CHAPTER 22

SQLite

As of PHP 5.0, support for the open source database server SQLite (`http://www.sqlite.org/`) is enabled by default. This was done in response to both the decision to unbundle MySQL from version 5 due to licensing discrepancies, and a realization that users might benefit from the availability of another powerful database server that nonetheless requires measurably less configuration and maintenance as compared to similar products. This chapter introduces both SQLite and PHP's ability to interface with this surprisingly capable database server.

Introduction to SQLite

SQLite is a very compact, multiplatform SQL database engine written in C. Practically SQL-92–compliant, SQLite offers many of the core database management features made available by competing products such as MySQL, Oracle, and PostgreSQL, yet at considerable savings in terms of cost, learning curve, and administration investment. Some of SQLite's more compelling characteristics include:

- SQLite stores an entire database in a single file, allowing for easy backup and transfer.
- SQLite's entire database security strategy is based entirely on the executing user's file permissions. So, for example, user `web` might own the Web server daemon process and, through a script executed on that server, attempt to open and write to an SQLite database named `mydatabase.db`. Whether this user is capable of doing so depends entirely on the `mydatabase.db` permissions.
- SQLite offers default transactional support, automatically integrating commit and rollback support.
- SQLite is available under a public domain license (it's free) for both the Microsoft Windows and Unix platforms.

This section offers a brief guide to the SQLite command-line interface. The purpose of this section is twofold. First, it provides you with at least an introductory look at this useful client. Second, the steps demonstrated create the data that will serve as the basis for all subsequent examples in this chapter.

Installing SQLite

As mentioned, SQLite comes bundled with PHP as of version 5.0, including both the database engine and the interface. This means you can take advantage of SQLite without having to install any other software. However, there is one related utility omitted from the PHP distribution, namely `sqlite`, a command-line interface to the engine. Because this utility is quite useful, consider installing the SQLite library from `http://www.sqlite.org/`, which includes a copy of the utility. Then configure (or reconfigure) PHP with the `--with-sqlite=/path/to/library` flag. The next section shows you how to use this interface.

Windows users need to download the SQLite extension from the following location:

```
http://snaps.php.net/win32/PECL_STABLE/php_sqlite.dll
```

Once downloaded, place this DLL file within the same directory as the others (`PHP-INSTALL-DIR\ext`) and add the following line to your `php.ini` file:

```
php_extension=php_sqlite.dll
```

■ **Note** Shortly before press time, PHP 5.1 was released, and with it came a significant change in which SQLite is supported in this and newer versions. According to the developers, users interested in taking advantage of SQLite should consider using PDO in conjunction with the SQLite version 3 driver. See Chapter 23 for more information about PDO.

Using the SQLite Command-Line Interface

The SQLite command-line interface offers a simple means for interacting with the SQLite database server. With this tool, you can create and maintain databases, execute administrative processes such as backups and scripts, and tweak the client's behavior. Begin by opening a terminal window and executing SQLite with the `help` option:

```
%>sqlite -help
```

If you've downloaded SQLite version 3 for Windows, then you need to execute it like so:

```
%>sqlite3 -help
```

In either case, before exiting back to the command line, you'll be greeted with the command's usage syntax and a menu consisting of numerous options. Note that the usage syntax specifies that a filename is required to enter the SQLite interface. This filename is actually the name of the database. When supplied, a connection to this database will be opened, if the executing user possesses adequate permissions. If the supplied database does not exist, it will be created, again if the executing user possesses the necessary privileges.

As an example, create a test database named `mydatabase.db`. This database consists of a single table, `employee`. In this section, you'll learn how to use SQLite's command-line program to create the database, table, and sample data. Although this section isn't intended as a replacement for the documentation, it should be sufficient to enable you to familiarize yourself with the very basic aspects of SQLite and its command-line interface.

1. Open a new SQLite database, as follows. Because this database presumably doesn't already exist, the mere act of opening a nonexistent database will first result in its creation.

