SQLite

Introduction

This is an extension for the SQLite Embeddable SQL Database Engine. SQLite is a C library that implements an embeddable SQL database engine. Programs that link with the SQLite library can have SQL database access without running a separate RDBMS process.

SQLite is not a client library used to connect to a big database server. SQLite is the server. The SQLite library reads and writes directly to and from the database files on disk.

Note
For further information see the SQLite Website (» http://sqlite.org/).

Installing/Configuring

Requirements

In order to have these functions available, you must compile PHP with SQLite support, or load the SQLite extension dynamically from your *php.ini*.

Installation

Read the INSTALL file, which comes with the package. Or just use the PEAR installer with *pecl install sqlite*. SQLite itself is already included, You do not need to install any additional software.

Windows users will enable *php_sqlite.dll* inside of *php.ini* in order to use these functions. The DLL for this PECL extension may be downloaded from either the » PHP Downloads page or from » http://pecl4win.php.net/

In PHP 5, the SQLite extension and the engine itself are bundled and compiled by default. However, since PHP 5.1.0 you need to manually activate the extension in *php.ini* (because it is now bundled as shared). Moreover, since PHP 5.1.0 SQLite depends on PDO it must be enabled too, by adding the following lines to *php.ini* (in order):

```
extension=php_pdo.dll
extension=php_sqlite.dll
```

On Linux or Unix operating systems, if you build PDO as a shared extension, you must build SQLite as a shared extension using the *--with-sqlite=shared* configure option.

SQLite 3 is supported through PDO SQLite.

Note

Windows installation for unprivileged accounts

On Windows operating systems, unprivileged accounts don't have the *TMP* environment variable set by default. This will make sqlite create temporary files in the windows directory, which is not desirable. So, you should set the *TMP* environment variable for the web server or the user account the web server is running under. If Apache is your web server, you can accomplish this via a *SetEnv* directive in your *httpd.conf* file. For example:

```
SetEnv TMP c:/temp
```

If you are unable to establish this setting at the server level, you can implement the setting in your script:

```
putenv('TMP=C:/temp');
```

The setting must refer to a directory that the web server has permission to create files in and subsequently write to and delete the files it created. Otherwise, you may receive the following error message: malformed database schema - unable to open a temporary database file for storing temporary tables

Runtime Configuration

The behaviour of these functions is affected by settings in *php.ini*.

SQLite Configure Options

Name	Default	Changeable	Changelog
sqlite.assoc_case	"0"	PHP_INI_ALL	Available since PHP 5.0.0.

For further details and definitions of the PHP_INI_* constants, see the php.ini directives.

Here's a short explanation of the configuration directives.

`sqlite.assoc_case` int
: Whether to use mixed case (*0*), upper case (*1*) or lower case (*2*) hash indexes. This option is primarily useful when you need compatibility with other database systems, where the names of the columns are always returned as uppercase or lowercase, regardless of the case of the actual field names in the database schema. The SQLite library returns the column names in their natural case (that matches the case you used in your schema). When `sqlite.assoc_case` is set to *0* the natural case will be preserved. When it is set to *1* or *2*, PHP will apply case folding on the hash keys to upper- or lower-case the keys, respectively. Use of this option incurs a slight performance penalty, but is MUCH faster than performing the case folding yourself using PHP script.

Resource Types

There are two resources used in the SQLite Interface. The first one is the database connection, the second one the result set.

Predefined Constants

The constants below are defined by this extension, and will only be available when the extension has either been compiled into PHP or dynamically loaded at runtime.

The functions sqlite_fetch_array() and sqlite_current() use a constant for the different types of result arrays. The following constants are defined:

SQLite result type constants

SQLITE_ASSOC (int)
Columns are returned into the array having the field name as the array index.

SQLITE_BOTH (int)
Columns are returned into the array having both a numerical index and the field name as the array index.

SQLITE_NUM (int)
Columns are returned into the array having a numerical index to the fields. This index starts with 0, the first field in the result.

A number of functions may return status codes. The following constants are defined:

SQLite status code constants

SQLITE_OK (int)
Successful result.

SQLITE_ERROR (int)
SQL error or missing database.

SQLITE_INTERNAL (int)
An internal logic error in SQLite.

SQLITE_PERM (int)
Access permission denied.

SQLITE_ABORT (int)
Callback routine requested an abort.

SQLITE_BUSY (int)
The database file is locked.

SQLITE_LOCKED (int)
A table in the database is locked.

SQLITE_NOMEM (int)
Memory allocation failed.

SQLITE_READONLY (int)
Attempt to write a readonly database.

SQLITE_INTERRUPT (int)
Operation terminated internally.

SQLITE_IOERR (int)
Disk I/O error occurred.

SQLITE_CORRUPT (int)
The database disk image is malformed.

SQLITE_NOTFOUND (int)
(Internal) Table or record not found.

SQLITE_FULL (int)
Insertion failed because database is full.

SQLITE_CANTOPEN (int)
Unable to open the database file.

SQLITE_PROTOCOL (int)
Database lock protocol error.

SQLITE_EMPTY (int)
(Internal) Database table is empty.

SQLITE_SCHEMA (int)
The database schema changed.

SQLITE_TOOBIG (int)
Too much data for one row of a table.

SQLITE_CONSTRAINT (int)
Abort due to constraint violation.

SQLITE_MISMATCH (int)
Data type mismatch.

SQLITE_MISUSE (int)
Library used incorrectly.

SQLITE_NOLFS (int)
Uses of OS features not supported on host.

SQLITE_AUTH (int)
Authorized failed.

SQLITE_ROW (int)
Internal process has another row ready.

SQLITE_DONE (int)
Internal process has finished executing.

SQLite Functions

Predefined Classes

SQLiteDatabase

Represents an opened SQLite database.

Constructor

- __construct - construct a new SQLiteDatabase object

Methods

- query - Execute a query
- queryExec - Execute a result-less query
- arrayQuery - Execute a query and return the result as an array
- singleQuery - Execute a query and return either an array for one single column or the value of the first row
- unbufferedQuery - Execute an unbuffered query
- lastInsertRowid - Returns the rowid of the most recently inserted row
- changes - Returns the number of rows changed by the most recent statement
- createAggregate - Register an aggregating UDF for use in SQL statements
- createFunction - Register a UDF for use in SQL statements
- busyTimeout - Sets or disables busy timeout duration
- lastError - Returns the last error code of the most recently encountered error
- fetchColumnTypes - Return an array of column types from a particular table

SQLiteResult

Represents a buffered SQLite result set.

Methods

- fetch - Fetches the next row from the result set as an array
- fetchObject - Fetches the next row from the result set as an object
- fetchSingle - Fetches the first column from the result set as a string
- fetchAll - Fetches all rows from the result set as an array of arrays
- column - Fetches a column from the current row of the result set
- numFields - Returns the number of fields in the result set
- fieldName - Returns the name of a particular field in the result set
- current - Fetches the current row from the result set as an array
- key - Return the current row index
- next - Seek to the next row number
- valid - Returns whether more rows are available
- rewind - Seek to the first row number of the result set
- prev - Seek to the previous row number of the result set
- hasPrev - Returns whether or not a previous row is available
- numRows - Returns the number of rows in the result set
- seek - Seek to a particular row number

SQLiteUnbuffered

Represents an unbuffered SQLite result set. Unbuffered results sets are sequential, forward-seeking only.

