S3M Documentation

Release 1.1.0

Ivan Konovalov

Jun 12, 2018

Contents

CHAPTER 1

Introduction

One of the problems of the built-in *sqlite3* module is that doesn't work very well with multithreading. S3M - is a wrapper of *sqlite3* that allows you to easily do multithreading:

- It locks parallel database operations so that only one can run at a time.
- It can also lock transactions (enabled by default) so that only one transaction can be active at a time.
- You won't get an *OperationalError* saying that the database is locked. All the database operations will just run in a queue.

Keep in mind that this library can only help you with **threads**, not **processes**.

1.1 What else is different from *sqlite3*?

- You can freely share connections between threads (not that you have to), given `check_same_thread=False`.
- You can use the `with` statement with the connection object to acquire the locks.

1.2 Example

```
import random
import threading

# Try replacing 's3m' with 'sqlite3' and see what happens
import s3m

# Open the database file,
# isolation_level=None is needed to prevent sqlite3 from starting transactions on its␣
↪own
conn = s3m.connect("s3m_example.db", isolation_level=None)
```

(continues on next page)

(continued from previous page)

```
# Create table if it doesn't already exist
conn.execute("CREATE TABLE IF NOT EXISTS numbers(number INTEGER)")

def thread_func():
    conn = s3m.connect("s3m_example.db", isolation_level=None)
    conn.execute("BEGIN IMMEDIATE")
    conn.execute("INSERT INTO numbers VALUES(?)", (random.randint(1, 100),))

    # Imagine there's some intense database work going on
    time.sleep(1)

    conn.commit()

# Make 10 threads
threads = [threading.Thread(target=thread_func) for i in range(10)]

# Start them
for thread in threads:
    thread.start()

for thread in threads:
    thread.join()

# Now let's look at what we got
result = conn.execute("SELECT * FROM numbers").fetchall()
print(result)
```

As you can see from this example, the usage is pretty much the same as with built-in *sqlite3*.

CHAPTER 2

Installation

Run

```
pip install s3m
```

or

```
python setup.py install
```

CHAPTER 3

Documentation

S3M - sqlite3 wrapper for multithreaded applications

s3m.**connect**(*path*, *lock_transactions=True*, *lock_timeout=-1*, *single_cursor_mode=False*, *factory=<class 's3m.Connection'>*, **args*, ***kwargs*)

Analogous to sqlite3.connect()

Parameters

- **path** – Path to the database
- **lock_transactions** – If *True*, parallel transactions will be blocked
- **lock_timeout** – Maximum amount of time the connection is allowed to wait for a lock. If the timeout i exceeded, *LockTimeoutError* will be thrown. -1 disables the timeout.
- **single_cursor_mode** – Use only one cursor (default: *True*)
- **factory** – Connection class (default: *Connection*)

class s3m.**Connection**(*path*, *lock_transactions=True*, *lock_timeout=-1*, *single_cursor_mode=False*, **args*, ***kwargs*)

The connection class. It won't let multiple database operations execute in parallel. It can also block parallel transactions (with lock_transactions=True).

with statement is also supported, it acquires the locks, thus blocking all the competing threads. This can be useful to ensure that database queries will complete in the specified order.

Parameters

- **path** – Path to the database
- **lock_transactions** – If True, parallel transactions will be blocked
- **lock_timeout** – Maximum amount of time the connection is allowed to wait for a lock. If the timeout is exceeded, LockTimeoutError will be thrown. -1 disables the timeout.
- **single_cursor_mode** – Use only one cursor (default: True)

acquire(*lock_transactions=None*)

Acquire the connection locks.

