

Overview of the sqlite3x and sq3 APIs

Abstract:

This document give an introduction on how to use the sqlite3x and sq3 C++ wrappers for sqlite3. Some knowledge of sqlite3 is assumed, but not much is needed.

Maintainer: stephan at wanderinghorse dot net

Table of Contents

1 Preliminaries	1
1.1 License	1
1.2 Credits	1
2 Introduction	1
2.1 Compiling/linking	2
3 Overview of the sqlite3x API	2
3.1 sqlite3_connection	2
3.2 sqlite3_command	2
3.3 sqlite3_cursor	3
3.4 sqlite3_transaction	3
3.5 settings_db	3
4 Overview of the sq3 API	4
4.1 database	4
4.2 statement	4
4.3 cursor	5
4.4 transaction	5
4.5 settings_db	6
4.6 log_db	6

1 Preliminaries

1.1 License

This document is released into the Public Domain. The sqlite3x API inherits a zlib-like license from its originating code, as explained in sqlite3x.hpp. The sq3 API is released into the Public Domain.

1.2 Credits

- D. Richard Hipp, author of sqlite (<http://sqlite.org>).
- Cory Nelson wrote the original sqlite3x code (<http://dev.int64.org/sqlite.html>)
- Me (stephan beal), i wrote the sq3 code, hacked upon and documented the sqlite3x code, and wrote this document.

2 Introduction

This document provides an overview of two similar C++ wrappers for the sqlite3 C API with distinctly different design goals. The first is known as **sqlite3x**. This code is an inherited/expanded/documented set of classes originally written by Cory Nelson. The older documentation suggested that it resembles an ADO interface, but i have no experience with ADO and can't comment on that. The sqlite3x API makes use of exceptions to report even the slightest errors. The second API, **sq3**, is functionally and structurally similar to sqlite3x, but was designed for a platform where the use of exceptions is not desired or not allowed. (Originally it was intended for use on a WinCE/PocketPC platform.) The sq3 API uses sqlite3's error codes (integers) to report errors.

Both libraries have a similar structure, and it goes a bit like this:

- A class encapsulating a database connection. This is either `sqlite3x::sqlite3_connection` or

sqlite3::database.

- A class encapsulating a prepared statement: sqlite3x::sqlite3_command or sqlite3::statement.
- A class encapsulating a cursor or query result record: sqlite3x::sqlite3_cursor or sqlite3::cursor.
- A class to simplify the use of transactions: sqlite3x::sqlite3_transaction or sqlite3::transaction.

A note about the longer class names in sqlite3x: their names were inherited from their original code (by another author).

This code can be downloaded from its home page:

<http://wanderinghorse.net/computing/sqlite/>

2.1 Compiling/linking

The code described here is pretty small, and can be copied into arbitrary projects for directly inclusion into the project tree. In that case, the only significant thing to know is that you'll need to have <sqlite3.h> in your includes path and you'll need to link to the sqlite3 static or shared library (e.g., libsqlite3.a or sqlite3.dll).

If you choose to not directly include it in your project then you will need to link against the appropriate library (libsqlite3x.a or libsqlite3.a) and have their headers in your include path. The headers are intended to live under <s11n.net/sqlite3x/...> and <s11n.net/sqlite3/...>, and thus you need to add the parent of <s11n.net> to your includes path.

This code was really intended to be included directly into arbitrary project trees, so please don't hesitate to do so.

3 Overview of the sqlite3x API

Cory Nelson's sqlite3x was, after quite a lot of searching, the best OO wrapper for sqlite3 that i had come across. Unfortunately, the code was almost completely undocumented. After failing to hear from Cory regarding a few questions/suggestions for the code, i took it over. The main difference is that this one is documented (all API docs are in sqlite3x.hpp), but this fork also has several more minor additions and changes.

This API relies 100% on exceptions for error reporting. For any type of error it throws a sqlite3x::database_error, which subclasses std::exception.

The code snippets shown below show no error handling because on any error an exception is thrown.

3.1 sqlite3_connection

This class encapsulates a database connection. It is created by passing a database file name to the constructor or by default-constructing and then calling open(filename). This API provides only the most basic operations of an sqlite3 database. If you need to do any low-level work, use the db() member to get a handle to the underlying sqlite3 database.

Example:

```
sqlite3_connection db( "my.db" );
sqlite3_connection db2();
db2.open( "your.db" );
```

You can perform simple queries through the database via the various executeXXX() functions, but the API does not provide a simple way of getting sets of results from the database. For that you need the remaining classes...