Methods

- fetch - Fetches the next row from the result set as an array
- fetchObject - Fetches the next row from the result set as an object
- fetchSingle - Fetches the first column from the result set as a string
- fetchAll - Fetches all rows from the result set as an array of arrays
- column - Fetches a column from the current row of the result set
- numFields - Returns the number of fields in the result set
- fieldName - Returns the name of a particular field in the result set
- current - Fetches the current row from the result set as an array
- next - Seek to the next row number

- valid - Returns whether more rows are available

sqlite_array_query

SQLiteDatabase->arrayQuery

sqlite_array_query -- SQLiteDatabase->arrayQuery -- Execute a query against a given database and returns an array

Description

array **sqlite_array_query** (resource `$dbhandle`, string `$query` [, int `$result_type` [, bool `$decode_binary`]])

array **sqlite_array_query** (string `$query`, resource `$dbhandle` [, int `$result_type` [, bool `$decode_binary`]])

Object oriented style (method):

SQLiteDatabase

array **arrayQuery** (string `$query` [, int `$result_type` [, bool `$decode_binary`]])

sqlite_array_query() executes the given query and returns an array of the entire result set. It is similar to calling sqlite_query() and then sqlite_fetch_array() for each row in the result set. sqlite_array_query() is significantly faster than the aforementioned.

Tip

sqlite_array_query() is best suited to queries returning 45 rows or less. If you have more data than that, it is recommended that you write your scripts to use sqlite_unbuffered_query() instead for more optimal performance.

Parameters

`query`
The query to be executed.

`dbhandle`
The SQLite Database resource; returned from sqlite_open() when used procedurally. This parameter is not required when using the object-oriented method.

`result_type`

The optional *result_type* parameter accepts a constant and determines how the returned array will be indexed. Using **SQLITE_ASSOC** will return only associative indices (named fields) while **SQLITE_NUM** will return only numerical indices (ordinal field numbers). **SQLITE_BOTH** will return both associative and numerical indices. **SQLITE_BOTH** is the default for this function.

decode_binary

When the *decode_binary* parameter is set to **TRUE** (the default), PHP will decode the binary encoding it applied to the data if it was encoded using the sqlite_escape_string() . You should normally leave this value at its default, unless you are interoperating with databases created by other sqlite capable applications.

Note

Two alternative syntaxes are supported for compatibility with other database extensions (such as MySQL). The preferred form is the first, where the *dbhandle* parameter is the first parameter to the function.

Return Values

Returns an array of the entire result set; **FALSE** otherwise.

The column names returned by **SQLITE_ASSOC** and **SQLITE_BOTH** will be case-folded according to the value of the sqlite.assoc_case configuration option.

Examples

Example #1 - Procedural style

```
<?php
$dbhandle = sqlite_open('sqlitedb');
$result = sqlite_array_query($dbhandle, 'SELECT name, email FROM users LIMIT
25', SQLITE_ASSOC);
foreach ($result as $entry) {
   echo 'Name: ' . $entry['name'] . '  E-mail: ' . $entry['email'];
}
?>
```

Example #2 - Object-oriented style

```
<?php
$dbhandle = new SQLiteDatabase('sqlitedb');
$result = $dbhandle->arrayQuery('SELECT name, email FROM users LIMIT 25',
SQLITE_ASSOC);
```

```
foreach ($result as $entry) {
   echo 'Name: ' . $entry['name'] . '  E-mail: ' . $entry['email'];
}
?>
```

See Also

- sqlite_query()
- sqlite_fetch_array()
- sqlite_fetch_string()

sqlite_busy_timeout

SQLiteDatabase->busyTimeout

sqlite_busy_timeout -- SQLiteDatabase->busyTimeout -- Set busy timeout duration, or disable busy handlers

Description

void **sqlite_busy_timeout** (resource `$dbhandle`, int `$milliseconds`)

Object oriented style (method):

SQLiteDatabase

void **busyTimeout** (int `$milliseconds`)

Set the maximum time, in milliseconds, that SQLite will wait for a *`dbhandle`* to become ready for use.

Parameters

`dbhandle`
: The SQLite Database resource; returned from sqlite_open() when used procedurally. This parameter is not required when using the object-oriented method.

`milliseconds`
: The number of milliseconds. When set to *0*, busy handlers will be disabled and SQLite will return immediately with a *SQLITE_BUSY* status code if another process/thread has the database locked for an update. PHP sets the default busy timeout to be 60 seconds when the database is opened.

Note
There are one thousand (1000) milliseconds in one second.

Return Values

No value is returned.

Examples

Example #3 - Procedural style

```
<?php
$dbhandle = sqlite_open('sqlitedb');
sqlite_busy_timeout($dbhandle, 10000); // set timeout to 10 seconds
sqlite_busy_timeout($dbhandle, 0); // disable busy handler
?>
```

Example #4 - Object oriented style

```
<?php
$dbhandle = new SQLiteDatabase('sqlitedb');
$dbhandle->busyTimeout(10000); // 10 seconds
$dbhandle->busyTimeout(0); // disable
?>
```

See Also

- sqlite_open()

sqlite_changes

SQLiteDatabase->changes

sqlite_changes -- SQLiteDatabase->changes -- Returns the number of rows that were changed by the most recent SQL statement

Description

int **sqlite_changes** (resource `$dbhandle`)

Object oriented style (method):

SQLiteDatabase

int **changes** (void)

Returns the numbers of rows that were changed by the most recent SQL statement executed against the *`dbhandle`* database handle.

Parameters

`dbhandle`
The SQLite Database resource; returned from sqlite_open() when used procedurally. This parameter is not required when using the object-oriented method.

Examples

Example #5 - Procedural style

```
<?php
$dbhandle = sqlite_open('mysqlitedb');
$query = sqlite_query($dbhandle, "UPDATE users SET email='jDoe@example.com'
WHERE username='jDoe'");
if (!$query) {
   exit('Error in query.');
} else {
   echo 'Number of rows modified: ', sqlite_changes($dbhandle);
}
?>
```

Example #6 - Object oriented style

```
<?php
$dbhandle = new SQLiteDatabase('mysqlitedb');
$query = $dbhandle->query("UPDATE users SET email='jDoe@example.com' WHERE
username='jDoe'");
if (!$query) {
   exit('Error in query.');
} else {
   echo 'Number of rows modified: ', $dbhandle->changes();
}
?>
```

See Also

- sqlite_open()

sqlite_close

sqlite_close -- Closes an open SQLite database

Description

void **sqlite_close** (resource `$dbhandle`)

Closes the given *`database`* handle. If the database was persistent, it will be closed and removed from the persistent list.

Parameters

`dbhandle`
The SQLite Database resource; returned from sqlite_open() when used procedurally.

Return Values

No value is returned.

Examples

Example #7 - sqlite_close() example

```
<?php
$dbhandle = sqlite_open('sqlitedb');
sqlite_close($dbhandle);
?>
```

See Also

- sqlite_open()
- sqlite_popen()

sqlite_column

SQLiteResult->column

SQLiteUnbuffered->column

sqlite_column -- SQLiteResult->column -- SQLiteUnbuffered->column -- Fetches a column from the current row of a result set

Description

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

SQLiteResult

mixed **column** (mixed `$index_or_name` [, bool `$decode_binary`])

SQLiteUnbuffered

mixed **column** (mixed `$index_or_name` [, bool `$decode_binary`])

Fetches the value of a column named *`index_or_name`* (if it is a string), or of the ordinal column numbered *`index_or_name`* (if it is an integer) from the current row of the query result handle *`result`*.

Parameters

`result`
: The SQLite result resource. This parameter is not required when using the object-oriented method.

`index_or_name`
: The column index or name to fetch.

`decode_binary`
: When the *`decode_binary`* parameter is set to **TRUE** (the default), PHP will decode the binary encoding it applied to the data if it was encoded using the sqlite_escape_string()

. You should normally leave this value at its default, unless you are interoperating with databases created by other sqlite capable applications.

