sqlite driver manual

A libdbi driver using the SQLite embedded database engine

Markus Hoenicka

mhoenicka@users.sourceforge.net

sqlite driver manual: A libdbi driver using the SQLite embedded database engine

by Markus Hoenicka

Revision History

Revision $Revision$ $Date$

Table of Contents

List of Tables

Preface

libdbi (http://libdbi.sourceforge.net) is a database abstraction layer written in C. It implements a framework that can utilize separate driver libraries for specific database servers. The libdbi-drivers (http://libdbi-drivers.sourceforge.net) project provides the drivers necessary to talk to the supported database servers.

This manual provides information about the sqlite driver. The manual is intended for programmers who write applications linked against libdbi and who want their applications to work with the sqlite driver.

Questions and comments about the sqlite driver should be sent to the libdbi-drivers-devel (mailto:libdbi-drivers-devel@lists.sourceforge.net) mailing list. Visit the libdbi-drivers-devel list page (http://lists.sourceforge.net/lists/listinfo/libdbi-drivers-devel) to subscribe and for further information. Questions and comments about the libdbi library should be sent to the appropriate libdbi mailing list.

The sqlite driver is maintained by Markus Hoenicka (mailto:mhoenicka@users.sourceforge.net).

Chapter 1. Introduction

SQLite (http://www.sqlite.org) is a smart library that implements an embeddable SQL database engine. No need for an external database server - an application linked against libsqlite can do it all by itself. Of course there are a few limitations of this approach compared to "real" SQL database servers, mostly for massively parallel high-throughput database applications, but on the other hand, installation and administration are a breeze. Your application should support the sqlite driver if one of the following applies:

- You want to support potential users of your application who don’t have the skills to administer a database server.
- You want to offer the simplest possible installation of your application.
- You want to let users test-drive your application without the need to fiddle with their production database servers.

Chapter 2. Installation

This chapter describes the prerequisites and the procedures to build and install the sqlite driver from the sources.

2.1. Prerequisites

The following packages have to be installed on your system:

libdbi

: This library provides the framework of the database abstraction layer which can utilize the sqlite driver to perform database operations. The download page as well as the mailing lists with bug reports and patches are accessible at sourceforge.net/projects/libdbi (http://sourceforge.net/projects/libdbi).

sqlite

: This library implements the embeddable database engine. Find the most recent release at www.sqlite.org (http://www.sqlite.org). The current version of the sqlite driver was tested with SQLite version 2.7.6. and should work ok with later releases.

2.2. Build and install the sqlite driver

First you have to unpack the libdbi-drivers archive in a suitable directory. Unpacking will create a new subdirectory `libdbi-drivers-X.Y` where "X.Y" denotes the version:

```
$ tar -xzf libdbi-drivers-0.3.tar.gz
```

The libdbi-drivers project consists of several drivers that use a common build system. Therefore you *must* tell configure explicitly that you want to build the sqlite driver (you can list as many drivers as you want to build):

```
$ cd libdbi-drivers
```

```
$ ./configure --with-sqlite
```

Run **./configure --help** to find out about additional options.

Then build the driver with the command:

```
$ make
```

> **Note:** Please note that you may have to invoke **gmake**, the GNU version of make, on some systems.

Then install the driver with the command (you'll need root permissions to do this):

```
$ make install
```

To test the operation of the newly installed driver, use the command:

```
$ make check
```

This command creates and runs a test program that performs a few basic input and output tests. The program will ask for a database name. This can be any name that is a valid filename on your system. It will also ask for a data directory. This is the directory that is used to create the test database. Needless to say that you need write access to that directory. If you accept the default ".", the database will be created in the `tests` subdirectory.

> **Note:** If for some reason you need to re-create the autoconf/automake-related files, try running **./autogen.sh**. I've found out that the current stable autoconf/automake/libtool versions (as found in FreeBSD 4.7 and Debian 3.0) do not cooperate well, so I found it necessary to run the older autoconf 2.13. If necessary, edit `autogen.sh` so that it will catch the older autoconf version on your system.

