A Brief Intro to Profiling in Python

Chasing the Fastest Possible Bulk Insert with SQLAlchemy

Russell Warren (russ@perspexis.com)

What is profiling?

Detailed accounting of what resources your code is using, and how.

This includes CPU time, memory, I/O, etc

Profiling is *CRITICAL* for optimization.

<u>We will be looking at profiling of CPU time only</u>

time.time()

```
import time

def fib(n):
    return n if n < 2 else fib(n - 1) + fib(n - 2)

start = time.time()
fib(30)
print "That took %.5fs" % (time.time() - start)
```

- Crude, but effective
- Won't lie to you (much)
  - *"not all systems provide time with a better precision than 1 second" -- python docs*
  - so be careful (but... 5 us resolution on this laptop)

timeit module

```
$ python -m timeit "[dict(a = 1) for i in xrange(1000)]"
1000 loops, best of 3: 304 usec per loop
$ python -m timeit "[{'a': 1} for i in xrange(1000)]"
10000 loops, best of 3: 183 usec per loop
```

- Great for quick "which is faster?" checks
- Is smart about timing...
  - Gives a best of 3
  - Removes overhead
  - Dynamic loop count to keep < 5s (or so)
  - Disables gc
- for help: `python -m timeit --help`
  - setup strings with `--setup`, and more

timeit is also importable

```
>>> import timeit
>>> setup = """import numpy
... a = numpy.arange(1e6)"""
>>> n = 20 #loop count
>>> for f in ("max", "numpy.max"):
...     tmr = timeit.Timer("%s(a)" % f, setup)
...     t_ms = tmr.timeit(n) * 1000 / n
...     print "%s() took %.2f ms" % (f, t_ms)
...
max() took 102.26 ms
numpy.max() took 4.37 ms
```

- useful for archiving and for complex setups
- must set loop count yourself
- `timeit()` result is the total (must divide by N)

cProfile

Provides detailed stats on *all* python functions executed during a profiled execution.

```
      72 function calls in 6.011 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    6.011    6.011 <string>:1(<module>)
     1    0.000    0.000    2.003    2.003 cp_ex.py:10(delay2)
     1    0.000    0.000    3.005    3.005 cp_ex.py:13(delay3)
     1    0.000    0.000    6.011    6.011 cp_ex.py:16(silly_delay)
     3    0.000    0.000    6.011    2.004 cp_ex.py:3(delay_loop)
     1    0.000    0.000    1.002    1.002 cp_ex.py:7(delay1)
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_ls
     3    0.000    0.000    0.000    0.000 {range}
    60    6.011    0.100    6.011    0.100 {time.sleep}
```

Using *cProfile*

```
#cProfile is in the standard library...
import cProfile

#Simple statement 'exec' profiling...
# - no filename prints the stats
cProfile.run(statement, filename)

#OR you can pass the environment for exec...
cProfile.runctx(statement, globals, locals)

#OR work with a Profile instance...
# - gives more control (see docs)
# - runcall is not documented, but useful
prof = cProfile.Profile()
prof.runcall(fn, *args, **kwargs)
prof.dump_stats(file_path)
```

Easier profiling with a decorator

```python
def profile_this(fn):
    def profiled_fn(*args, **kwargs):
        fpath = fn.__name__ + ".profile"
        prof = cProfile.Profile()
        ret = prof.runcall(fn, *args, **kwargs)
        prof.dump_stats(fpath)
        return ret
    return profiled_fn

@profile_this
def silly_delay():
    [f() for f in (delay1, delay2, delay3)]
```

We will use this decorator from now on

A simple/contrived *cProfile* example

```python
import cProfile, time

def delay_loop(n):
    for i in xrange(n):
        time.sleep(0.1)

def delay1():
    delay_loop(10)

def delay2():
    delay_loop(20)

def delay3():
    delay_loop(30)

@profile_this("silly_delay.profile")
def silly_delay():
    delay1()
    delay2()
    delay3()

cProfile.run("silly_delay()")
```

cProfile output for `silly_delay`

```
$ python cp_ex.py
         72 function calls in 6.011 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    6.011    6.011 <string>:1(<module>)
        1    0.000    0.000    2.004    2.004 cp_ex.py:10(delay2)
        1    0.000    0.000    3.006    3.006 cp_ex.py:13(delay3)
        1    0.000    0.000    6.011    6.011 cp_ex.py:16(silly_delay)
        3    0.000    0.000    6.011    2.004 cp_ex.py:3(delay_loop)
        1    0.000    0.000    1.002    1.002 cp_ex.py:7(delay1)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsp
        3    0.000    0.000    0.000    0.000 {range}
       60    6.011    0.100    6.011    0.100 {time.sleep}
```