Notes

Note
Use this function when you are iterating a large result set with many columns, or with columns that contain large amounts of data.

See Also

- sqlite_fetch_string()

sqlite_create_aggregate

SQLiteDatabase->createAggregate

sqlite_create_aggregate -- SQLiteDatabase->createAggregate -- Register an aggregating UDF for use in SQL statements

Description

void **sqlite_create_aggregate** (resource `$dbhandle`, string `$function_name`, callback `$step_func`, callback `$finalize_func` [, int `$num_args`])

Object oriented style (method):

SQLiteDatabase

void **createAggregate** (string `$function_name`, callback `$step_func`, callback `$finalize_func` [, int `$num_args`])

sqlite_create_aggregate() is similar to sqlite_create_function() except that it registers functions that can be used to calculate a result aggregated across all the rows of a query.

The key difference between this function and sqlite_create_function() is that two functions are required to manage the aggregate; *`step_func`* is called for each row of the result set. Your PHP function should accumulate the result and store it into the aggregation context. Once all the rows have been processed, *`finalize_func`* will be called and it should then take the data from the aggregation context and return the result. Callback functions should return a type understood by SQLite (i.e. scalar type).

Parameters

`dbhandle`
The SQLite Database resource; returned from sqlite_open() when used procedurally. This parameter is not required when using the object-oriented method.

`function_name`
The name of the function used in SQL statements.

`step_func`
Callback function called for each row of the result set.

`finalize_func`
Callback function to aggregate the "stepped" data from each row.

num_args
: Hint to the SQLite parser if the callback function accepts a predetermined number of arguments.

Return Values

No value is returned.

Examples

Example #8 - 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'));

?>
```

In this example, we are creating an aggregating function that will calculate the length of the

longest string in one of the columns of the table. For each row, the *max_len_step* function is called and passed a `context` parameter. The context parameter is just like any other PHP variable and be set to hold an array or even an object value. In this example, we are simply using it to hold the maximum length we have seen so far; if the `string` has a length longer than the current maximum, we update the context to hold this new maximum length.

After all of the rows have been processed, SQLite calls the *max_len_finalize* function to determine the aggregate result. Here, we could perform some kind of calculation based on the data found in the `context`. In our simple example though, we have been calculating the result as the query progressed, so we simply need to return the context value.

Note
The example above will not work correctly if the column contains binary data. Take a look at the manual page for sqlite_udf_decode_binary() for an explanation of why this is so, and an example of how to make it respect the binary encoding.

Tip
It is NOT recommended for you to store a copy of the values in the context and then process them at the end, as you would cause SQLite to use a lot of memory to process the query - just think of how much memory you would need if a million rows were stored in memory, each containing a string 32 bytes in length.

Tip
You can use sqlite_create_function() and sqlite_create_aggregate() to override SQLite native SQL functions.

See Also

- sqlite_create_function()
- sqlite_udf_encode_binary()
- sqlite_udf_decode_binary()

sqlite_create_function

SQLiteDatabase->createFunction

sqlite_create_function -- SQLiteDatabase->createFunction -- Registers a "regular" User Defined Function for use in SQL statements

Description

void **sqlite_create_function** (resource `$dbhandle`, string `$function_name`, callback `$callback` [, int `$num_args`])

Object oriented style (method):

SQLiteDatabase

void **createFunction** (string `$function_name`, callback `$callback` [, int `$num_args`])

sqlite_create_function() allows you to register a PHP function with SQLite as an UDF (User Defined Function), so that it can be called from within your SQL statements.

The UDF can be used in any SQL statement that can call functions, such as SELECT and UPDATE statements and also in triggers.

Parameters

`dbhandle`
: The SQLite Database resource; returned from sqlite_open() when used procedurally. This parameter is not required when using the object-oriented method.

`function_name`
: The name of the function used in SQL statements.

`callback`
: Callback function to handle the defined SQL function.

Note
Callback functions should return a type understood by SQLite (i.e. scalar type).

`num_args`

Hint to the SQLite parser if the callback function accepts a predetermined number of arguments.

Note

Two alternative syntaxes are supported for compatibility with other database extensions (such as MySQL). The preferred form is the first, where the *dbhandle* parameter is the first parameter to the function.

Return Values

No value is returned.

Examples

Example #9 - sqlite_create_function() example

```
<?php
function md5_and_reverse($string)
{
   return strrev(md5($string));
}

if ($dbhandle = sqlite_open('mysqlitedb', 0666, $sqliteerror)) {

   sqlite_create_function($dbhandle, 'md5rev', 'md5_and_reverse', 1);

   $sql  = 'SELECT md5rev(filename) FROM files';
   $rows = sqlite_array_query($dbhandle, $sql);
} else {
   echo 'Error opening sqlite db: ' . $sqliteerror;
   exit;
}
?>
```

In this example, we have a function that calculates the md5 sum of a string, and then reverses it. When the SQL statement executes, it returns the value of the filename transformed by our function. The data returned in *$rows* contains the processed result.

The beauty of this technique is that you do not need to process the result using a foreach() loop after you have queried for the data.

PHP registers a special function named *php* when the database is first opened. The php function can be used to call any PHP function without having to register it first.

Example #10 - Example of using the PHP function

