sqlite3 egg

Bindings to version 3.x of the SQLite API.
Extension for Chicken Scheme
Version 1.5.4

Thomas Chust

Table of Contents

1 About this egg

1.1 Version history

`1.5.4` Typo fixed thanks to the new imports checking code

`1.5.3` Proper multithreading locks for the callback code

`1.5.2` Code cleanups

`1.5.1` Potential memory leaks removed

`1.5.0` Support for user defined collation sequences and functions has been added

`1.4.0` Stale statement handles due to schema changes are now automagically recompiled to keep them valid

`1.3.1` Several small routines added

`1.3.0` Special hacks removed as CVS version of SQLite3.3.4 has been fixed

`1.2.0` Special hacks to deal with "valid" NULL statements

`1.1.4` Tightened security measures against `#f` pointers

`1.1.3` Integers not fitting in a fixnum are now read correctly from the database [thanks to Zbigniew]

`1.1.2` Setup file patched to call compiled output .dll on Windows

`1.1.1` All procedures now reset prepared statements where you would do that by hand anyway

`1.1.0` Promoted `sqlite3:call-with-temporary-statement` to `sqlite3:call-with-temporary-statements` and fixed a really stupid coding mistake in `sqlite3:changes`

`1.0.3` Fixed C compiler warnings

`1.0.2` Added a typecheck for increased safety

`1.0.1` Fixed type mistakes in the source

`1.0.0` Initial release

1.2 Usage

Load this egg like so:

```
(require-extension sqlite3)
```

1.3 Requirements

This egg requires the following extensions:

`synch`

2 Documentation

The API of SQLite changed significantly from version 2.x to 3.x. These are new bindings to the modified API, which are reasonably complete – most procedures that take callback arguments are missing, though.

For in-depth information on the functionality of the routines and general information you should consult the SQLite documentation as well as this manual.

Unless otherwise indicated, all procedures and methods in this egg may throw an exception of the kind `(exn sqlite3)` if something goes wrong. This exception will contain a `status` property indicating the return value of the operation that failed:

Symbol	**Meaning**
`error`	SQL error or missing database
`internal`	An internal logic error in SQLite
`permission`	Access permission denied
`abort`	Callback routine requested an abort
`busy`	The database file is locked
`locked`	A table in the database is locked
`no-memory`	A malloc() failed
`read-only`	Attempt to write a readonly database
`interrupt`	Operation terminated by sqlite-interrupt()
`io-error`	Some kind of disk I/O error occurred
`corrupt`	The database disk image is malformed
`not-found`	(Internal Only) Table or record not found
`full`	Insertion failed because database is full
`cant-open`	Unable to open the database file
`protocol`	Database lock protocol error
`empty`	(Internal Only) Database table is empty
`schema`	The database schema changed

`too-big`	Too much data for one row of a table
`constraint`	Abort due to contraint violation
`mismatch`	Data type mismatch
`misuse`	Library used incorrectly
`no-lfs`	Uses OS features not supported on host
`authorization`	Authorization denied
`done`	`sqlite3:step!` has finished executing, so no further data is ready

2.1 Classes

`<sqlite3:database>` [class]
`<sqlite3:statement>` [class]

These classes are derived from `<c++-object>`. They hold a pointer to the underlying C-structure in their `this` slot.

`<sqlite3:statement>` also has a `database` slot pointing to the database object it belongs to.

2.2 Managing databases

`sqlite3:open` [procedure]

```
(sqlite3:open (path <string>)) => <sqlite3:database>
```

Opens the indicated database file and returns a `<sqlite3:database>` object for it.

The given path is subject to the same special expansion as paths passed to open-input-file and similar procedures.

`sqlite3:define-collation` [method]

```
(sqlite3:define-collation (db <sqlite3:database>) (name <string>)) => <void>
(sqlite3:define-collation (db <sqlite3:database>) (name <string>) (proc <procedur
```

If `proc` is given, registers a new collation sequence identified by `name` for use in the context of database handle `db`. If no procedure is passed, the collation sequence with the given name is removed.