cProfile output for `silly_delay`

```
$ python cp_ex.py
         72 function calls in 6.011 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    6.011    6.011 <string>:1(<module>)
        1    0.000    0.000    2.004    2.004 cp_ex.py:10(delay2)
        1    0.000    0.000    3.006    3.006 cp_ex.py:13(delay3)
        1    0.000    0.000    6.011    6.011 cp_ex.py:16(silly_delay)
        3    0.000    0.000    6.011    2.004 cp_ex.py:3(delay_loop)
        1    0.000    0.000    1.002    1.002 cp_ex.py:7(delay1)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsp
        3    0.000    0.000    0.000    0.000 {range}
       60    6.011    0.100    6.011    0.100 {time.sleep}
```

Note that there is no info on nested calls!

RunSnakeRun

Is a *much* nicer way to view cProfile results.

```
>>> profile = cProfile.Profile()
>>> profile.runcall(silly_delay)
>>> profile.dump_stats("silly_delay.profile")
```

```
$ runsnake silly_delay.profile
```

This gives you a convenient, and interactive, SquareMap view where <u>area is call time</u>...

runsnake silly_delay.profile

Name	Calls	RCalls	Local	/Call	Cum	/Call
	0	2	0.00...	0....	6....	3.00540
silly_delay	1	1	0.00...	0....	6....	6.01079
delay_loop	3	3	0.00...	0....	6....	2.00359
<time.sleep>	60	60	6.01...	0....	6....	0.10017
delay3	1	1	0.00...	0....	3....	3.00463
delay2	1	1	0.00...	0....	2....	2.00439
delay1	1	1	0.00...	0....	1....	1.00176
<range>	3	3	0.00...	0....	0....	0.00000
<method 'dis...	1	1	0.00...	0....	0....	0.00000

Callees | All Callees | Callers | All Callers | Source Code

Name	Calls	RCalls	Local	/Call	Cum	/Call
<time.sleep>	60	60	6.01046	0.10017	6.01046	0.1001
<range>	3	3	0.00001	0.00000	0.00001	0.0000

Note that there *is* nested call info here!

<Live RunSnakeRun Demo Here>

- details on hover/click
- "drill down" on dbl-click
- stats on clicked context
- file view
- % view

Time to optimize some SQLAlchemy!

Time to optimize some SQLAlchemy!

This is *NOT* an SQLAlchemy talk!
It's just a good/recent profiling example.
We'll do a *brief* intro for context *only*.

Time to optimize some SQLAlchemy!

This is *NOT* an SQLAlchemy talk!
It's just a good/recent profiling example.
We'll do a ***brief*** intro for context *only*.

(TL;DR: SQLAlchemy is awesome)

SQLAlchemy is an ORM+
... but what is an ORM?

ORM == "Object Relational Mapping"

An ORM lets you map your normal objects to/from a database, *without directly writing SQL*

1. Much simpler/faster development cycles
2. Database agnostic
   a. ie: Code base works for any DB
   b. (for a lot of use cases, anyway)

Actually - No time for any more intro!

Let's just dive in...

(You don't need to know SQLAlchemy to get the gist)

Actually - No time for any more intro!

Let's just dive in...

(You don't need to know SQLAlchemy to get the gist)

I do have a small demo prepared if people want, later.

Base setup (demo_base.py)