```
<?php
$rows = sqlite_array_query($dbhandle, "SELECT php('md5', filename) from
files");
?>
```

This example will call the md5() on each *filename* column in the database and return the result into `$rows`

Note

For performance reasons, PHP will not automatically encode/decode binary data passed to and from your UDF's. You need to manually encode/decode the parameters and return values if you need to process binary data in this way. Take a look at sqlite_udf_encode_binary() and sqlite_udf_decode_binary() for more details.

Tip

It is not recommended to use UDF's to handle processing of binary data, unless high performance is not a key requirement of your application.

Tip

You can use sqlite_create_function() and sqlite_create_aggregate() to override SQLite native SQL functions.

See Also

- sqlite_create_aggregate()

sqlite_current

SQLiteResult->current

SQLiteUnbuffered->current

sqlite_current -- SQLiteResult->current -- SQLiteUnbuffered->current -- Fetches the current row from a result set as an array

Description

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

Object oriented style (method):

SQLiteResult

array **current** ([int `$result_type` [, bool `$decode_binary`]])

SQLiteUnbuffered

array **current** ([int `$result_type` [, bool `$decode_binary`]])

sqlite_current() is identical to sqlite_fetch_array() except that it does not advance to the next row prior to returning the data; it returns the data from the current position only.

Parameters

`result`
: The SQLite result resource. This parameter is not required when using the object-oriented method.

`result_type`
: The optional *`result_type`* parameter accepts a constant and determines how the returned array will be indexed. Using **SQLITE_ASSOC** will return only associative indices (named fields) while **SQLITE_NUM** will return only numerical indices (ordinal field numbers). **SQLITE_BOTH** will return both associative and numerical indices. **SQLITE_BOTH** is the default for this function.

decode_binary
: When the *decode_binary* parameter is set to **TRUE** (the default), PHP will decode the binary encoding it applied to the data if it was encoded using the sqlite_escape_string() . You should normally leave this value at its default, unless you are interoperating with databases created by other sqlite capable applications.

Return Values

Returns an array of the current row from a result set; **FALSE** if the current position is beyond the final row.

The column names returned by **SQLITE_ASSOC** and **SQLITE_BOTH** will be case-folded according to the value of the sqlite.assoc_case configuration option.

See Also

- sqlite_seek()
- sqlite_next()
- sqlite_fetch_array()

sqlite_error_string

sqlite_error_string -- Returns the textual description of an error code

Description

string **sqlite_error_string** (int `$error_code`)

Returns a human readable description of the *`error_code`* returned from sqlite_last_error().

Parameters

`error_code`
: The error code being used, which might be passed in from sqlite_last_error().

Return Values

Returns a human readable description of the *`error_code`*, as a string.

See Also

- sqlite_last_error()

sqlite_escape_string

sqlite_escape_string -- Escapes a string for use as a query parameter

Description

string **sqlite_escape_string** (string `$item`)

sqlite_escape_string() will correctly quote the string specified by *`item`* for use in an SQLite SQL statement. This includes doubling up single-quote characters (') and checking for binary-unsafe characters in the query string.

Although the encoding makes it safe to insert the data, it will render simple text comparisons and *LIKE* clauses in your queries unusable for the columns that contain the binary data. In practice, this shouldn't be a problem, as your schema should be such that you don't use such things on binary columns (in fact, it might be better to store binary data using other means, such as in files).

Parameters

`item`
: The string being quoted. If the *`item`* contains a *NUL* character, or if it begins with a character whose ordinal value is *0x01*, PHP will apply a binary encoding scheme so that you can safely store and retrieve binary data.

Return Values

Returns an escaped string for use in an SQLite SQL statement.

Notes

Note
Do not use this function to encode the return values from UDF's created using sqlite_create_function() or sqlite_create_aggregate() - use sqlite_udf_encode_binary() instead.

Warning
addslashes() should *NOT* be used to quote your strings for SQLite queries; it will lead to strange results when retrieving your data.

See Also

- sqlite_udf_encode_binary()

sqlite_exec

SQLiteDatabase->exec

sqlite_exec -- SQLiteDatabase->exec -- Executes a result-less query against a given database

Description

bool **sqlite_exec** (resource `$dbhandle`, string `$query` [, string `&$error_msg`])

bool **sqlite_exec** (string `$query`, resource `$dbhandle`)

Object oriented style (method):

SQLiteDatabase

bool **queryExec** (string `$query` [, string `&$error_msg`])

Executes an SQL statement given by the *`query`* against a given database handle (specified by the *`dbhandle`* parameter).

Warning

SQLite *will* execute multiple queries separated by semicolons, so you can use it to execute a batch of SQL that you have loaded from a file or have embedded in a script.

Parameters

`dbhandle`
: The SQLite Database resource; returned from sqlite_open() when used procedurally. This parameter is not required when using the object-oriented method.

`query`
: The query to be executed.

`error_msg`
: The specified variable will be filled if an error occurs. This is specially important because SQL syntax errors can't be fetched using the sqlite_last_error() function.

Note
Two alternative syntaxes are supported for compatibility with other database extensions (such as MySQL). The preferred form is the first, where the *dbhandle* parameter is the first parameter to the function.

Return Values

This function will return a boolean result; **TRUE** for success or **FALSE** for failure. If you need to run a query that returns rows, see sqlite_query().

The column names returned by **SQLITE_ASSOC** and **SQLITE_BOTH** will be case-folded according to the value of the sqlite.assoc_case configuration option.

ChangeLog

Version	Description
5.1.0	Added the *error_msg* parameter

Examples

Example #11 - Procedural example

```
<?php
$dbhandle = sqlite_open('mysqlitedb');
$query = sqlite_exec($dbhandle, "UPDATE users SET email='jDoe@example.com'
WHERE username='jDoe'", $error);
if (!$query) {
   exit("Error in query: '$error'");
} else {
   echo 'Number of rows modified: ', sqlite_changes($dbhandle);
}
?>
```

Example #12 - Object-oriented example

```
<?php
$dbhandle = new SQLiteDatabase('mysqlitedb');
$query = $dbhandle->queryExec("UPDATE users SET email='jDoe@example.com'
WHERE username='jDoe'", $error);
```

```
if (!$query) {
   exit("Error in query: '$error'");
} else {
   echo 'Number of rows modified: ', $dbhandle->changes();
}
?>
```

See Also

- sqlite_query()
- sqlite_unbuffered_query()
- sqlite_array_query()

sqlite_factory

sqlite_factory -- Opens a SQLite database and returns a SQLiteDatabase object

Description

SQLiteDatabase **sqlite_factory** (string `$filename` [, int `$mode` [, string `&$error_message`]])

sqlite_factory() behaves similarly to sqlite_open() in that it opens an SQLite database or attempts to create it if it does not exist. However, a SQLiteDatabase object is returned rather than a resource. Please see the sqlite_open() reference page for further usage and caveats.

Parameters

`filename`
: The filename of the SQLite database.

`mode`
: The mode of the file. Intended to be used to open the database in read-only mode. Presently, this parameter is ignored by the sqlite library. The default value for mode is the octal value *0666* and this is the recommended value.

`error_message`
: Passed by reference and is set to hold a descriptive error message explaining why the database could not be opened if there was an error.

Return Values

Returns a SQLiteDatabase object on success, **NULL** on error.

Examples

Example #13 - sqlite_factory() example

```
<?php
$dbhandle = sqlite_factory('sqlitedb');
$dbhandle->query('SELECT user_id, username FROM users');

/* functionally equivalent to: */

$dbhandle = new SQLiteDatabase('sqlitedb');
$dbhandle->query('SELECT user_id, username FROM users');

?>
```

See Also

- sqlite_open()
- sqlite_popen()

sqlite_fetch_all

SQLiteResult->fetchAll

SQLiteUnbuffered->fetchAll

sqlite_fetch_all -- SQLiteResult->fetchAll -- SQLiteUnbuffered->fetchAll -- Fetches all rows from a result set as an array of arrays

Description

array **sqlite_fetch_all** (resource `$result` [, int `$result_type` [, bool `$decode_binary`]])

Object oriented style (method):

SQLiteResult

array **fetchAll** ([int `$result_type` [, bool `$decode_binary`]])

SQLiteUnbuffered

array **fetchAll** ([int `$result_type` [, bool `$decode_binary`]])

sqlite_fetch_all() returns an array of the entire result set from the *`result`* resource. It is similar to calling sqlite_query() (or sqlite_unbuffered_query()) and then sqlite_fetch_array() for each row in the result set.

Parameters

`result`
The SQLite result resource. This parameter is not required when using the object-oriented method.

`result_type`
The optional *`result_type`* parameter accepts a constant and determines how the returned array will be indexed. Using **SQLITE_ASSOC** will return only associative indices (named fields) while **SQLITE_NUM** will return only numerical indices (ordinal field numbers). **SQLITE_BOTH** will return both associative and numerical indices.

SQLITE_BOTH is the default for this function.

decode_binary
When the *decode_binary* parameter is set to **TRUE** (the default), PHP will decode the binary encoding it applied to the data if it was encoded using the sqlite_escape_string() . You should normally leave this value at its default, unless you are interoperating with databases created by other sqlite capable applications.

Return Values

Returns an array of the remaining rows in a result set. If called right after sqlite_query(), it returns all rows. If called after sqlite_fetch_array(), it returns the rest. If there are no rows in a result set, it returns an empty array.

The column names returned by **SQLITE_ASSOC** and **SQLITE_BOTH** will be case-folded according to the value of the sqlite.assoc_case configuration option.

Examples

Example #14 - Procedural example

```
<?php
$dbhandle = sqlite_open('sqlitedb');
$query = sqlite_query($dbhandle, 'SELECT name, email FROM users LIMIT 25');
$result = sqlite_fetch_all($query, SQLITE_ASSOC);
foreach ($result as $entry) {
   echo 'Name: ' . $entry['name'] . '  E-mail: ' . $entry['email'];
}
?>
```

Example #15 - Object-oriented example