`proc` should have the signature `(proc (a <string>) (b <string>)) => <exact>`. It should return a negative number if `a` sorts before `b`, a positive number if `b` sorts before `a` and zero if `a` and `b` are equal.

As `proc` will be called in a callback context from within `sqlite3:step!`, safety measures are installed to avoid throwing any exceptions, invoking continuations or returning invalid values from it. Attempts to do so will result in a `0` return value and warning messages.

`sqlite3:define-function` [method]

```
(sqlite3:define-function (db <sqlite3:database>) (name <string>) (n <exact>) (pro
(sqlite3:define-function (db <sqlite3:database>) (name <string>) (n <exact>) (ste
```

If `proc` is given, registers a new SQL function identified by `name` for use in the context of database handle `db`. If `step-proc` and `final-proc` are given, the new function becomes an aggregate function. Once registered, functions cannot be deleted.

`n` is the number of parameters the new SQL function takes or `-1` to allow any number of arguments.

`proc` should have the signature `(proc . params) => <top>`. It is called with the `n` parameters given to the SQL function converted into Scheme objects like by `sqlite3:column-data`. The return value is converted into an SQLite3 data object like by `sqlite3:bind!`. A return value of `#f` corresponds to `NULL` in SQLite3.

`step-proc` should have the signature `(step-proc (seed <top>) . params) => <top>`. It is called with the parameters given to the SQL function for every row being processed. The seed value passed is initially the one given as an argument to `sqlite3:define-function`; for subsequent calls it is the last value returned by `step-proc` and after completion of `final-proc` it will be the initial value again.

`final-proc` should have the signature `(final-proc (seed <top>)) => <top>` and transforms the last seed value into the value to be returned from the aggregate function.

As `proc`, `step-proc` and `final-proc` will be called in a callback context from within `sqlite3:step!`, safety measures are installed to avoid throwing any exceptions, invoking continuations or returning invalid values from them. Attempts to do such things will result in `NULL` return values and warning messages.

`sqlite3:set-busy-timeout!` [procedure]

```
(sqlite3:set-busy-timeout! (db <sqlite3:database>) #!optional ((ms <exact>) 0)) =
```

Installs a busy handler that waits at least the specified amount of milliseconds for locks on the given database. If `(<= ms 0)` though, all busy handlers for the database are uninstalled.

`sqlite3:interrupt!` [procedure]

```
(sqlite3:interrupt! (db <sqlite3:database>)) => <void>
```

Cancels any running database operation as soon as possible.

This function is always successful and never throws an exception.

`sqlite3:auto-committing?` [procedure]

```
(sqlite3:auto-committing? (db <sqlite3:database>)) => <bool>
```

Checks whether the database is currently in auto committing mode, i.e. no transaction is currently active.

This function always returns a state and never throws an exception.

`sqlite3:changes` [procedure]

```
(sqlite3:changes (db <sqlite3:database>) #!optional ((total <bool>) #f)) => <numb
```

Returns the number of rows changed by the last statement (if `(not total)`) or since the database was opened (if `total`).

This function always returns a count and never throws an exception.

`sqlite3:last-insert-rowid` [procedure]

```
(sqlite3:last-insert-rowid (db <sqlite3:database>)) => <number>
```

Returns the row ID of the last row inserted in `db`.

This function always returns a number and never throws an exception.

`sqlite3:finalize!` [method]

```
(sqlite3:finalize! (db <sqlite3:database>)) => <void>
```

Closes the given database.

2.3 Managing statements

`sqlite3:prepare` [procedure]

```
(sqlite3:prepare (db <sqlite3:database>) (sql <string>)) => <sqlite3:statement>,
```

Compiles the first SQL statement in `sql` and returns a `<sqlite3:statement>` and the rest of `sql`, which was not compiled (or an empty string).

`sqlite3:repair!` [procedure]

```
(sqlite3:repair! (stmt <sqlite3:statement>)) => <void>
```

Recompiles the SQL statement used to create `stmt`, transfers all existing bindings from the old statement handle to the new one and destructively modifies `stmt` to point to the new statement handle.

If the operation is successful, the old handle is finalized, in case of error, the new handle is finalized and the old one stays untouched.