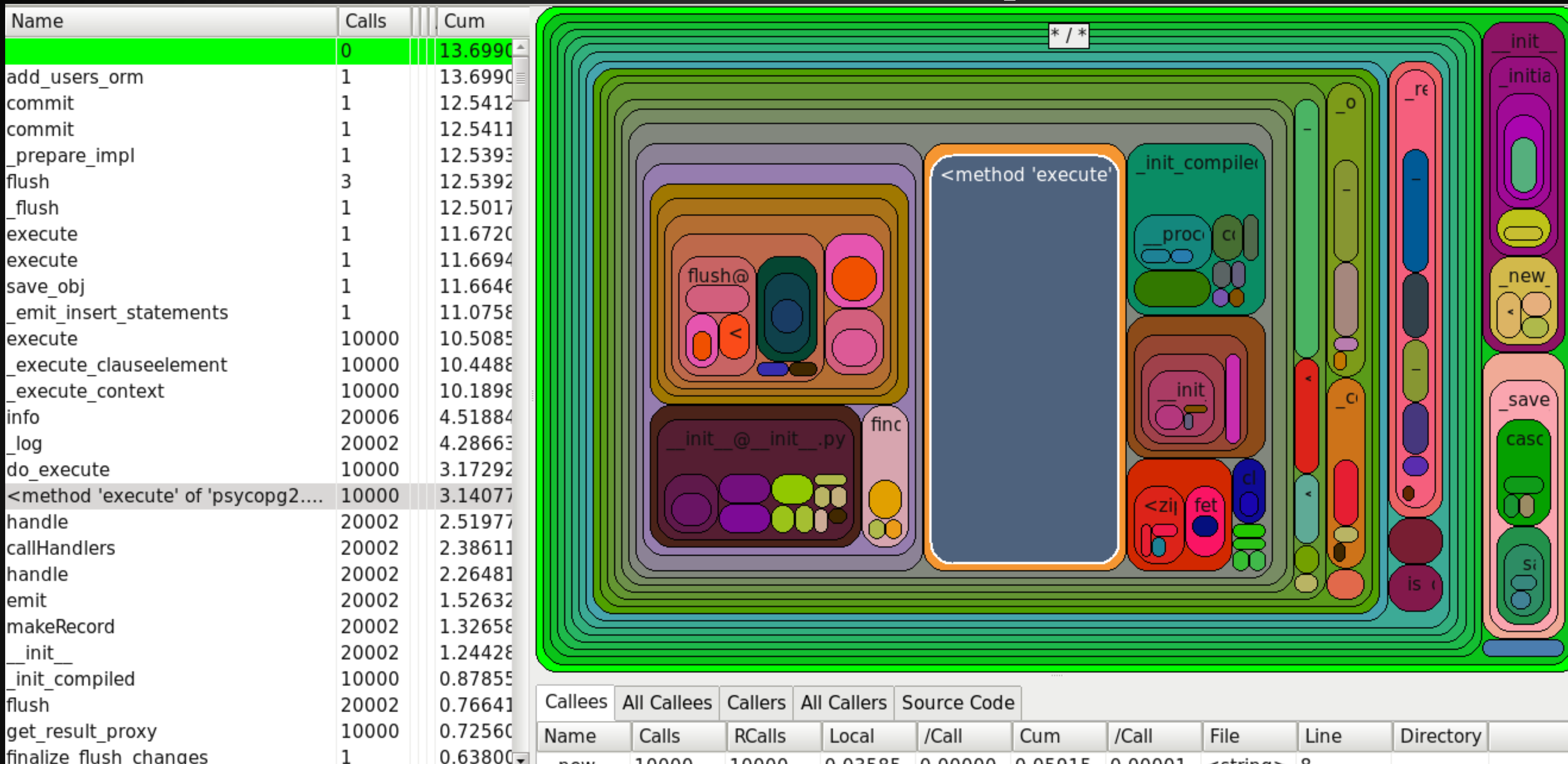
```python
import sqlalchemy as sa
from sqlalchemy.ext.declarative import declarative_base
import sqlalchemy.orm as orm

def drop_all_tables(engine):
    meta = sa.MetaData(engine)
    meta.reflect()
    meta.drop_all()

ORMBase = declarative_base()
class User(ORMBase):
    __tablename__ = "users"
    id = sa.Column(sa.Integer, primary_key = True, autoincrement = True)
    name = sa.Column(sa.String, unique = True)
    def __init__(self, Name):
        self.name = Name

def init():
    engine_str = "postgresql://postgres:postgres@localhost:5433/sandbox"
    #engine = sa.create_engine("sqlite://", echo = False)
    engine = sa.create_engine(engine_str, echo = False)
    orm_session = sa.orm.sessionmaker(bind = engine)
    drop_all_tables(engine)
    ORMBase.metadata.create_all(engine)
    return User, orm_session, engine
```

Bulk insert with ORM --- code

```
import demo_base
from profile_this import profile_this

User, orm_session, engine = demo_base.init()

@profile_this
def add_users_orm(n):
    user_names = ("USER%04d" % i for i in xrange(n))
    s = orm_session()
    for name in user_names:
        new_user = User(name)
        s.add(new_user)
    s.commit()

add_users_orm(10000)
```

(from `ex1_simple_orm.py`)