```
<?php
$dbhandle = new SQLiteDatabase('sqlitedb');

$query = $dbhandle->query('SELECT name, email FROM users LIMIT 25'); //
buffered result set
$query = $dbhandle->unbufferedQuery('SELECT name, email FROM users LIMIT
25'); // unbuffered result set

$result = $query->fetchAll(SQLITE_ASSOC);
foreach ($result as $entry) {
   echo 'Name: ' . $entry['name'] . '  E-mail: ' . $entry['email'];
}
?>
```

See Also

- sqlite_fetch_array()

sqlite_fetch_array

SQLiteResult->fetch

SQLiteUnbuffered->fetch

sqlite_fetch_array -- SQLiteResult->fetch -- SQLiteUnbuffered->fetch -- Fetches the next row from a result set as an array

Description

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

Object oriented style (method):

SQLiteResult

array **fetch** ([int `$result_type` [, bool `$decode_binary`]])

SQLiteUnbuffered

array **fetch** ([int `$result_type` [, bool `$decode_binary`]])

Fetches the next row from the given *`result`* handle. If there are no more rows, returns **FALSE**, otherwise returns an associative array representing the row data.

Parameters

`result`
The SQLite result resource. This parameter is not required when using the object-oriented method.

`result_type`
The optional *`result_type`* parameter accepts a constant and determines how the returned array will be indexed. Using **SQLITE_ASSOC** will return only associative indices (named fields) while **SQLITE_NUM** will return only numerical indices (ordinal field numbers). **SQLITE_BOTH** will return both associative and numerical indices.

SQLITE_BOTH is the default for this function.

decode_binary
When the *decode_binary* parameter is set to **TRUE** (the default), PHP will decode the binary encoding it applied to the data if it was encoded using the sqlite_escape_string() . You should normally leave this value at its default, unless you are interoperating with databases created by other sqlite capable applications.

Return Values

Returns an array of the next row from a result set; **FALSE** if the next position is beyond the final row.

The column names returned by **SQLITE_ASSOC** and **SQLITE_BOTH** will be case-folded according to the value of the sqlite.assoc_case configuration option.

Examples

Example #16 - Procedural example

```
<?php
$dbhandle = sqlite_open('sqlitedb');
$query = sqlite_query($dbhandle, 'SELECT name, email FROM users LIMIT 25');
while ($entry = sqlite_fetch_array($query, SQLITE_ASSOC)) {
   echo 'Name: ' . $entry['name'] . '  E-mail: ' . $entry['email'];
}
?>
```

Example #17 - Object-oriented example

```
<?php
$dbhandle = new SQLiteDatabase('sqlitedb');

$query = $dbhandle->query('SELECT name, email FROM users LIMIT 25'); //
buffered result set
$query = $dbhandle->unbufferedQuery('SELECT name, email FROM users LIMIT
25'); // unbuffered result set

while ($entry = $query->fetch(SQLITE_ASSOC)) {
   echo 'Name: ' . $entry['name'] . '  E-mail: ' . $entry['email'];
}
?>
```

See Also

- sqlite_array_query()
- sqlite_fetch_string()

sqlite_fetch_column_types

SQLiteDatabase->fetchColumnTypes

sqlite_fetch_column_types -- SQLiteDatabase->fetchColumnTypes -- Return an array of column types from a particular table

Description

array **sqlite_fetch_column_types** (string `$table_name`, resource `$dbhandle` [, int `$result_type`])

Object oriented style (method):

SQLiteDatabase

array **fetchColumnTypes** (string `$table_name` [, int `$result_type`])

sqlite_fetch_column_types() returns an array of column data types from the specified *`table_name`* table.

Parameters

`table_name`
: The table name to query.

`dbhandle`
: The SQLite Database resource; returned from sqlite_open() when used procedurally. This parameter is not required when using the object-oriented method.

`result_type`
: The optional *`result_type`* parameter accepts a constant and determines how the returned array will be indexed. Using **SQLITE_ASSOC** will return only associative indices (named fields) while **SQLITE_NUM** will return only numerical indices (ordinal field numbers). **SQLITE_BOTH** will return both associative and numerical indices. **SQLITE_ASSOC** is the default for this function.

Return Values

Returns an array of column data types; **FALSE** on error.

The column names returned by **SQLITE_ASSOC** and **SQLITE_BOTH** will be case-folded

according to the value of the sqlite.assoc_case configuration option.

ChangeLog

Version	Description
5.1.0	Added *result_type*

Examples

Example #18 - Procedural example

```
<?php
$db = sqlite_open('mysqlitedb');
sqlite_query($db, 'CREATE TABLE foo (bar varchar(10), arf text)');
$cols = sqlite_fetch_column_types('foo', $db, SQLITE_ASSOC);

foreach ($cols as $column => $type) {
   echo "Column: $column  Type: $type";
}
?>
```

Example #19 - Object-oriented example

```
<?php
$db = new SQLiteDatabase('mysqlitedb');
$db->query('CREATE TABLE foo (bar varchar(10), arf text)');
$cols = $db->fetchColumnTypes('foo', SQLITE_ASSOC);

foreach ($cols as $column => $type) {
   echo "Column: $column  Type: $type";
}
?>
```

The above example will output:

```
Column: bar  Type: VARCHAR
Column: arf  Type: TEXT
```

sqlite_fetch_object

SQLiteResult->fetchObject

SQLiteUnbuffered->fetchObject

sqlite_fetch_object -- SQLiteResult->fetchObject -- SQLiteUnbuffered->fetchObject -- Fetches the next row from a result set as an object

Description

object **sqlite_fetch_object** (resource `$result` [, string `$class_name` [, array `$ctor_params` [, bool `$decode_binary`]]])

Object oriented style (method):

SQLiteResult

object **fetchObject** ([string `$class_name` [, array `$ctor_params` [, bool `$decode_binary`]]])

SQLiteUnbuffered

object **fetchObject** ([string `$class_name` [, array `$ctor_params` [, bool `$decode_binary`]]])

Warning

This function is currently not documented; only its argument list is available.

sqlite_fetch_single

SQLiteResult->fetchSingle

SQLiteUnbuffered->fetchSingle

sqlite_fetch_single -- SQLiteResult->fetchSingle -- SQLiteUnbuffered->fetchSingle -- Fetches the first column of a result set as a string

Description

string **sqlite_fetch_single** (resource `$result` [, bool `$decode_binary`])

Object oriented style (method):

SQLiteResult

string **fetchSingle** ([bool `$decode_binary`])

SQLiteUnbuffered

string **fetchSingle** ([bool `$decode_binary`])

sqlite_fetch_single() is identical to sqlite_fetch_array() except that it returns the value of the first column of the rowset.

This is the most optimal way to retrieve data when you are only interested in the values from a single column of data.

Parameters

`result`
: The SQLite result resource. This parameter is not required when using the object-oriented method.

`decode_binary`
: When the *`decode_binary`* parameter is set to **TRUE** (the default), PHP will decode the binary encoding it applied to the data if it was encoded using the sqlite_escape_string()

. You should normally leave this value at its default, unless you are interoperating with databases created by other sqlite capable applications.

Examples

Example #20 - A sqlite_fetch_single() example