   ```
   %>sqlite mydatabase.db
   ```

2. Create a table:

   ```
   sqlite>create table employee (
   ...>empid integer primary key,
   ...>name varchar(25),
   ...>title varchar(25));
   ```

3. Check the table structure for accuracy:

   ```
   sqlite>.schema employee
   ```

Note that a period (`.`) prefaces the `schema` command. This syntax requirement holds true for all commands found under the help menu.

4. Insert a few data rows:

   ```
   sqlite> insert into employee values(NULL,"Jason Gilmore","Chief Slacker");
   sqlite> insert into employee values(NULL,"Sam Spade","Technologist");
   sqlite> insert into employee values(NULL,"Ray Fox","Comedian");
   ```

5. Query the table, just to ensure that all is correct:

   ```
   sqlite>select * from employee;
   ```

   You should see:

```
1|Jason Gilmore|Chief Slacker
2|Sam Spade|Technologist
3|Ray Fox|Comedian
```

6. Quit the interface with the following command:

   ```
   sqlite>.quit
   ```

PHP's SQLite Library

The SQLite functions introduced in this section are quite similar to those found in the other PHP-supported database libraries such as MySQL and PostgreSQL. In fact, for many of the functions the name is the only real differentiating factor. If you have a background in PostgreSQL, picking up SQLite should be a snap. Even if you're entirely new to the concept, don't worry; you'll likely find that these functions are extremely easy to use.

SQLite Directives

One PHP configuration directive is pertinent to SQLite. It's introduced in this section.

sqlite.assoc_case (0,1,2)

Scope: PHP_INI_ALL; Default value: 0

While SQLite uses (and retrieves) column names in exactly the same format in which they appear in the database schema, various other database servers attempt to standardize name formats by always returning them in uppercase letters. This dichotomy can be problematic when porting an application to SQLite, because the column names used in the application may be standardized in uppercase to account for the database server's tendencies. To modify this behavior, you can use the sqlite.assoc_case directive. It determines the case used for retrieved column names. By default, this directive is set to 0, which retains the case used in the table definitions. If it's set to 1, the names will be converted to uppercase. If it's set to 2, the names will be converted to lowercase.

Opening a Connection

Before you can retrieve or manipulate any data located in an SQLite database, you must first establish a connection. Two functions are available for doing so, sqlite_open() and sqlite_popen().

sqlite_open()

```
resource sqlite_open (string filename [,int mode [,string &error_message]])
```

The sqlite_open() function opens an SQLite database, first creating the database if it doesn't already exist. The filename parameter specifies the database name. The optional mode parameter determines the access privilege level under which the database will be opened, and is specified as an octal value (the default is 0666) as might be used to specify modes in Unix. Currently, this parameter is unsupported by the API. The optional error_message parameter is actually automatically assigned a value specifying an error if the database could not be opened. If the database is successfully opened, the function returns a resource handle pointing to that database.

Consider an example:

```
<?php
   $sqldb = sqlite_open("/home/book/20/mydatabase.db")
                        or die("Could not connect!");
?>
```

This either opens an existing database named mydatabase.db, creates a database named mydatabase.db within the directory /home/book/20/, or results in an error, likely because of privilege problems. If you experience problems creating or opening the database, be sure that the user owning the Web server process possesses adequate permissions for writing to this directory.

sqlite_popen()

```
resource sqlite_popen (string filename [,int mode [,string &error_message]])
```

The function sqlite_popen() operates identically to sqlite_open() except that it uses PHP's persistent connection feature in an effort to conserve resources. The function first verifies whether a connection already exists; if it does, it reuses this connection; otherwise, it creates a new one. Because of the performance improvements offered by this function, you should use sqlite_popen() instead of sqlite_open().