Usually you should not have to call this routine by hand. It is invoked by `sqlite3:step!` to automagically repair a stale statement handle after a database schema change.

`sqlite3:column-count` [procedure]

```
(sqlite3:column-count (stmt <sqlite3:statement>)) => <exact>
```

Can be applied to any statement and returns the number of columns it will return as results.

This procedure always succeeds and never throws an exception.

`sqlite3:column-name` [procedure]

```
(sqlite3:column-name (stmt <sqlite3:statement>) (i <exact>)) => <string>
```

Can be applied to any statement and returns the name of the column number `i` (counting from 0) as a string or `#f` if the column has no name.

This procedure always succeeds and never throws an exception.

`sqlite3:column-declared-type` [procedure]

```
(sqlite3:column-declared-type (stmt <sqlite3:statement>) (i <exact>)) => <string>
```

Can be applied to any statement and returns the declared type (as given in the `CREATE` statement) of the column number `i` (counting from 0) as a string or `#f` if the column has no declared type.

This procedure always succeeds and never throws an exception.

`sqlite3:bind-parameter-count` [procedure]

```
(sqlite3:bind-parameter-count (stmt <sqlite3:statement>)) => <exact>
```

Can be applied to any statement and returns the number of free parameters that can be bound in the statement.

This procedure always succeeds and never throws an exception.

`sqlite3:bind-parameter-index` [procedure]

```
(sqlite3:bind-parameter-index (stmt <sqlite3:statement>) (name <string>)) => <exa
```

Can be applied to any statement and returns the index of the bindable parameter called `name` or `#f` if no such parameter exists.

This procedure always succeeds and never throws an exception.

`sqlite3:bind-parameter-name` [procedure]

```
(sqlite3:bind-parameter-name (stmt <sqlite3:statement>) (i <exact>)) => <string>
```

Can be applied to any statement and returns the name of the bindable parameter number `i` (counting from 0) or `#f` if no such parameter exists or the parameter has no name.

This procedure always succeeds and never throws an exception.

`sqlite3:bind!` [method]

```
(sqlite3:bind! (stmt <sqlite3:statement>) (i <exact>)) => <void>
(sqlite3:bind! (stmt <sqlite3:statement>) (i <exact>) (v <exact>)) => <void>
(sqlite3:bind! (stmt <sqlite3:statement>) (i <exact>) (v <number>)) => <void>
(sqlite3:bind! (stmt <sqlite3:statement>) (i <exact>) (v <string>)) => <void>
(sqlite3:bind! (stmt <sqlite3:statement>) (i <exact>) (v <byte-vector>)) => <void
```

Can be applied to any statement to bind its free parameter number `i` (counting from 0) to the given value. Scheme types of the value map to SQLite types as follows:

Scheme type	**SQLite type**
none	`null`
`<exact>`	`integer`
`<number>`	`float`
`<string>`	`text`
`<byte-vector>`	`blob`

Unless there is internal trouble in SQLite3, this method should always succeeds and never throw an exception. For invalid parameter indices the method just silently does nothing.

`sqlite3:step!` [procedure]

```
(sqlite3:step! (stmt <sqlite3:statement>)) => <bool>
```

Single-steps the execution of `stmt` and returns `#t` if a result row was produced, `#f` if no further results are available as the statement has been stepped through. This procedure must be called at least once before any results can be retrieved from the statement.

`sqlite3:column-type` [procedure]

```
(sqlite3:column-type (stmt <sqlite3:statement>) (i <exact>)) => <symbol>
```

Can be applied to a statement that has just been stepped (otherwise it returns `#f`) and returns the SQLite type of the result column number `i` (counting from 0) as a symbol.

The return value can be one of the symbols `null`, `integer`, `float`, `text` or `blob`.

This procedure always succeeds and never throws an exception.

`sqlite3:column-data` [procedure]

```
(sqlite3:column-data (stmt <sqlite3:statement>) (i <exact>)) => <bool | exact | n
```

Can be applied to a statement that has just been stepped. Consults `sqlite3:column-type` to determine the type of the indicated column and to return its data as an appropriate scheme object.