```
<?php
if ($dbhandle = sqlite_open('mysqlitedb', 0666, $sqliteerror)) {

   $sql = "SELECT id FROM sometable WHERE id = 42";
   $res = sqlite_query($dbhandle, $sql);

   if (sqlite_num_rows($res) > 0) {
       echo sqlite_fetch_single($res); // 42
   }

   sqlite_close($dbhandle);
}
?>
```

See Also

- sqlite_fetch_array()

sqlite_fetch_string

sqlite_fetch_string -- Alias of sqlite_fetch_single()

Description

This function is an alias of: sqlite_fetch_single().

sqlite_field_name

SQLiteResult->fieldName

SQLiteUnbuffered->fieldName

sqlite_field_name -- SQLiteResult->fieldName -- SQLiteUnbuffered->fieldName -- Returns the name of a particular field

Description

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

Object oriented style (method):

SQLiteResult

string **fieldName** (int `$field_index`)

SQLiteUnbuffered

string **fieldName** (int `$field_index`)

Given the ordinal column number, *`field_index`*, sqlite_field_name() returns the name of that field in the result set *`result`*.

Parameters

`result`
: The SQLite result resource. This parameter is not required when using the object-oriented method.

`field_index`
: The ordinal column number in the result set.

Return Values

Returns the name of a field in an SQLite result set, given the ordinal column number; **FALSE** on error.

The column names returned by **SQLITE_ASSOC** and **SQLITE_BOTH** will be case-folded according to the value of the sqlite.assoc_case configuration option.

sqlite_has_more

sqlite_has_more -- Finds whether or not more rows are available

Description

bool **sqlite_has_more** (resource `$result`)

Finds whether more rows are available from the given result set.

Parameters

`result`
The SQLite result resource.

Return Values

Returns **TRUE** if there are more rows available from the *`result`* handle, or **FALSE** otherwise.

See Also

- sqlite_num_rows()
- sqlite_changes()

sqlite_has_prev

SQLiteResult->hasPrev

sqlite_has_prev -- SQLiteResult->hasPrev -- Returns whether or not a previous row is available

Description

bool **sqlite_has_prev** (resource `$result`)

Object oriented style (method):

SQLiteResult

bool **hasPrev** (void)

Find whether there are more previous rows from the given result handle.

Parameters

`result`
The SQLite result resource. This parameter is not required when using the object-oriented method.

Note
This function cannot be used with unbuffered result handles.

Return Values

Returns **TRUE** if there are more previous rows available from the *`result`* handle, or **FALSE** otherwise.

See Also

- sqlite_prev()

- sqlite_has_more()
- sqlite_num_rows()

sqlite_key

SQLiteResult->key

sqlite_key -- SQLiteResult->key -- Returns the current row index

Description

int **sqlite_key** (resource `$result`)

Object oriented style (method):

SQLiteResult

int **key** (void)

sqlite_key() returns the current row index of the buffered result set *`result`*.

Parameters

`result`
The SQLite result resource. This parameter is not required when using the object-oriented method.

Note
This function cannot be used with unbuffered result handles.

Return Values

Returns the current row index of the buffered result set *`result`*.

ChangeLog

Version	Description
5.0.4	Prior to PHP 5.0.4, sqlite_key() was only

	able to be called as a method on a SQLiteResult object, not procedurally.

See Also

- sqlite_next()
- sqlite_current()
- sqlite_rewind()

sqlite_last_error

SQLiteDatabase->lastError

sqlite_last_error -- SQLiteDatabase->lastError -- Returns the error code of the last error for a database

Description

int **sqlite_last_error** (resource `$dbhandle`)

Object oriented style (method):

SQLiteDatabase

int **lastError** (void)

Returns the error code from the last operation performed on *`dbhandle`* (the database handle), or *0* when no error occurred. A human readable description of the error code can be retrieved using sqlite_error_string().

Parameters

`dbhandle`
: The SQLite Database resource; returned from sqlite_open() when used procedurally. This parameter is not required when using the object-oriented method.

See Also

- sqlite_error_string()

sqlite_last_insert_rowid

SQLiteDatabase->lastInsertRowid

sqlite_last_insert_rowid -- SQLiteDatabase->lastInsertRowid -- Returns the rowid of the most recently inserted row

Description

int **sqlite_last_insert_rowid** (resource `$dbhandle`)

Object oriented style (method):

SQLiteDatabase

int **lastInsertRowid** (void)

Returns the rowid of the row that was most recently inserted into the database *`dbhandle`*, if it was created as an auto-increment field.

Tip
You can create auto-increment fields in SQLite by declaring them as *INTEGER PRIMARY KEY* in your table schema.

Parameters

`dbhandle`
: The SQLite Database resource; returned from sqlite_open() when used procedurally. This parameter is not required when using the object-oriented method.

sqlite_libencoding

sqlite_libencoding -- Returns the encoding of the linked SQLite library

Description

string **sqlite_libencoding** (void)

The SQLite library may be compiled in either ISO-8859-1 or UTF-8 compatible modes. This function allows you to determine which encoding scheme is used by your version of the library.

Warning
The default PHP distribution builds libsqlite in ISO-8859-1 encoding mode. However, this is a misnomer; rather than handling ISO-8859-1, it operates according to your current locale settings for string comparisons and sort ordering. So, rather than ISO-8859-1, you should think of it as being '8-bit' instead.

When compiled with UTF-8 support, sqlite handles encoding and decoding of UTF-8 multi-byte character sequences, but does not yet do a complete job when working with the data (no normalization is performed for example), and some comparison operations may still not be carried out correctly.

Warning
It is not recommended that you use PHP in a web-server configuration with a version of the SQLite library compiled with UTF-8 support, since libsqlite will abort the process if it detects a problem with the UTF-8 encoding.

See Also

- **sqlite_lib_version()**

sqlite_libversion

sqlite_libversion -- Returns the version of the linked SQLite library

Description

string **sqlite_libversion** (void)

Returns the version of the linked SQLite library.

See Also

- sqlite_libencoding()

sqlite_next

SQLiteResult->next

SQLiteUnbuffered->next

sqlite_next -- SQLiteResult->next -- SQLiteUnbuffered->next -- Seek to the next row number

Description

bool **sqlite_next** (resource `$result`)

Object oriented style (method):

SQLiteResult

bool **next** (void)

SQLiteUnbuffered

bool **next** (void)

sqlite_next() advances the result handle *`result`* to the next row.

Parameters

`result`
The SQLite result resource. This parameter is not required when using the object-oriented method.

Return Values

Returns **TRUE** on success, or **FALSE** if there are no more rows.

See Also

- sqlite_seek()
- sqlite_current()
- sqlite_rewind()

sqlite_num_fields

SQLiteResult->numFields

SQLiteUnbuffered->numFields

sqlite_num_fields -- SQLiteResult->numFields -- SQLiteUnbuffered->numFields -- Returns the number of fields in a result set

Description

int **sqlite_num_fields** (resource `$result`)

Object oriented style (method):

SQLiteResult

int **numFields** (void)

SQLiteUnbuffered

int **numFields** (void)

Returns the number of fields in the *`result`* set.

Parameters

`result`
: The SQLite result resource. This parameter is not required when using the object-oriented method.

See Also

- sqlite_changes()

- sqlite_num_rows()

sqlite_num_rows

SQLiteResult->numRows

sqlite_num_rows -- SQLiteResult->numRows -- Returns the number of rows in a buffered result set

Description

int **sqlite_num_rows** (resource `$result`)

Object oriented style (method):

SQLiteResult

int **numRows** (void)

Returns the number of rows in the buffered *`result`* set.

Parameters

`result`
The SQLite result resource. This parameter is not required when using the object-oriented method.

Note
This function cannot be used with unbuffered result handles.

Examples

Example #21 - Procedural example

```
<?php
$db = sqlite_open('mysqlitedb');
$result = sqlite_query($db, "SELECT * FROM mytable WHERE name='John Doe'");
$rows = sqlite_num_rows($result);
```