3.2 sqlite3_command

This class represents a prepared statement. Each command must be atomic. That is, once a command is opened, it must be closed before another one can successfully run. Closing a statement object implicitly closes the underlying statement. Here is a simple example which does not require a result set:

```
int getFooCount( sqlite3_connection & con ) {
    sqlite3_command st( con, "select count(*) from foo" );
```

```
    return st.executeint();
}
```

The class can do more interesting things, like binding values to placeholder arguments in SQL queries:

```
sqlite3_command st( mydb, "insert into foo (id, a, b, c) values(NULL, ?, ?, ?)" );
st.bind( 1, "This is field A" );
st.bind( 2, 42 );
st.bind( 3, 42.42 );
st.executenonquery(); // runs the INSERT statement.
```

Note that the first argument to `bind()` is an index in the prepared statement is 1-based, not 0-based. On the contrary, `sqlite3_cursor::get()` also takes an index, but it is 0-based. This inconsistency is unfortunately inherited from the core `sqlite3` API.

3.3 `sqlite3_cursor`

This class is used for stepping through result sets. It is never created directly, but through `sqlite3_command::executecursor()`. Here is an example:

```
sqlite3_connection db(":memory:");
db.executenonquery("create table foo(a,b,c)");
db.executenonquery("create table bar(a,b,c)");
sqlite3_command st(db, "select * from sqlite_master where type='table'");
sqlite3_cursor cur( st.executecursor() );
int row = 0;
const int colcount = st.colcount();
while( cur.step() ) {

    if( 0 == row++ ) { // show column names
        for( int i = 0; i < colcount; ++i ) {
            std::cout << cur.getcolname(i);
            if( i < (colcount-1) ) std::cout << '\t';
        }
        std::cout << '\n';
    }
    // Print the row info in tab-delimited format...
    char const * val = 0;
    int strsize = 0;
    for( int i = 0; i < colcount; ++i ) {
        val = cur.getstring( i, strsize );
        std::cout << (val ? val : "NULL");
        if( i < (colcount-1) ) std::cout << '\t';
    }
    std::cout << '\n';
}
std::cout << row << " row(s) processed.\n";
```

3.4 `sqlite3_transaction`

One of the easiest ways to get more speed out of `sqlite3` is to make careful use of transactions when doing

multiple inserts/updates. Using it is trivial:

```
sqlite3_transaction tr( mydb );
// ... do updates/inserts in mydb ...
tr.commit();
```

You can optionally pass `false` as the second constructor parameter and call `begin()` to start the transaction. If you do not call `commit()` before the transaction goes out of scope, it is the same as calling `rollback()`, which undoes all changes made since the transaction was opened.

Unfortunately, `sqlite3` does not support nested transactions, so you may not use multiple instances of `sqlite3_transaction` at once.

3.5 settings_db

This `sqlite3_connection` subclass provides a simple get/set API intended to be used as a basic configuration file interface. It is not particularly fast because each set operation requires its own insert/update code and each get operation requires its own custom query. It can be made to run faster if multiple set operations are wrapped in a transaction.

```
settings_db db("settings.db");

// Set some values...
sqlite3_transaction tr(*db.db());
db.set("f1", 42 );
db.set("f2", 42.24 );
db.set("f1", 43 );
db.set("f3", "fourty two" );
db.set("f3.5", std::string("fourty two" ) );
db.set("f4", true );
tr.commit();

// Read them back...
bool got = true;
int i1;
double d1;
std::string s1;
bool b1;
got = db.get( "f1", i1 ); // got == true, i1 == 42
got = db.get( "f2", d1 ); // got == true, d1 == 42.24
got = db.get( "f3", s1 ); // got == true, s1 == "fourty two"
got = db.get( "f3.5", s1 ); // got == true, s1 == "fourty two"
got = db.get( "f4", b1 ); // got == true, b1 == true
got = db.get( "f27", i1 ); // got == false, i1 not modified
```

Note that `get()` does not throw if it fails to find an entry. It simply returns `false` and does not modify the reference passed as the second argument.

4 Overview of the sq3 API

The `sq3` API is structurally almost identical to `sqlite3x`, but does not use exceptions to report errors. Instead it uses the `sqlite3` result codes. For example, `SQLITE_OK` is normally the success code and `SQLITE_ERROR` is the generic error code.

