**Improve INSERT-per-second performance of SQLite in Python 3.6?**

**Environment**

Intel Core i7-7700 @ 3.6GHz, 16GB, Micron 1100 256GB SSD, Windows 10 x64

Python 3.6.5 Minconda

**GenerateData.py**

import time

start\_time = time.time()

with open('input.ssv', 'w') as out:

symbols = ['AUDUSD','EURUSD','GBPUSD','NZDUSD','USDCAD','USDCHF','USDJPY','USDCNY','USDHKD']

lines = []

for i in range(0,1\*1000\*1000):

q1, r1, q2, r2 = i//100000, i%100000, (i+1)//100000, (i+1)%100000

line = '{} {}.{:05d} {}.{:05d}'.format(symbols[i%len(symbols)], q1, r1, q2, r2)

lines.append(line)

out.write('\n'.join(lines))

print(time.time()-start\_time, i)

**input.ssv**

AUDUSD 0.00000 0.00001

EURUSD 0.00001 0.00002

GBPUSD 0.00002 0.00003

NZDUSD 0.00003 0.00004

USDCAD 0.00004 0.00005

...

USDCHF 9.99995 9.99996

USDJPY 9.99996 9.99997

USDCNY 9.99997 9.99998

USDHKD 9.99998 9.99999

AUDUSD 9.99999 10.00000

// total 1 million of lines, taken 1.38 second for Python code

Windows shows 23,999,999 bytes file size.

**Baseline Code InsertData.py**

import time

class Timer:

def \_\_enter\_\_(self):

self.start = time.time()

return self

def \_\_exit\_\_(self, \*args):

elapsed = time.time()-self.start

print('Imported in {:.2f} seconds or {:.0f} per second'.format(elapsed, 1\*1000\*1000/elapsed))

with Timer() as t:

with open('input.ssv', 'r') as infile:

infile.read()

**Basic I/O**

with open('input.ssv', 'r') as infile:

infile.**read()**

Imported in 0.13 seconds or 7.6 M per second

with open('input.ssv', 'r') as infile:

lines = infile.read().splitlines()

Imported in 0.21 seconds or 4.7 M per second

with open('input.ssv', 'r') as infile:

with open('output.ssv', 'w') as outfile:

**outfile.write(infile.read()) // insert here**

Imported in 0.26 seconds or 3.84 M per second

This gives us a sense about how fast the IO and string processing operations on my testing machine.

1. **Write File**

outfile.**write**(line)

Imported in 0.52 seconds or 1.93 M per second

1. **Split to floats to string**

tokens = line.split()

sym, bid, ask = tokens[0], **float**(tokens[1]), **float**(tokens[2])

outfile.**write**('{} {:.5f} {%.5f}\n'.format(sym, bid, ask)) **// real insert here**

Imported in 2.25 seconds or 445 K per second

1. **Insert Statement with autocommit**

conn = sqlite3.connect('example.db', isolation\_level=None)

c.**execute**("INSERT INTO stocks VALUES ('{}',{:.5f},{:.5f})".format(sym,bid,ask))

When isolation\_level = None (autocommit), program takes many hours to complete. Note the output database file size is 32,325,632 bytes, which is 32MB. It is bigger than the input file ssv file size of 23MB by 10MB.

1. **Insert Statement with BEGIN (DEFERRED)**

conn = sqlite3.connect('example.db', isolation\_level=’**DEFERRED’**) # default

c.execute("INSERT INTO stocks VALUES ('{}',{:.5f},{:.5f})".format(sym,bid,ask))

Imported in 7.50 seconds or 133,296 per second

This is the same as writing “BEGIN”, “BEGIN TRANSACTION” or “BEGIN DEFERRED TRANSACTION”, not “BEGIN IMMEDIATE” nor “BEGIN EXCLUSIVE”.