Bulk insert with ORM -- profile



Name	Calls	Cum
	0	13.6990
add_users_orm	1	13.6990
commit	1	12.5412
commit	1	12.5411
_prepare_impl	1	12.5393
flush	3	12.5392
_flush	1	12.5017
execute	1	11.6720
execute	1	11.6694
save_obj	1	11.6646
_emit_insert_statements	1	11.0758
execute	10000	10.5085
_execute_clauseelement	10000	10.4488
_execute_context	10000	10.1898
info	20006	4.51884
_log	20002	4.28663
do_execute	10000	3.17292
<method 'execute' of 'psycopg2....	10000	3.14077
handle	20002	2.51977
callHandlers	20002	2.38611
handle	20002	2.26481
emit	20002	1.52632
makeRecord	20002	1.32658
__init__	20002	1.24428
_init_compiled	10000	0.87855
flush	20002	0.76641
get_result_proxy	10000	0.72560
finalize_flush_changes	1	0.63800

Name	Calls	RCalls	Local	/Call	Cum	/Call	File	Line	Directory
new	10000	10000	0.03585	0.00000	0.05915	0.00001	<string>	8	

13.7 s total; only 3.1 s (23%) in `execute()` (DB API)

The rest is all ORM overhead.

Also note that there are 10k calls to `execute()`

<< Live viewing of profile here >>

runsnake add_users_orm.profile

DON'T PANIC

An ORM is not meant for bulk loading!!!

(Although you often see ORMs get grief for 'benchmarks' like this)

We can do better -- preload the PKs!

```
@profile_this
def add_users_with_pk(n):
    user_names = ("USER%04d" % i for i in xrange(n))
    s = orm_session()
    for i, name in enumerate(user_names):
        new_user = User(name)
        new_user.id = i #manual PK assignment
        s.add(new_user)
    s.commit()

add_users_with_pk(10000)
```

We can do better -- preload the PKs!

```
@profile_this
def add_users_with_pk(n):
    user_names = ("USER%04d" % i for i in xrange(n))
    s = orm_session()
    for i, name in enumerate(user_names):
        new_user = User(name)
        new_user.id = i #manual PK assignment
        s.add(new_user)
    s.commit()

add_users_with_pk(10000)
```

Wait... what? That is an autoincrement!

This helps the SQLAlchemy internals, resulting in *one* SQL emission instead of 10,000

Pre-loaded PKs -- profile

Name	Calls	Cum
	0	5.966
add_users_with_pk	1	5.966
commit	1	4.673
commit	1	4.673
_prepare_impl	1	4.671
flush	3	4.671
_flush	1	4.634
execute	1	3.800
execute	1	3.798
save_obj	1	3.792
_emit_insert_statements	1	3.192
execute	1	3.105
_execute_clauseelement	1	3.105
_execute_context	1	3.105
do_executemany	1	2.967
<method 'executemany'...	1	2.967
finalize_flush_changes	1	0.627
_register_newly_persistent	1	0.624
__init__	10000	0.558
add	10000	0.482
_save_or_update_state	10000	0.449
_initialize_instance	10000	0.349
__set__	20000	0.339
_organize_states_for_save	1	0.316
set	20000	0.265
_collect_insert_commands	1	0.250
__init__	10000	0.216
cascade_iterator	10000	0.205

6.0 s (> 2x faster!), and our square grew! :)

Still waay too much ORM overhead... :(

Notice the single call to the db api's `executemany()`

Scoresheet

Method	Total time	DB API time
Basic ORM	13.7 s	3.1 s (23%)
PK preload	6.0 s	3.0 s (50%)

- Pregenerating autoincrementing IDs may seem odd, but it helps a lot
    - ORM internals don't need to read back the autoinc id
- With PostgreSQL you can do this quickly/properly with `generate_series()`

Forget the ORM already... this is a bulk load!

ORMs were not made for this!

(Were you listening?)

SQLAlchemy's ORM is built on a lower level (but still amazing) expression language that has a lot less overhead.

Let's use that...