OBJECT-ORIENTED SQLITE

Although this chapter introduces PHP's SQLite library using the procedural approach, an object-oriented interface is also supported. All functions introduced in this chapter are also supported as methods when using the object-oriented interface (however, the names differ slightly in that the sqlite_ prefix is removed from them); therefore, the only significant usage deviation is in regard to referencing the methods by way of an object ($objectname->methodname()) rather than by passing around a resource handle. Also, the constructor takes the place of the sqlite_open() function, negating the need to specifically open a database connection. The class is instantiated by calling the constructor like so:

```
$sqldb = new SQLiteDatabase(string databasename [, int mode
                            [, string &error_message]]);
```

Once the object is created, you can call methods just as you do for any other class. For example, you can execute a query and determine the number of rows returned with the following code:

```
$sqldb = new SQLiteDatabase("mydatabase.db");
$sqldb->query("SELECT * FROM employee");
echo $sqldb->numRows()." rows returned.";
```

See the PHP manual (http://www.php.net/sqlite) for a complete listing of the available methods.

Creating a Table in Memory

Sometimes your application may require database access performance surpassing even that offered by SQLite's default behavior, which is to manage databases in self-contained files. To satisfy such requirements, SQLite supports the creation of in-memory (RAM-based) databases, accomplished by calling sqlite_open() like so:

```
$sqldb = sqlite_open(":memory:");
```

Once open, you can create a table that will reside in memory by calling sqlite_query(), passing in a CREATE TABLE statement. Keep in mind that such tables are volatile, disappearing once the script has finished executing!

Closing a Connection

Good programming practice dictates that you close pointers to resources once you're finished with them. This maxim holds true for SQLite; once you've completed working with a database, you should close the open handle. One function, sqlite_close(), accomplishes just this.

sqlite_close()

```
void sqlite_close (resource dbh)
```

The function `sqlite_close()` closes the connection to the database resource specified by `dbh`. You should call it after all necessary tasks involving the database have been completed. An example follows:

```
<?php
    $sqldb = sqlite_open("mydatabase.db");
    // Perform necessary tasks
    sqlite_close($sqldb);
?>
```

Note that if a pending transaction has not been completed at the time of closure, the transaction will automatically be rolled back.

Querying a Database

The majority of your time spent interacting with a database server takes the form of SQL queries. The functions `sqlite_query()` and `sqlite_unbuffered_query()` offer the main vehicles for submitting these queries to SQLite and returning the subsequent result sets. You should pay particular attention to the specific advantages of each, however, because applying them inappropriately can negatively impact performance and capabilities.

sqlite_query()

```
resource sqlite_query (resource dbh, string query)
```

The `sqlite_query()` function executes a SQL query, `query`, against the database specified by `dbh`. If the query is intended to return a result set, `FALSE` is returned if the query fails. All other queries return `TRUE` if the query was successful, and `FALSE` otherwise.

In order to provide a practical example, other functions are used in this example that have not yet been introduced. Not to worry; just understand that the `sqlite_query()` function is responsible for sending and executing a SQL query. Soon enough, you'll learn the specifics regarding the other functions used in the example.

```
<?php
    $sqldb = sqlite_open("mydatabase.db");
    $results = sqlite_query($sqldb, "SELECT * FROM employee");
    while (list($empid, $name) = sqlite_fetch_array($results)) {
        echo "Name: $name (Employee ID: $empid) <br />";
    }
    sqlite_close($sqldb);
?>
```

This yields the following results:

```
Name: Jason Gilmore (Employee ID: 1)
Name: Sam Spade (Employee ID: 2)
Name: Ray Fox (Employee ID: 3)
```

Keep in mind that `sqlite_query()` will only execute the query and return a result set (if one is warranted); it will not output or offer any additional information regarding the returned data. To obtain such information, you need to pass the result set into one or several other functions, all of which are introduced in the following sections. Furthermore, `sqlite_query()` is not limited to executing `SELECT` queries. You can use this function to execute any supported SQL-92 query.

sqlite_unbuffered_query()

```
resource sqlite_unbuffered_query (resource dbh, string query)
```

The `sqlite_unbuffered_query()` function can be thought of as an optimized version of `sqlite_query()`, identical in every way except that it returns the result set in a format intended to be used in the order in which it is returned, without any need to search or navigate it in any other way. This function is particularly useful if you're solely interested in dumping a result set to output, an HTML table or a text file, for example.