1. **Insert by Prepared Statement**

Using the transaction above gives a satisfactory results but it should be noted that using Python’s string operations is undesired because it is subjected to SQL injection. Moreover using string is slow compared to parameter substitution.

c.executemany("INSERT INTO stocks VALUES (?,?,?)", [(sym,bid,ask)])

Imported in 2.31 seconds or 432,124 per second

1. **Turn off Synchronous**

Power failure corrupts the database file when synchronous is not set to EXTRA nor FULL before data reaches the physical disk surface. When we can ensure the power and OS is healthy, we can turn synchronous to OFF so that it doe not synchronized after data handed to OS layer.

conn = sqlite3.connect('example.db', isolation\_level='DEFERRED')

c = conn.cursor()

c.execute('''PRAGMA synchronous = OFF''')

Imported in 2.25 seconds or 444,247 per second

1. **Turn off journal and so no rollback nor atomic commit**

In some applications the rollback function of a database is not required, for example a time series data insertion. When we can ensure the power and OS is healthy, we can turn journal\_mode to off so that rollback journal is disabled completely and it disables the atomic commit and rollback capabilities.

conn = sqlite3.connect('example.db', isolation\_level='DEFERRED')

c = conn.cursor()

c.execute('''PRAGMA synchronous = OFF''')

c.execute('''PRAGMA journal\_mode = OFF''')

Imported in 2.22 seconds or 450,653 per second

1. **Using in-memory database**

In some applications writing data back to disks is not required, such as applications providing queried data to web applications.

conn = sqlite3.connect(":memory:")

Imported in 2.17 seconds or 460,405 per second

1. **Faster Python code**

We should consider to save every bit of computation inside an intensive loop, such as avoiding assignment to variable and string operations.

1. **Avoid assignment to variable**

tokens = line.split()

c.executemany("INSERT INTO stocks VALUES (?,?,?)", [(tokens[0], float(tokens[1]), float(tokens[2]))])

Imported in 2.10 seconds or 475,964 per second

1. **Avoid string.split()**

When we can treat the space separated data as fixed width format, we can directly indicate the distance between each data to the head of data.

It means line.split()[1] becomes line[7:14]

c.executemany("INSERT INTO stocks VALUES (?,?,?)", [(line[0:6], float(line[7:14]), float(line[15:]))])

Imported in 1.94 seconds or 514661 per second

1. **Avoid float() to ?**

When we are using executemany() with ? placeholder, we don’t need to turn the string into float beforehand.

executemany("INSERT INTO stocks VALUES (?,?,?)", [(line[0:6], line[7:14], line[15:])])

Imported in 1.59 seconds or 630,520 per second

1. **The fastest full functioned and robust code so far**

import time

class Timer:

def \_\_enter\_\_(self):

self.start = time.time()

return self

def \_\_exit\_\_(self, \*args):

elapsed = time.time()-self.start

print('Imported in {:.2f} seconds or {:.0f} per second'.format(elapsed, 1\*1000\*1000/elapsed))

import sqlite3

conn = sqlite3.connect('example.db')

c = conn.cursor()

c.execute('''DROP TABLE IF EXISTS stocks''')

c.execute('''CREATE TABLE IF NOT EXISTS stocks

(sym text, bid real, ask real)''')

c.execute('''PRAGMA synchronous = EXTRA''')

c.execute('''PRAGMA journal\_mode = DELETE''')

with Timer() as t:

with open('input.ssv', 'r') as infile:

lines = infile.read().splitlines()

for line in lines:

c.executemany("INSERT INTO stocks VALUES (?,?,?)", [(line[0:6], line[7:14], line[15:])])

conn.commit()

conn.close()

Imported in 1.77 seconds or 564,611 per second

**Summary and Quesstion**

I have a 23MB file with 1 million records composing of a piece of text as symbol name and 2 floating point number as bid and ask. Reading them then writing them back to disk takes 0.26 seconds or 3.84 M per second while reading it then inserting them to SQLite takes 1.77 seconds or 0.564 M per second. Do I miss something to make it faster?