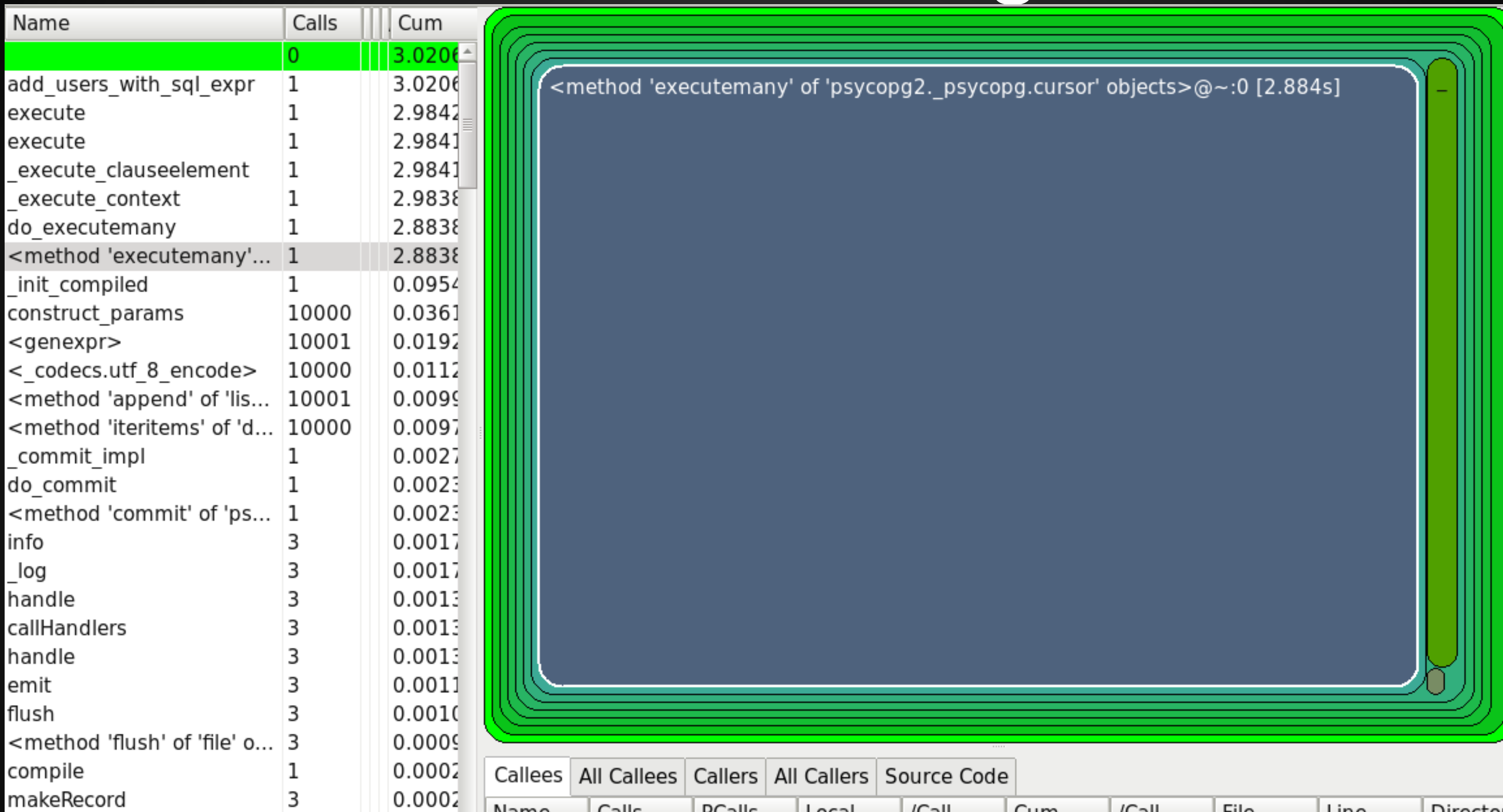
Using the SQLAlchemy Core

```
@profile_this
def add_users_with_sql_expr(n):
    user_names = ("USER%04d" % i for i in xrange(n))
    insert = User.__table__.insert()
    values = [{"name": name} for name in user_names]
    engine.execute(insert, values)

add_users_with_sql_expr(10000)
```

- Core constructs (like tables) are accessible through ORM objects
    - We get the table through `User.__table__`
- ORM and Core work well together
    - can also use session.execute() within an ORM session

(from `ex3_sql_expr.py`)

Profile for SQL Core usage



3.0 s (2x faster again!)

Look at that big square!! dbapi now 95% of call!!

Scoresheet

Method	Total time	DB API time
Basic ORM	13.7 s	3.1 s (23%)
PK preload	6.0 s	3.0 s (50%)
SQLAlchemy expr. lang.	3.0 s	2.9 s (95%)

Great!! We're almost as fast as the pure DB API!