Because this function is optimized for returning result sets intended to be output in a straightforward fashion, you cannot pass its output to functions like `sqlite_num_rows()`, `sqlite_seek()`, or any other function with the purpose of examining or modifying the output or output pointers. If you require the use of such functions, use `sqlite_query()` to retrieve the result set instead.

sqlite_last_insert_rowid()

```
int sqlite_last_insert_rowid (resource dbh)
```

It's common to reference a newly inserted row immediately after the insertion is completed, which in many cases is accomplished by referencing the row's auto-increment field. Because this value will contain the highest integer value for the field, determining it is as simple as searching for the column's maximum value. The function `sqlite_last_insert_rowid()` accomplishes this for you, returning that value.

Parsing Result Sets

Once a result set has been returned, you'll likely want to do something with the data. The functions in this section demonstrate the many ways that you can parse the result set.

sqlite_fetch_array()

```
array sqlite_fetch_array (resource result [, int result_type [, bool decode_binary]])
```

The `sqlite_fetch_array()` function returns an associative array consisting of the items found in the `result` set's next available row, or returns `FALSE` if no more rows are available. The optional `result_type` parameter can be used to specify whether the columns found in the `result` set row

should be referenced by their integer-based position in the row or by their actual name. Specifying SQLITE_NUM enables the former, while SQLITE_ASSOC enables the latter. You can return both referential indexes by specifying SQLITE_BOTH. Finally, the optional decode_binary parameter determines whether PHP will decode the binary-encoded target data that had been previously encoded using the function sqlite_escape_string(). This function is introduced in the later section, "Working with Binary Data."

■ **Tip** If SQLITE_ASSOC or SQLITE_BOTH are used, PHP will look to the sqlite.assoc_case configuration directive to determine the case of the characters.

Consider an example:

```
<?php
   $sqldb = sqlite_open("mydatabase.db");
   $results = sqlite_query($sqldb, "SELECT * FROM employee");
   while ($row = sqlite_fetch_array($results,SQLITE_BOTH)) {
          echo "Name: $row[1] (Employee ID: ".$row['empid'].")<br />";
   }
   sqlite_close($sqldb);
?>
```

This returns:

```
Name: Jason Gilmore (Employee ID: 1)
Name: Sam Spade (Employee ID: 2)
Name: Ray Fox (Employee ID: 3)
```

Note that the SQLITE_BOTH option was used so that the returned columns could be referenced both by their numerically indexed position and by their name. Although it's not entirely practical, this example serves as an ideal means for demonstrating the function's flexibility.

One great way to render your code a tad more readable is to use PHP's list() function in conjunction with sql_fetch_array(). With it, you can both return and parse the array into the required components all on the same line. Let's revise the previous example to take this idea into account:

```
<?php
   $sqldb = sqlite_open("mydatabase.db");
   $results = sqlite_query($sqldb, "SELECT * FROM employee");
   while (list($empid, $name) = sqlite_fetch_array($results)) {
          echo "Name: $name (Employee ID: $empid)<br />";
   }
   sqlite_close($sqldb);
?>
```

sqlite_array_query()

```
array sqlite_array_query ( resource dbh, string query [, int res_type
                           [, bool decode_binary]])
```

The sqlite_array_query() function consolidates the capabilities of sqlite_query() and sqlite_fetch_array() into a single function call, both executing the query and returning the result set as an array. The input parameters work exactly like those introduced in the component functions sqlite_query() and sqlite_fetch_array(). According to the PHP manual, this function should only be used for retrieving result sets of fewer than 45 rows. However, in instances where 45 or fewer rows are involved, this function provides both a considerable improvement in performance and, in certain cases, a slight reduction in total lines of code. Consider an example:

```
<?php
   $sqldb = sqlite_open("mydatabase.db");
   $rows = sqlite_array_query($sqldb, "SELECT empid, name FROM employee");
   foreach ($rows AS $row) {
      echo $row["name"]." (Employee ID: ".$row["empid"].")<br />";
   }
   sqlite_close($sqldb);
?>
```