4.1 database

This class represents a single connection to a database. It has essentially the same features as `sqlite3x::sqlite3_connection`. Example:

```
database mydb( "my.db" );
if( ! mydb.is_open() ) { ... error ... }
// mydb is open and ready...
int tablecount = 0;
char const * sql = "select count from sqlite_master where type='table'";
if( ! rc_is_okay( mydb.execute(sql, tablecount) ) ) { ... error ... }
if( 0 == tablecount ) { ... mydb has no tables. Let's set some up... }
```

4.2 statement

This class represents a single prepared SQL statement. See the notes for `sqlite3x::sqlite3_command`, as those apply here. Example usage:

```
statement st( mydb,
    "select count from sqlite_master where type='table'" );
if( ! st.is_prepared() )
    { ... bad sql ... }
int tablecount = -1;
if( ! rc_is_okay( st.execute( tablecount ) ) )
    { ... error running query ... }
if( 0 == tablecount )
    { ... mydb has no tables. Let's set some up... }
```

We can also binding values to placeholder arguments in SQL queries:

```
statement st( mydb, "insert into foo (id,a,b,c) values(NULL,?,?,?)" );
st.bind( 1, "This is field A" );
st.bind( 2, 42 );
st.bind( 3, 42.42 );
st.execute(); // runs the INSERT statement.
```

Note that the first argument to `bind()` is an index in the prepared statement is 1-based, not 0-based. On the contrary, `sq3::get()` also takes an index, but it is 0-based. This inconsistency is unfortunately inherited from the core `sqlite3` API.

4.3 cursor

The cursor class is used to iterate through a set of query results. It is never instantiated directly, but is instead created through `statement::get_cursor()`.

It is important that no cursor be used after its underlying statement has gone out of scope. Ideally, no cursor will outlive a statement object.

Here is an example:

```
database db( ":memory:" );
db.execute("create table foo(a,b,c)");
db.execute("create table bar(a,b,c)");
statement st(db, "select * from sqlite_master where type='table'");
cursor cur( st.get_cursor() );
int row = 0;
```

```

const int colcount = st.colcount();
while( SQLITE_ROW == cur.step() ) {
    if( 0 == row++ ) { // show column names
        std::string colname;
        for( int i = 0; i < colcount; ++i ) {
            cur.colname( i, &colname );
            std::cout << colname;
            if( i < (colcount-1) ) std::cout <<'\t';
        }
        std::cout << '\n';
    }
    // Print the row info in tab-delimited format...
    int strsize = 0;
    sqlite3_text_char_t1 const * data = 0;
    for( int i = 0; i < colcount; ++i ) {
        data = 0;
        cur.get( i, & data, strsize );
        if( ! strsize ) std::cout << "NULL";
        else std::cout << data;
        if( i < (colcount-1) ) std::cout <<'\t';
    }
    std::cout << '\n';
}
std::cout << row << " row(s) processed.\n";

```

The major difference between this and the `sqlite3x` example code is the way that result values are fetched. In the `sqlite3x` API we do not pass a reference to a result type because that API throws on errors, allowing us to directly return the result type.

4.4 transaction

One of the easiest ways to get more speed out of `sqlite3` is to make careful use of transactions when doing multiple inserts/updates. Using it is trivial:

```

transaction tr( mydb );
// ... do updates/inserts in mydb ...
tr.commit();

```

You can optionally pass `false` as the second constructor parameter and call `begin()` to start the transaction. If you do not call `commit()` before the transaction goes out of scope, it is the same as calling `rollback()`, which undoes all changes made since the transaction was opened.

Unfortunately, `sqlite3` does not support nested transactions, so you may not use multiple instances of `sqlite3_transaction` at once.

4.5 settings_db

This database subclass provides a simple `get/set` API intended to be used as a basic configuration file interface. It is not particularly fast because each `set` operation requires its own `insert/update` code and each `get` operation requires its own custom query. It can be made to run faster if multiple `set` operations are wrapped in a transaction.

Example code:

¹ This unfortunate typedef comes from the `sq3` namespace and is required in some places to avoid an overload ambiguity with an overload of `get()` which takes a `(void const **)`. In the `sqlite3` API, some of the string-related functions use `(unsigned char *)` instead of `(char *)`. Not sure why.

```

settings_db sset( "settings.db" );
if( ! sset.is_open() )
{
    ... error ...
}
sset.set( "foo", "bar" );
sset.set( "bar", "42" );
s1 = "error";
i1 = -1;
sset.get( "foo", s1 ); // s1 == "bar"
sset.get( "bar", i1 ); // i1 == 42

```

4.6 log_db

This database subclass provides a very simple way to log events within an application. It has functions for logging messages and a virtual interface for presenting log data to the user (the default implementation uses `std::cout`, but subclasses could show a dialog box or similar).

An example of using it:

```

// This function provides the logger for our whole application:
sq3::log_db & MyLog() {
    static sq3::log_db bob;
    if( ! bob.is_open() ) {
        bob.open( "/path/to/log.db" );
    }
    return bob;
}

... later on ...
MyLog().log( "Pay attention: %d, %s", anInteger, aCString );
MyLog().log( "This is entry #2." );
MyLog().trim( 5 ); // removes all but newest 5 entries
MyLog().show_last( 5 ); // "shows" last 5 entries.
MyLog().clear(); // empties the db

```