Defined Aggregate Functions

As discussed earlier, most aggregate functions are statistical functions, such as `AVG( )` or `STDEV( )`. People usually use aggregate functions to return a single row from their query, but that's not a requirement. You can use them to link together a set of related rows, to compact your query and return one row per group.

This extension to the earlier referrer log sorting example shows how to use an aggregate function to provide the total number of hits per hostname, in addition to everything in the previous section:

```
SELECT DISTINCT host(lower(url)) AS clean_host, COUNT(*) AS hits
FROM access_log GROUP BY clean_host ORDER BY hits DESC, reverse(clean_host)
```

The `COUNT(*)` function sums the total number of rows per host. However, this won't work without adding the `GROUP BY host` clause. `GROUP`ing rows allows `COUNT(*)` to know which sets of entries belong together. Whenever you have an aggregate function—such as `COUNT( )`, `SUM( )`, or any function that takes a set of rows as input and returns only a single value as its output—use `GROUP BY` when you want your query to return multiple rows. (If you're just doing a basic `SELECT COUNT(*) FROM host` to find the total number of rows in the table, there's no need for any `GROUP`ing.)

`COUNT(*)` is aliased to `hits`, which allows you to refer to it in the `ORDER BY` clause. Then, to sort the results first by total hits, from most to least, and then alphabetically within each total, use `ORDER BY hits DESC, reverse(host)`. By putting `hits` first, you prioritize it over `reverse(clean_host)` and the `DESC` keyword flips the sorting order to descending (the default is ascending).

Using that query, this set of sites:

```
http://www.example.org
http://www.example.org
```

```
http://www.example.com
http://php.example.org
```

and this PHP code:

```
while ($row = sqlite_fetch_array($r)) {
    print "$row[hits]: $row[clean_host]\n";
}
```

gives:

```
2: http://www.example.org
1: http://www.example.com
1: http://php.example.org
```

Furthermore, to restrict results to sites with more hits than a specified amount, use a `HAVING` clause:

```
SELECT DISTINCT host(lower(url)) AS clean_host, COUNT(*) AS hits
       FROM access_log
       GROUP BY clean_host
       HAVING hits > 1
       ORDER BY hits DESC, reverse(clean_host)
```

You cannot use `WHERE` here, because `WHERE` can only operate on data directly from a table. Here the restriction `hits > 1` compares against the result of a `GROUP BY`, so you need to employ `HAVING` instead.

You can define your own aggregate functions for SQLite in PHP. Unlike standard UDFs, you actually need to define two functions: one that's called for each row and one that's called after all the rows have been passed in.

The code in Example 4-5 shows how to create a basic SQLite user-defined aggregate function that calculates the average of a set of numbers.

Example 4-5. Averaging numbers using an SQLite aggregate function

```
// CREATE table and INSERT numbers
$db = sqlite_open('/www/support/data.db');
$sql = 'CREATE TABLE numbers(number);';
$numbers = array(1, 2, 3, 4, 5);
foreach ($numbers as $n) {
    $sql .= "INSERT INTO numbers VALUES($n);";
}
sqlite_query($db, $sql);

// average_step( ) is called on each row.
function average_step(&$existing_data, $new_data) {
    $existing_data['total'] += $new_data;
    $existing_data['count']++;
}

// average_final( ) computes the average and returns it.
function average_final(&$existing_data) {
```

Example 4-5. Averaging numbers using an SQLite aggregate function (continued)

```
    return $existing_data['total'] / $existing_data['count'];
}

sqlite_create_aggregate($db, 'average', 'average_step', 'average_final');

$r = sqlite_query($db, 'SELECT average(number) FROM numbers');
$average = sqlite_fetch_single($r);
print $average;
```

3

First, you define the two aggregate functions in PHP, just as you do for regular UDFs. However, the first parameter for both functions is a variable passed by reference that is used to keep track of the UDF's state. In this example, you need to track both the running sum of the numbers and how many rows have contributed to this total. That's done in `average_step( )`.

In `average_final( )`, the final sum is divided by the number of elements to find the average. This is the value that's returned by the function and passed back to SQLite (and, eventually, to you).

To formally create an aggregate UDF, use `sqlite_create_aggregate( )`. It works like `sqlite_create_function( )`, but you pass both PHP function names instead of just one.

Binary Data

SQLite is not binary safe by default. Requiring PHP to automatically protect against problems caused by binary data causes a significant reduction in speed, so you must manually encode and decode data when it might be anything other than plain text. If your UDFs only operate on text, this isn't a problem.

Inside a UDF, use `sqlite_udf_binary_decode( )` to convert data stored in SQLite into usable strings in PHP:

```
function udf_function_encode($encoded_data) {
    $data = sqlite_udf_binary_decode($encoded_data);
    // rest of the function...
}
```

When you're finished, if the return value might also be binary unsafe, re-encode it using `sqlite_udf_binary_encode( )`:

```
function udf_function_decode($encoded_data) {
    // rest of the function...
    return sqlite_udf_binary_encode($return_value);
}
```