This returns:

```
Jason Gilmore (Employee ID: 1)
Sam Spade (Employee ID: 2)
Ray Fox (Employee ID: 3)
```

sqlite_column()

```
mixed sqlite_column (resource result, mixed index_or_name [, bool decode_binary])
```

The sqlite_column() function is useful if you're interested in just a single column from a given result row or set. You can retrieve the column either by name or by index offset. Finally, the optional decode_binary parameter determines whether PHP will decode the binary-encoded target data that had been previously encoded using the function sqlite_escape_string(). This function is introduced in the later section, "Working with Binary Data."

For example, suppose you retrieved all rows from the employee table. Using this function, you could selectively poll columns, like so:

```
<?php
    $sqldb = sqlite_open("mydatabase.db");
    $results = sqlite_query($sqldb,"SELECT * FROM employee WHERE empid = '1'");
    $name = sqlite_column($results,"name");
    $empid = sqlite_column($results,"empid");
    echo "Name: $name (Employee ID: $empid) <br />";
    sqlite_close($sqldb);
?>
```

This returns:

```
Name: Jason Gilmore (Employee ID: 1)
```

Ideally, you'll want to use this function when you're working either with result sets consisting of numerous columns or with particularly large columns.

sqlite_fetch_single()

```
string sqlite_fetch_single (resource row_set [, int result_type
                            [, bool decode_binary]])
```

The `sqlite_fetch_single()` function operates identically to `sql_fetch_array()` except that it returns just the value located in the first column of the `row_set`.

■ **Tip** This function has an alias: `sqlite_fetch_string()`. Except for the name, it's identical in every way.

Consider an example. Suppose you're interested in querying the database for a single column. To reduce otherwise unnecessary overhead, you should opt to use `sqlite_fetch_single()` over `sqlite_fetch_array()`, like so:

```
<?php
    $sqldb = sqlite_open("mydatabase.db");
    $results = sqlite_query($sqldb,"SELECT name FROM employee WHERE empid < 3");
    while ($name = sqlite_fetch_single($results)) {
       echo "Employee: $name <br />";
    }
    sqlite_close($sqldb);
?>
```

This returns:

```
Employee: Jason Gilmore
Employee: Sam Spade
```

Retrieving Result Set Details

You'll often want to learn more about a result set than just its contents. Several SQLite-specific functions are available for determining information such as the returned field names, the number of fields and rows returned, and the number of rows changed by the most recent statement. These functions are introduced in this section.

sqlite_field_name()

```
string sqlite_field_name (resource result, int field_index)
```

The `sqlite_field_name()` function returns the name of the field located at the index offset `field_index` found in the `result` set. For example:

```
<?php
   $sqldb = sqlite_open("mydatabase.db");
   $results = sqlite_query($sqldb,"SELECT * FROM employee");
   echo "Field name found at offset #0: ".sqlite_field_name($results,0)."<br />";
   echo "Field name found at offset #1: ".sqlite_field_name($results,1)."<br />";
   echo "Field name found at offset #2: ".sqlite_field_name($results,2)."<br />";
   sqlite_close($sqldb);
?>
```

This returns:

```
Field name found at offset #0: empid
Field name found at offset #1: name
Field name found at offset #2: title
```

As is the case with all numerically indexed arrays, the offset starts at 0, not 1.

sqlite_num_fields()

```
int sqlite_num_fields (resource result_set)
```

The `sqlite_num_fields()` function returns the number of columns located in the `result_set`. For example:

```
<?php
   $sqldb = sqlite_open("mydatabase.db");
   $results = sqlite_query($sqldb, "SELECT * FROM employee");
   echo "Total fields returned: ".sqlite_num_fields($results)."<br />";
   sqlite_close($sqldb);
?>
```