```
echo "Number of rows: $rows";
?>
```

Example #22 - Object-oriented example

```
<?php
$db = new SQLiteDatabase('mysqlitedb');
$result = $db->query("SELECT * FROM mytable WHERE name='John Doe'");
$rows = $result->numRows();

echo "Number of rows: $rows";
?>
```

See Also

- sqlite_changes()
- sqlite_query()
- sqlite_num_fields()

sqlite_open

sqlite_open -- Opens a SQLite database and create the database if it does not exist

Description

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

Object oriented style (constructor):

SQLiteDatabase

__construct (string `$filename` [, int `$mode` [, string `&$error_message`]])

Opens a SQLite database or creates the database if it does not exist.

Parameters

`filename`
: The filename of the SQLite database. If the file does not exist, SQLite will attempt to create it. PHP must have write permissions to the file if data is inserted, the database schema is modified or to create the database if it does not exist.

`mode`
: The mode of the file. Intended to be used to open the database in read-only mode. Presently, this parameter is ignored by the sqlite library. The default value for mode is the octal value *0666* and this is the recommended value.

`error_message`
: Passed by reference and is set to hold a descriptive error message explaining why the database could not be opened if there was an error.

Return Values

Returns a resource (database handle) on success, **FALSE** on error.

Examples

Example #23 - sqlite_open() example

```
<?php
if ($db = sqlite_open('mysqlitedb', 0666, $sqliteerror)) {
   sqlite_query($db, 'CREATE TABLE foo (bar varchar(10))');
   sqlite_query($db, "INSERT INTO foo VALUES ('fnord')");
   $result = sqlite_query($db, 'select bar from foo');
   var_dump(sqlite_fetch_array($result));
} else {
   die($sqliteerror);
}
?>
```

Notes

Tip

On Unix platforms, SQLite is sensitive to scripts that use the fork() system call. If you do have such a script, it is recommended that you close the handle prior to forking and then re-open it in the child and/or parent. For more information on this issue, see » The C language interface to the SQLite library in the section entitled *Multi-Threading And SQLite*.

Tip

It is not recommended to work with SQLite databases mounted on NFS partitions. Since NFS is notoriously bad when it comes to locking you may find that you cannot even open the database at all, and if it succeeds, the locking behaviour may be undefined.

Note

Starting with SQLite library version 2.8.2, you can specify*:memory:* as the `filename` to create a database that lives only in the memory of the computer. This is useful mostly for temporary processing, as the in-memory database will be destroyed when the process ends. It can also be useful when coupled with the *ATTACH DATABASE* SQL statement to load other databases and move and query data between them.

Note

SQLite is safe mode and open_basedir aware.

See Also

- sqlite_popen()
- sqlite_close()
- sqlite_factory()

sqlite_popen

sqlite_popen -- Opens a persistent handle to an SQLite database and create the database if it does not exist

Description

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

This function behaves identically to sqlite_open() except that is uses the persistent resource mechanism of PHP. For information about the meaning of the parameters, read the sqlite_open() manual page.

sqlite_popen() will first check to see if a persistent handle has already been opened for the given *`filename`*. If it finds one, it returns that handle to your script, otherwise it opens a fresh handle to the database.

The benefit of this approach is that you don't incur the performance cost of re-reading the database and index schema on each page hit served by persistent web server SAPI's (any SAPI except for regular CGI or CLI).

Note
If you use persistent handles and have the database updated by a background process (perhaps via a crontab), and that process re-creates the database by overwriting it (either by unlinking and rebuilding, or moving the updated version to replace the current version), you may experience undefined behaviour when a persistent handle on the old version of the database is recycled. To avoid this situation, have your background processes open the same database file and perform their updates in a transaction.

Parameters

`filename`
: The filename of the SQLite database. If the file does not exist, SQLite will attempt to create it. PHP must have write permissions to the file if data is inserted, the database schema is modified or to create the database if it does not exist.

`mode`
: The mode of the file. Intended to be used to open the database in read-only mode. Presently, this parameter is ignored by the sqlite library. The default value for mode is the octal value *0666* and this is the recommended value.

`error_message`
: Passed by reference and is set to hold a descriptive error message explaining why the

database could not be opened if there was an error.

Return Values

Returns a resource (database handle) on success, **FALSE** on error.

See Also

- sqlite_open()
- sqlite_close()
- sqlite_factory()

sqlite_prev

SQLiteResult->prev

sqlite_prev -- SQLiteResult->prev -- Seek to the previous row number of a result set

Description

bool **sqlite_prev** (resource `$result`)

Object oriented style (method):

SQLiteResult

bool **prev** (void)

sqlite_prev() seeks back the *`result`* handle to the previous row.

Parameters

`result`
The SQLite result resource. This parameter is not required when using the object-oriented method.

Note
This function cannot be used with unbuffered result handles.

Return Values

Returns **TRUE** on success, or **FALSE** if there are no more previous rows.

See Also

- sqlite_has_prev()
- sqlite_rewind()
- sqlite_next()

sqlite_query

SQLiteDatabase->query

sqlite_query -- SQLiteDatabase->query -- Executes a query against a given database and returns a result handle

Description

resource **sqlite_query** (resource `$dbhandle`, string `$query` [, int `$result_type` [, string `&$error_msg`]])

resource **sqlite_query** (string `$query`, resource `$dbhandle` [, int `$result_type` [, string `&$error_msg`]])

Object oriented style (method):

SQLiteDatabase

SQLiteResult **query** (string `$query` [, int `$result_type` [, string `&$error_msg`]])

Executes an SQL statement given by the *`query`* against a given database handle.

Parameters

`dbhandle`
: The SQLite Database resource; returned from sqlite_open() when used procedurally. This parameter is not required when using the object-oriented method.

`query`
: The query to be executed.

`result_type`
: The optional *`result_type`* parameter accepts a constant and determines how the returned array will be indexed. Using **SQLITE_ASSOC** will return only associative indices (named fields) while **SQLITE_NUM** will return only numerical indices (ordinal field numbers). **SQLITE_BOTH** will return both associative and numerical indices. **SQLITE_BOTH** is the default for this function.

`error_msg`
: The specified variable will be filled if an error occurs. This is specially important because SQL syntax errors can't be fetched using the sqlite_last_error() function.

Note
Two alternative syntaxes are supported for compatibility with other database extensions (such as MySQL). The preferred form is the first, where the *dbhandle* parameter is the first parameter to the function.

Return Values

This function will return a result handle or **FALSE** on failure. For queries that return rows, the result handle can then be used with functions such as sqlite_fetch_array() and sqlite_seek().

Regardless of the query type, this function will return **FALSE** if the query failed.

sqlite_query() returns a buffered, seekable result handle. This is useful for reasonably small queries where you need to be able to randomly access the rows. Buffered result handles will allocate memory to hold the entire result and will not return until it has been fetched. If you only need sequential access to the data, it is recommended that you use the much higher performance sqlite_unbuffered_query() instead.

ChangeLog

Version	Description
5.1.0	Added the *error_msg* parameter

Notes

Warning
SQLite *will* execute multiple queries separated by semicolons, so you can use it to execute a batch of SQL that you have loaded from a file or have embedded in a script. However, this works only when the result of the function is not used - if it is used, only the first SQL statement would be executed. Function sqlite_exec() will always execute multiple SQL statements. When executing multiple queries, the return value of this function will be **FALSE** if there was an error, but undefined otherwise (it might be **TRUE** for success or it might return a result handle).

See Also

- sqlite_unbuffered_query()
- sqlite_array_query()

sqlite_rewind

SQLiteResult->rewind

sqlite_rewind -- SQLiteResult->rewind -- Seek to the first row number

Description

bool **sqlite_rewind** (resource `$result`)

Object oriented style (method):

SQLiteResult

bool **rewind** (void)

sqlite_rewind() seeks back to the first row in the given result set.

Parameters

`result`
The SQLite result resource. This parameter is not required when using the object-oriented method.

Note
This function cannot be used with unbuffered result handles.

Return Values

Returns **FALSE** if there are no rows in the result set, **TRUE** otherwise.

See Also

- sqlite_next()
- sqlite_current()
- sqlite_seek()

sqlite_seek

SQLiteResult->seek

sqlite_seek -- SQLiteResult->seek -- Seek to a particular row number of a buffered result set

Description

bool **sqlite_seek** (resource `$result`, int `$rownum`)

Object oriented style (method):

SQLiteResult

bool **seek** (int `$rownum`)

sqlite_seek() seeks to the row given by the parameter *`rownum`*.

Parameters

`result`
The SQLite result resource. This parameter is not required when using the object-oriented method.

Note
This function cannot be used with unbuffered result handles.

`rownum`
The ordinal row number to seek to. The row number is zero-based (0 is the first row).

Note
This function cannot be used with unbuffered result handles.

Return Values

Returns **FALSE** if the row does not exist, **TRUE** otherwise.

See Also

- sqlite_next()
- sqlite_current()
- sqlite_rewind()

sqlite_single_query

SQLiteDatabase->singleQuery

sqlite_single_query -- SQLiteDatabase->singleQuery -- Executes a query and returns either an array for one single column or the value of the first row

Description

array **sqlite_single_query** (resource `$db`, string `$query` [, bool `$first_row_only` [, bool `$decode_binary`]])

Object oriented style (method):

SQLiteDatabase

array **singleQuery** (string `$query` [, bool `$first_row_only` [, bool `$decode_binary`]])

Warning

This function is currently not documented; only its argument list is available.

sqlite_udf_decode_binary

sqlite_udf_decode_binary -- Decode binary data passed as parameters to an UDF

Description

string **sqlite_udf_decode_binary** (string `$data`)

Decodes binary data passed as parameters to a UDF.

You must call this function on parameters passed to your UDF if you need them to handle binary data, as the binary encoding employed by PHP will obscure the content and of the parameter in its natural, non-coded form.

PHP does not perform this encode/decode operation automatically as it would severely impact performance if it did.

Parameters

`data`
: The encoded data that will be decoded, data that was applied by either sqlite_udf_encode_binary() or sqlite_escape_string().

Return Values

The decoded string.

Examples

Example #24 - binary-safe max_length aggregation function example