See `sqlite3:bind!` for the mapping between Scheme and SQLite data types. Columns of type `null` are returned as `#f`. Also keep in mind that CHICKEN's `<exact>` datatype can only hold a subset of the values an SQLite `integer` can store. Large integer values may therefore be returned as floating point numbers from the database, but they will still be of class `<integer>`.

This procedure always succeeds and never throws an exception.

`sqlite3:reset!` [procedure]

```
(sqlite3:reset! (stmt <sqlite3:statement>)) => <void>
```

Can be applied to any statement and resets it such that execution using `sqlite3:step!` will perform all operations of the statement again.

`sqlite3:finalize!` [method]

```
(sqlite3:finalize! (stmt <sqlite3:statement>)) => <void>
```

Must be applied to every statement to free its resources and discard it.

`sqlite3:close` will not be able to close a database that has associated unfinalized statements.

2.4 Simple statement interface

`sqlite3:call-with-temporary-statements` [procedure]

```
(sqlite3:call-with-temporary-statements (proc <procedure-class>) (db <sqlite3:dat
```

Compiles the SQL sources in `sqls` into statements in the context of `db`, applies `proc` to these statements and returns `proc`'s result. The statements are created and finalized in `dynamic-wind` entry and exit blocks around the application of `proc`.

`sqlite3:exec` [method]

```
(sqlite3:exec (stmt <sqlite3:statement>) . params) => <void>
(sqlite3:exec (db <sqlite3:database>) (sql <string>) . params) => <void>
```

(Compiles the given SQL), resets the statement, binds the statement's free parameters and executes the statement ignoring possible results from it.

`sqlite3:update` [method]

```
(sqlite3:update (stmt <sqlite3:statement>) . params) => <exact>
(sqlite3:update (db <sqlite3:database>) (sql <string>) . params) => <exact>
```

(Compiles the given SQL), resets the statement, binds the statement's free parameters and executes the specified statement ignoring possible results from it, returning the result of applying `sqlite3:changes` to the affected database after the execution of the statement instead.

`sqlite3:first-result` [method]

```
(sqlite3:first-result (stmt <sqlite3:statement>) . params) => <bool | exact | num
(sqlite3:first-result (db <sqlite3:database>) (sql <string>) . params) => <bool |
```

(Compiles the given SQL), resets the statement, binds the statement's free parameters and single-steps the statement once returning the value of the first column in the first result row. Resets the statement again just before returning.

If the given statement does not yield any results, an `(exn sqlite3)` is thrown with the `status`-property set to `done`.

`sqlite3:first-row` [method]

```
(sqlite3:first-row (stmt <sqlite3:statement>) . params) => <list>
(sqlite3:first-row (db <sqlite3:database>) (sql <string>) . params) => <list>
```

(Compiles the given SQL), resets the statement, binds the statement's free parameters and single-steps the statement once returning all columns in the first result row as a list.

If the given statement does not yield any results, an `(exn sqlite3)` is thrown with the `status`-property set to `done`.

`sqlite3:for-each-row` [method]

```
(sqlite3:for-each-row (proc <procedure-class>) (stmt <sqlite3:statement>) . param
(sqlite3:for-each-row (proc <procedure-class>) (db <sqlite3:database>) (sql <stri
```

(Compiles the given SQL), resets the statement, binds the statement's free parameters and executes it step by step. After each step, the column values of the current result row are retrieved and `proc` is applied to them. The results of this application are discarded.

`sqlite3:map-row` [method]

```
(sqlite3:map-row (proc <procedure-class>) (stmt <sqlite3:statement>) . params) =>
(sqlite3:map-row (proc <procedure-class>) (db <sqlite3:database>) (sql <string>)
```

(Compiles the given SQL), resets the statement, binds the statement's free parameters and executes it step by step. After each step, the column values of the current result row are retrieved and `proc` is applied to them. The results of these applications are collected into a list.

2.5 Utility functions

`sqlite3:complete?` [procedure]

```
(sqlite3:complete? (sql <string>)) => <bool>
```

Checks whether `sql` comprises at least one complete SQL statement.

`sqlite3:library-version` [procedure]

```
(sqlite3:library-version) => <string>
```

Returns a string identifying the version of SQLite in use.

3 License

Index

<

S