This returns:

```
Total fields returned: 3
```

sqlite_num_rows()

```
int sqlite_num_rows (resource result_set)
```

The sqlite_num_rows() function returns the number of rows located in the result_set. An example follows:

```
<?php
   $sqldb = sqlite_open("mydatabase.db");
   $results = sqlite_query($sqldb, "SELECT * FROM employee");
   echo "Total rows returned: ".sqlite_num_rows($results)."<br />";
   sqlite_close($sqldb);
?>
```

This returns:

```
Total rows returned: 3
```

sqlite_changes()

```
int sqlite_changes (resource dbh)
```

The sqlite_changes() function returns the total number of rows affected by the most recent modification query. For instance, if an UPDATE query modified a field located in 12 rows, then executing this function following that query would return 12.

Manipulating the Result Set Pointer

Although SQLite is indeed a database server, in many ways it behaves much like what you experience when working with file I/O. One such way involves the ability to move the row "pointer" around the result set. Several functions are offered for doing just this, all of which are introduced in this section.

sqlite_current()

```
array sqlite_current (resource result [, int result_type [, bool decode_binary]])
```

The sqlite_current() function is identical to sqlite_fetch_array() in every way except that it does not advance the pointer to the next row of the result set. Instead, it only returns the row residing at the current pointer position. If the pointer already resides at the end of the result set, FALSE is returned.

sqlite_has_more()

```
boolean sqlite_has_more (resource result_set)
```

The sqlite_has_more() function determines whether the end of the result_set has been reached, returning TRUE if additional rows are still available, and FALSE otherwise. An example follows:

```
<?php
    $sqldb = sqlite_open("mydatabase.db");
    $results = sqlite_query($sqldb, "SELECT * FROM employee");
    while ($row = sqlite_fetch_array($results,SQLITE_BOTH)) {
        echo "Name: $row[1] (Employee ID: ".$row['empid'].")<br />";
        if (sqlite_has_more($results)) echo "Still more rows to go!<br />";
           else echo "No more rows!<br />";
    }
    sqlite_close($sqldb);
?>
```

This returns:

```
Name: Jason Gilmore (Employee ID: 1)
Still more rows to go!
Name: Sam Spade (Employee ID: 2)
Still more rows to go!
Name: Ray Fox (Employee ID: 3)
No more rows!
```

sqlite_next()

```
boolean sqlite_next (resource result)
```

The `sqlite_next()` function moves the `result` set pointer to the next position, returning `TRUE` on success and `FALSE` if the pointer already resides at the end of the `result` set.

sqlite_rewind()

```
boolean sqlite_rewind (resource result)
```

The `sqlite_rewind()` function moves the `result` set pointer back to the first row, returning `FALSE` if no rows exist in the result set and `TRUE` otherwise.

sqlite_seek()

```
boolean sqlite_seek (resource result, int row_number)
```

The `sqlite_seek()` function moves the pointer to the row specified by `row_number`, returning `TRUE` if the row exists and `FALSE` otherwise. Consider an example in which an employee of the month will be randomly selected from a result set consisting of the entire staff:

```
<?php
   $sqldb = sqlite_open("mydatabase.db");
   $results = sqlite_query($sqldb, "SELECT empid, name FROM employee");

   // Choose a random number found within the range of total returned rows
   $random = rand(0,sqlite_num_rows($results)-1);

   // Move the pointer to the row specified by the random number
   sqlite_seek($results, $random);

   // Retrieve the employee ID and name found at this row
   list($empid, $name) = sqlite_current($results);
   echo "Randomly chosen employee of the month: $name (Employee ID: $empid)";
   sqlite_close($sqldb);
?>
```

This returns the following (this shows only one of three possible outcomes):

```
Randomly chosen employee of the month: Ray Fox (Employee ID: 3)
```

One point of common confusion that arises in this example regards the starting index offset of result sets. The offset always begins with 0, not 1, which is why you need to subtract 1 from the total rows returned in this example. As a result, the randomly generated row offset integer must fall within a range of 0 and one less than the total number of returned rows.