Chapter 3. Driver options

Your application has to initialize libdbi drivers by setting some driver options with the `dbi_conn_set_option()` and the `dbi_conn_set_option_numeric()` library functions. The sqlite driver supports the following options:

dbname

The name of the database you want to work with. As a SQLite database corresponds to one file in your filesystem, `dbname` can be any legal filename. If the database/file doesn't exist when you first try to access it, SQLite will create it on the fly.

It is important to understand that the full path of the database is composed of `sqlite_dbdir` and `dbname`. Therefore `dbname` should not contain the full path of a file, but just the name.

sqlite_dbdir

This is the directory that contains all sqlite databases. Use the full path please.

> **Note:** It is necessary to keep all sqlite databases in one directory to make it possible to list all existing databases through the libdbi API. However, you are free to open more than one connection simultaneously, each one using a different setting of *`sqlite_dbdir`*.

sqlite_timeout

The design of SQLite does not allow concurrent access by two clients. However, if the timeout is larger than zero, the second client will wait for the given amount of time for the first client to release its lock. If the timeout is set to zero, the second client will return immediately, indicating a busy status. The numerical value of this option specifies the timeout in milliseconds.

Chapter 4. Peculiarities you should know about

This chapter lists known peculiarities of the sqlite driver. This includes SQLite features that differ from what you know from the other database servers supported by libdbi, and it includes features and misfeatures introduced by the sqlite driver. It is the intention of the driver author to reduce the number of misfeatures in future releases if possible.

4.1. SQLite (mis)features

As the SQLite package is constantly being improved, you should refer to the original documentation about the SQL features it supports (http://www.sqlite.org/lang.html) and about the SQL features it doesn't support (http://www.sqlite.org/omitted.html).

One noticeable difference between SQLite and other SQL database engines is that the former is typeless. All data are stored as strings, and you can insert any type of data into any column. While the SQLite author has good reasons for this feature, it is an obstacle to using the strongly typed retrieval functions of libdbi. The only way out is to declare the column types in a **CREATE TABLE** statement just as you would with any other SQL database engine. As an example, the following statement is perfectly fine with SQLite, but not with the sqlite driver:

```
CREATE TABLE foo (a,b,c)
```

However, the following statement is fine with SQLite, the sqlite driver, and just about any other SQL database server:

```
CREATE TABLE foo (a INTEGER,b TEXT,c VARCHAR(64))
```

The following table lists the column types which are positively recognized by the sqlite driver. Essentially all column types supported by MySQL and PostgreSQL are supported by this driver as well, making it reasonably easy to write portable SQL code. All other column types are treated as strings.

Table 4-1. SQL column types supported by the sqlite driver

type	description
TINYBLOB, BLOB, MEDIUMBLOB, LONGBLOB, BYTEA	String types of unlimited length. Binary data must be safely encoded, see text.
CHAR(), VARCHAR(), TINYTEXT, TEXT, MEDIUMTEXT, LONGTEXT	String types of unlimited length. There is no chopping or padding performed by the database engine.
ENUM	String type of unlimited length. In contrast to MySQL, choosing ENUM over VARCHAR does not save any storage space.
SET	String type of unlimited length. In contrast to MySQL, the input is not checked against the list of allowed values.
YEAR	String type of unlimited length. MySQL stores 2 or 4 digit years as a 1 byte value, whereas the SQLite drivers stores the string as provided.