Parameters **lock_transactions** – *bool*, acquire the transaction lock (*self.lock_transactions* is the default value)

arraysize
Analogous to sqlite3.Cursor.arraysize

Works only in single cursor mode.

close()
Close the connection

commit()
Analogous to sqlite3.Connection.commit

create_aggregate(**args*, ***kwargs*)
Analogous to sqlite3.Connection.create_aggregate

create_collation(**args*, ***kwargs*)
Analogous to sqlite3.Connection.create_collation

create_function(**args*, ***kwargs*)
Analogous to sqlite3.Connection.create_function

cursor()
Analogous to sqlite3.Connection.cursor

description
Analogous to sqlite3.Cursor.description

Works only in single cursor mode.

enable_load_extension(**args*, ***kwargs*)
Analogous to sqlite3.Connection.enable_load_extension

execute(**args*, ***kwargs*)
Analogous to sqlite3.Cursor.execute

executemany(**args*, ***kwargs*)
Analogous to sqlite3.Cursor.executemany

executescript(**args*, ***kwargs*)
Analogous to sqlite3.Cursor.executescript

fetchall()
Analogous to sqlite3.Cursor.fetchall.

Works only in single cursor mode.

fetchmany(**args*, ***kwargs*)
Analogous to sqlite3.Cursor.fetchmany.

Works only in single cursor mode.

fetchone()
Analogous to sqlite3.Cursor.fetchone.

Works only in single cursor mode.

in_transaction
Analogous to sqlite3.Connection.in_transaction

interrupt()
Analogous to sqlite3.Connection.interrupt

isolation_level
Analogous to sqlite3.Connection.isolation_level

iterdump(**args*, ***kwargs*)
Analogous to sqlite3.Connection.iterdump

lastrowid
Analogous to sqlite3.Cursor.lastrowid.

Works only in single cursor mode.

load_extension(**args*, ***kwargs*)
Analogous to sqlite3.Connection.load_extension

release(*lock_transactions=None*)
Release the connection locks.

Parameters **lock_transactions** – *bool*, release the transaction lock (*self.lock_transactions* is the default value)

rollback()
Analogous to sqlite3.Connection.rollback

row_factory
Analogous to sqlite3.Connection.row_factory

rowcount
Analogous to sqlite3.Cursor.rowcount.

Works only in single cursor mode.

set_authorizer(**args*, ***kwargs*)
Analogous to sqlite3.Connection.set_authorizer

set_progress_handler(**args*, ***kwargs*)
Analogous to sqlite3.Connection.set_progress_handler

set_trace_callback(**args*, ***kwargs*)
Analogous to sqlite3.Connection.set_trace_callback

text_factory
Analogous to sqlite3.Connection.text_factory

class s3m.**Cursor**(*connection*)
The cursor class, analogous to sqlite3.Cursor.

arraysize
Analogous to sqlite3.Cursor.arraysize

close()
Close the cursor

connection
Connection used by the cursor

description
Analogous to sqlite3.Cursor.description

execute(**args*, ***kwargs*)
Analogous to sqlite3.Cursor.execute

Returns self

executemany(**args*, ***kwargs*)
Analogous to sqlite3.Cursor.executemany

Returns self

executescript(**args*, ***kwargs*)
Analogous to sqlite3.Cursor.executescript

Returns self

fetchall()
Analogous to sqlite3.Cursor.fetchall

fetchmany(**args*, ***kwargs*)
Analogous to sqlite3.Cursor.fetchmany

fetchone()
Analogous to sqlite3.Cursor.fetchone

lastrowid
Analogous to sqlite3.Cursor.lastrowid

rowcount
Analogous to sqlite3.Cursor.rowcount

exception s3m.**S3MError**
The base class of all the other exceptions in this module

exception s3m.**LockTimeoutError**(*conn*, *msg=None*)
Thrown when Lock.acquire() took too long

3.1 Using `with` statement

The *Connection* (as well as *Cursor*) object supports the `with` statement. It acquires the locks which will result either in the current thread waiting for other threads or making other threads wait until the current thread exits the `with` block.

```
conn = s3m.connect("database.db", ...)
...

with conn: # This blocks other threads
    conn.execute(<something>)
    conn.execute(<something else>)

# The other threads are no longer blocked
```

Python Module Index

s

Index