Learning More About Table Schemas

There is one function available for learning more about an SQLite table schema. It's introduced in this section.

sqlite_fetch_column_types()

```
array sqlite_fetch_column_types (string table, resource dbh)
```

The function `sqlite_fetch_column_types()` returns an array consisting of the column types located in the table identified by `table`. The returned array includes both the associative and numerical hash indices. The following example outputs an array of column types located in the `employee` table used earlier in this chapter:

```
<?php
   $sqldb = sqlite_open("mydatabase.db");
   $columnTypes = sqlite_fetch_column_types("employee", $sqldb);
   print_r($columnTypes);
   sqlite_close($sqldb);
?>
```

This example returns:

```
Array (
   [empid] => integer
   [name] => varchar(25)
   [title] => varchar(25)
)
```

Working with Binary Data

SQLite is capable of storing binary information in a table, such as a GIF or JPEG image, a PDF document, or a Microsoft Word document. However, unless you treat this data carefully, errors in both storage and communication could arise. Several functions are available for carrying out the tasks necessary for managing this data, one of which is introduced in this section. The other two relevant functions are introduced in the next section.

sqlite_escape_string()

```
string sqlite_escape_string (string item)
```

Some characters or character sequences have special meaning to a database, and therefore they must be treated with special care when trying to insert them into a table. For example, SQLite expects that single quotes signal the delimitation of a string. However, because this character is often used within data that you might want to include in a table column, a means is required for tricking the database server into ignoring single quotes on these occasions. This is commonly referred to as "escaping" these special characters, often done by prefacing the special character with some other character, a single quote (') for example. Although you can do this manually, a function is available that will do the job for you. The `sqlite_escape_string()` function escapes any single quotes and other binary-unsafe characters intended for insertion in an SQLite table found in `item`.

Let's use this function to escape an otherwise invalid query string:

```
<?php
   $str = "As they always say, this is 'an' example.";
   echo sqlite_escape_string($str);
?>
```

This returns:

```
As they always say, this is ''an'' example.
```

If the string contains a `NULL` character or begins with `0x01`, circumstances that have special meaning when working with binary data, `sqlite_escape_string()` will take the steps necessary to properly encode the information so that it can be safely stored and later retrieved.

■ **Note** The NULL character typically signals the end of a binary string, while 0x01 is the escape character used within binary data. Therefore, to ensure that the escape character was properly interpreted by the binary data parser, it would need to be decoded.

When you're using user-defined functions, a topic discussed in the next section, you should never use this function. Instead, use the sqlite_udf_encode_binary() and sqlite_udf_decode_binary() functions. Both are introduced in the next section.

Creating and Overriding SQLite Functions

An intelligent programmer will take every opportunity to reuse code. Because many database-driven applications often require the use of a core task set, there are ample opportunities to reuse code. Such tasks often seek to manipulate database data, producing some sort of outcome based on the retrieved data. As a result, it would be quite convenient if the task results could be directly returned via the SQL query, like so:

```
sqlite>SELECT convert_salary_to_gold(salary)
   ...> FROM employee WHERE empID=1";
```

PHP's SQLite library offers a means for registering and maintaining customized functions such as this. This section shows you how it's accomplished.

sqlite_create_function()

```
boolean sqlite_create_function (resource dbh, string func, mixed callback
                                [, int num_args])
```