```
<?php
$data = array(
  'one',
  'two',
  'three',
  'four',
  'five',
  'six',
  'seven',
  'eight',
  'nine',
  'ten',
  );
$db = sqlite_open(':memory:');
sqlite_query($db, "CREATE TABLE strings(a)");
foreach ($data as $str) {
```

```
   $str = sqlite_escape_string($str);
   sqlite_query($db, "INSERT INTO strings VALUES ('$str')");
}

function max_len_step(&$context, $string)
{
   $string = sqlite_udf_decode_binary($string);
   if (strlen($string) > $context) {
       $context = strlen($string);
   }
}

function max_len_finalize(&$context)
{
   return $context;
}

sqlite_create_aggregate($db, 'max_len', 'max_len_step', 'max_len_finalize');

var_dump(sqlite_array_query($db, 'SELECT max_len(a) from strings'));

?>
```

See Also

- sqlite_udf_encode_binary()
- sqlite_create_function()
- sqlite_create_aggregate()

sqlite_udf_encode_binary

sqlite_udf_encode_binary -- Encode binary data before returning it from an UDF

Description

string **sqlite_udf_encode_binary** (string `$data`)

sqlite_udf_encode_binary() applies a binary encoding to the *`data`* so that it can be safely returned from queries (since the underlying libsqlite API is not binary safe).

If there is a chance that your data might be binary unsafe (e.g.: it contains a NUL byte in the middle rather than at the end, or if it has and *0x01* byte as the first character) then you must call this function to encode the return value from your UDF.

PHP does not perform this encode/decode operation automatically as it would severely impact performance if it did.

Note
Do not use sqlite_escape_string() to quote strings returned from UDF's as it will lead to double-quoting of the data. Use sqlite_udf_encode_binary() instead!

Parameters

`data`
: The string being encoded.

Return Values

The encoded string.

See Also

- sqlite_udf_decode_binary()
- sqlite_escape_string()
- sqlite_create_function()
- sqlite_create_aggregate()

sqlite_unbuffered_query

SQLiteDatabase->unbufferedQuery

sqlite_unbuffered_query -- SQLiteDatabase->unbufferedQuery -- Execute a query that does not prefetch and buffer all data

Description

resource **sqlite_unbuffered_query** (resource `$dbhandle`, string `$query` [, int `$result_type` [, string `&$error_msg`]])

resource **sqlite_unbuffered_query** (string `$query`, resource `$dbhandle` [, int `$result_type` [, string `&$error_msg`]])

Object oriented style (method):

SQLiteDatabase

SQLiteUnbuffered **unbufferedQuery** (string `$query` [, int `$result_type` [, string `&$error_msg`]])

sqlite_unbuffered_query() is identical to sqlite_query() except that the result that is returned is a sequential forward-only result set that can only be used to read each row, one after the other.

This function is ideal for generating things such as HTML tables where you only need to process one row at a time and don't need to randomly access the row data.

Note

Functions such as sqlite_seek(), sqlite_rewind(), sqlite_next(), sqlite_current(), and sqlite_num_rows() do not work on result handles returned from sqlite_unbuffered_query().

Parameters

`dbhandle`
: The SQLite Database resource; returned from sqlite_open() when used procedurally. This parameter is not required when using the object-oriented method.

query
: The query to be executed.

result_type
: The optional *result_type* parameter accepts a constant and determines how the returned array will be indexed. Using **SQLITE_ASSOC** will return only associative indices (named fields) while **SQLITE_NUM** will return only numerical indices (ordinal field numbers). **SQLITE_BOTH** will return both associative and numerical indices. **SQLITE_BOTH** is the default for this function.

error_msg
: The specified variable will be filled if an error occurs. This is specially important because SQL syntax errors can't be fetched using the sqlite_last_error() function.

Note
Two alternative syntaxes are supported for compatibility with other database extensions (such as MySQL). The preferred form is the first, where the *dbhandle* parameter is the first parameter to the function.

Return Values

Returns a result handle or **FALSE** on failure.

sqlite_unbuffered_query() returns a sequential forward-only result set that can only be used to read each row, one after the other.

ChangeLog

Version	Description
5.1.0	Added the *error_msg* parameter

See Also

- sqlite_query()

sqlite_valid

SQLiteResult->valid

SQLiteUnbuffered->valid

sqlite_valid -- SQLiteResult->valid -- SQLiteUnbuffered->valid -- Returns whether more rows are available

Description

bool **sqlite_valid** (resource `$result`)

Object oriented style (method):

SQLiteResult

bool **valid** (void)

SQLiteUnbuffered

bool **valid** (void)

Finds whether more rows are available from the given result handle.

Parameters

`result`
The SQLite result resource. This parameter is not required when using the object-oriented method.

Note

This function cannot be used with unbuffered result handles.

Return Values

Returns **TRUE** if there are more rows available from the *result* handle, or **FALSE** otherwise.

See Also

- sqlite_num_rows()
- sqlite_changes()