type	description
TINYINT, INT1, CHAR	A 1 byte type used to store one character, a signed integer between -128 and 127, or an unsigned integer between 0 and 255.
SMALLINT, INT2	2 byte (short) integer type used to store a signed integer between -32768 and 32767 or an unsigned integer between 0 and 65535.
MEDIUMINT	3 byte integer type used to store a signed integer between -8388608 and 8388607 or an unsigned integer between 0 and 16777215.
INT, INTEGER, INT4	4 byte (long) integer type used to store a signed integer between -2147483648 and 2147483647 or an unsigned integer between 0 and 4294967295.
BIGINT, INT8	8 byte (long long) integer type used to store a signed integer between -9223372036854775808 and 9223372036854775807 or an unsigned integer between 0 and 18446744073709551615.
DECIMAL, NUMERIC	A string type of unlimited length used to store floating-point numbers of arbitrary precision.
TIMESTAMP, DATETIME	A string type of unlimited length used to store date/time combinations. The required format is 'YYYY-MM-DD HH:MM:SS', anything following this pattern is ignored.
DATE	A string type of unlimited length used to store a date. The required format is 'YYYY-MM-DD', anything following this pattern is ignored.
TIME	A string type of unlimited length used to store a time. The required format is 'HH:MM:SS', anything following this pattern is ignored.
FLOAT, FLOAT4, REAL	A 4 byte floating-point number. The range is -3.402823466E+38 to -1.175494351E-38, 0, and 1.175494351E-38 to 3.402823466E+38. Please note that MySQL treats REAL as an 8 byte instead of a 4 byte float like PostgreSQL.
DOUBLE, DOUBLE PRECISION, FLOAT8	An 8 byte floating-point number. The range is -1.7976931348623157E+308 to -2.2250738585072014E-308, 0, and 2.2250738585072014E-308 to 1.7976931348623157E+308.

Another difference is the lack of access control on the database engine level. Most SQL database servers implement some mechanisms to restrict who is allowed to fiddle with the databases and who is not. As SQLite uses regular files to store its databases, all available access control is on the filesystem level. There is no SQL interface to this kind of access control, but **chmod** and **chown** are your friends.

SQLite appears to implement row and column counters as C long int values. This limits the maximum number of rows somewhat compared to other SQL database engines.

SQLite does not have specific support for binary data. If you want to store binary data (e.g. character sequences containing NULL bytes) in a fashion that is portable across all database servers supported by libdbi, use the libdbi function `_dbd_encode_binary`.

SQLite currently supports different character encodings only as compile-time options. There is no way to change the encoding at runtime or per database.

4.2. sqlite driver misfeatures

And now we have to discuss how successful the sqlite driver is in squeezing the SQLite idea of a database engine into the libdbi framework which was shaped after MySQL and PostgreSQL. Keep in mind that the limitations mentioned here are not intrinsic, that is a sufficient amount of coding might fix these problems eventually.

- The sqlite driver has not been tested for memory leaks yet. It may leak memory like hell.
- The typeless nature of SQLite has some nasty consequences. The sqlite driver takes great care to reconstruct the type of a field that you request in a query, but this isn't always successful. Some of the functions that SQLite supports work both on numeric and text data. The sqlite driver currently cannot deduce the field type correctly as it would have to check all arguments of each function. Instead the sqlite driver makes a few assumptions that may be right or wrong in a given case. The affected functions are `coalesce(X,Y,...)`, `max(X)`, `min(X)`, and `count(X)`.
- The sqlite driver currently assumes that the directory separator of your filesystem is a slash (/). This may be wrong on your particular system. It is not a problem for Windows systems as long as the sqlite driver is built with the Cygwin tools (see `README.win32`).

  Note: Building the sqlite driver on Windows/Cygwin has not been tested yet, but it uses the same procedure as the other libdbi drivers. Chances are that it works.

- Listing tables with the `dbi_conn_get_table_list()` libdbi function currently returns only permanent tables. Temporary tables are ignored.
- The sqlite driver assumes that table and field names do not exceed 128 characters in length, including the trailing \0. I don't know whether SQLite internally has such a limit or not (both MySQL and PostgreSQL have a lower limit). The limit can be increased by changing a single #define in the `dbd_sqlite.h` header file.
- In a few cases, the sqlite driver expects you to type SQL keywords in all lowercase or all uppercase, but not mixed. This holds true for the 'from' in a SELECT statement. Type it either as 'from' or as 'FROM', but refrain from using 'fRoM' or other funny mixtures of uppercase and lowercase. Most other database engines treat the keywords as case-insensitive and would accept all variants.