The sqlite_create_function() function enables you to register custom PHP functions as SQLite user-defined functions (UDFs). For example, this function would be used to register the convert_salary_to_gold() function discussed in the opening paragraphs of this section, like so:

```
<?php
   /* Define gold's current price-per-ounce. */
   define("PPO",400);

   /* Calculate how much gold an employee can purchase with salary. */
   function convert_salary_to_gold($salary)
   {
      return $salary / PPO;
   }

   /* Connect to the SQLite database. */
   $sqldb = sqlite_open("mydatabase.db");
```

```
    /* Create the user-defined function. */
    sqlite_create_function($sqldb,"salarytogold", "convert_salary_to_gold", 1);

    /* Query the database using the UDF. */
    $query = "select salarytogold(salary) FROM employee WHERE empid=1";
    $result = sqlite_query($sqldb, $query);
    list($salaryToGold) = sqlite_fetch_array($result);

    /* Display the results. */
    echo "The employee can purchase: ".$salaryToGold." ounces.";

    /* End the database connection. */
    sqlite_close($sqldb);
?>
```

Assuming user Jason makes $10,000 per year, you can expect the following output:

```
The employee can purchase 25 ounces.
```

sqlite_udf_encode_binary()

```
string sqlite_udf_encode_binary (string data)
```

The `sqlite_udf_encode_binary()` function encodes any binary data intended for storage within an SQLite table. Use this function instead of `sqlite_escape_string()` when you're working with data sent to a UDF.

sqlite_udf_decode_binary()

```
string sqlite_udf_decode_binary (string data)
```

The `sqlite_udf_decode_binary()` function decodes any binary data previously encoded with the `sqlite_udf_encode_binary()` function. Use this function when you're returning possibly binary unsafe data from a UDF.

Creating Aggregate Functions

When you work with database-driven applications, it's often useful to derive some value based on some collective calculation of all values found within a particular column or set of columns. Several such functions are commonly made available within a SQL server's core functionality set. A few such commonly implemented functions, known as aggregate functions, include `sum()`, `max()`, and `min()`. However, you might require a custom aggregate function not otherwise available within the server's default capabilities. SQLite compensates for this by offering the ability to create your own. The function used to register your custom aggregate functions, `sqlite_create_aggregate()`, is introduced in this section.

sqlite_create_aggregate()

```
boolean sqlite_create_aggregate (resource dbh, string func, mixed step_func,
                                 mixed final_func [, int num_args])
```

The `sqlite_create_aggregate()` function is used to register a user-defined aggregate function, `step_func`. Actually it registers two functions: `step_func`, which is called on every row of the query target, and `final_func`, which is used to return the aggregate value back to the caller. Once registered, you can call `final_func` within the caller by the alias `func`. The optional `num_args` parameter specifies the number of parameters the aggregate function should take. Although the SQLite parser attempts to discern the number if this parameter is omitted, you should always include it for clarity's sake.

Consider an example. Building on the salary conversion example from the previous section, suppose you want to calculate the total amount of gold employees could collectively purchase:

```
<?php
   /* Define gold's current price-per-ounce. */
   define("PPO",400);

   /* Create the aggregate function. */
   function total_salary(&$total,$salary)
   {
      $total += $salary;
   }

   /* Create the aggregate finalization function. */
   function convert_to_gold(&$total)
   {
      return $total / PPO;
   }

   /* Connect to the SQLite database. */
   $sqldb = sqlite_open("mydatabase.db");

   /* Register the aggregate function. */
   sqlite_create_aggregate($sqldb, "computetotalgold", "total_salary",
        "convert_to_gold",1);

   /* Query the database using the UDF. */
   $query = "select computetotalgold(salary) FROM employee";
   $result = sqlite_query($sqldb, $query);
   list($salaryToGold) = sqlite_fetch_array($result);

   /* Display the results. */
   echo "The employees can purchase: ".$salaryToGold." ounces.";

   /* End the database connection. */
   sqlite_close($sqldb);
?>
```

If your employees' salaries total $16,000, you could expect the following output:

```
The employees can purchase 40 ounces.
```

Summary

The administrative overhead required of many database servers often outweighs the advantages of added power they offer to many projects. SQLite offers an ideal remedy to this dilemma, providing a fast and capable back end at a cost of minimum maintenance. Given SQLite's commitment to standards, ideal licensing arrangements, and quality, consider saving yourself time, resources, and money by using SQLite for your future projects.