We're done!!!

Scoresheet

Method	Total time	DB API time
Basic ORM	13.7 s	3.1 s (23%)
PK preload	6.0 s	3.0 s (50%)
SQLAlchemy expr. lang.	3.0 s	2.9 s (95%)

Great!! We're almost as fast as the pure DB API!

We're done!!! **RIGHT?!!**

That still seems slow...
Let's look behind the scenes:

```
INSERT INTO users (name) VALUES (%(name)s)
({'name': 'USER0000'}, {'name': 'USER0001'}, ...
```

Above is the output of SQLAlchemy's logging.

One INSERT! Very nice!

We also know we had one `executemany()` call.

That still seems slow...
Let's look behind the scenes:

```
INSERT INTO users (name) VALUES (%(name)s)
({'name': 'USER0000'}, {'name': 'USER0001'}, ...
```

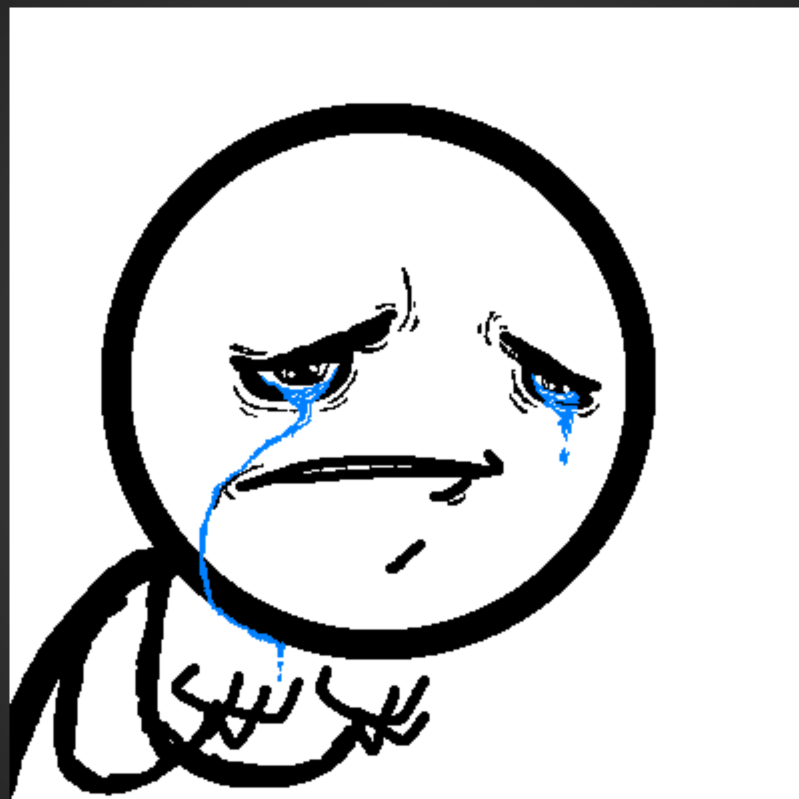
Above is the output of SQLAlchemy's logging.

One INSERT! Very nice!

We also know we had one `executemany()` call.

But... look at the PostgreSQL logs!!?!?!!

```
statement: INSERT INTO users (name) VALUES ('USER9996')
statement: INSERT INTO users (name) VALUES ('USER9997')
statement: INSERT INTO users (name) VALUES ('USER9998')
statement: INSERT INTO users (name) VALUES ('USER9999')
statement: COMMIT
```



cProfile can't see inside C extensions!!

- The DB API in this case (psycopg2) is a C-extension
    - *cProfile* profiles python. Not C.
- Internally, *psycopg2*'s `executemany()` implementation issues many INSERTs
    - The DB API spec allows this
    - This is out of SQLAlchemy's hands

LESSON: Profiling can be deceiving!

Let's do one more optimization...

Compile a direct SQL statement

```
col_names = User.__mapper__.columns.keys()
assert col_names == ["id", "name"] #in order of def

@profile_this
def add_users_with_compiled_sql(n):
    user_names = ("USER%04d" % i for i in xrange(n))
    stmt = "INSERT INTO {table} ({cols}) VALUES {vals};".\
        format(table = "users",
               cols = ", ".join(col_names),
               vals = ", ".join(["%s"] * n),
               )
    data = list(enumerate(user_names)) #must support indexing
    cursor = engine.connect().connection.cursor()
    sql = cursor.mogrify(stmt, data)
    engine.execute(sql)
```

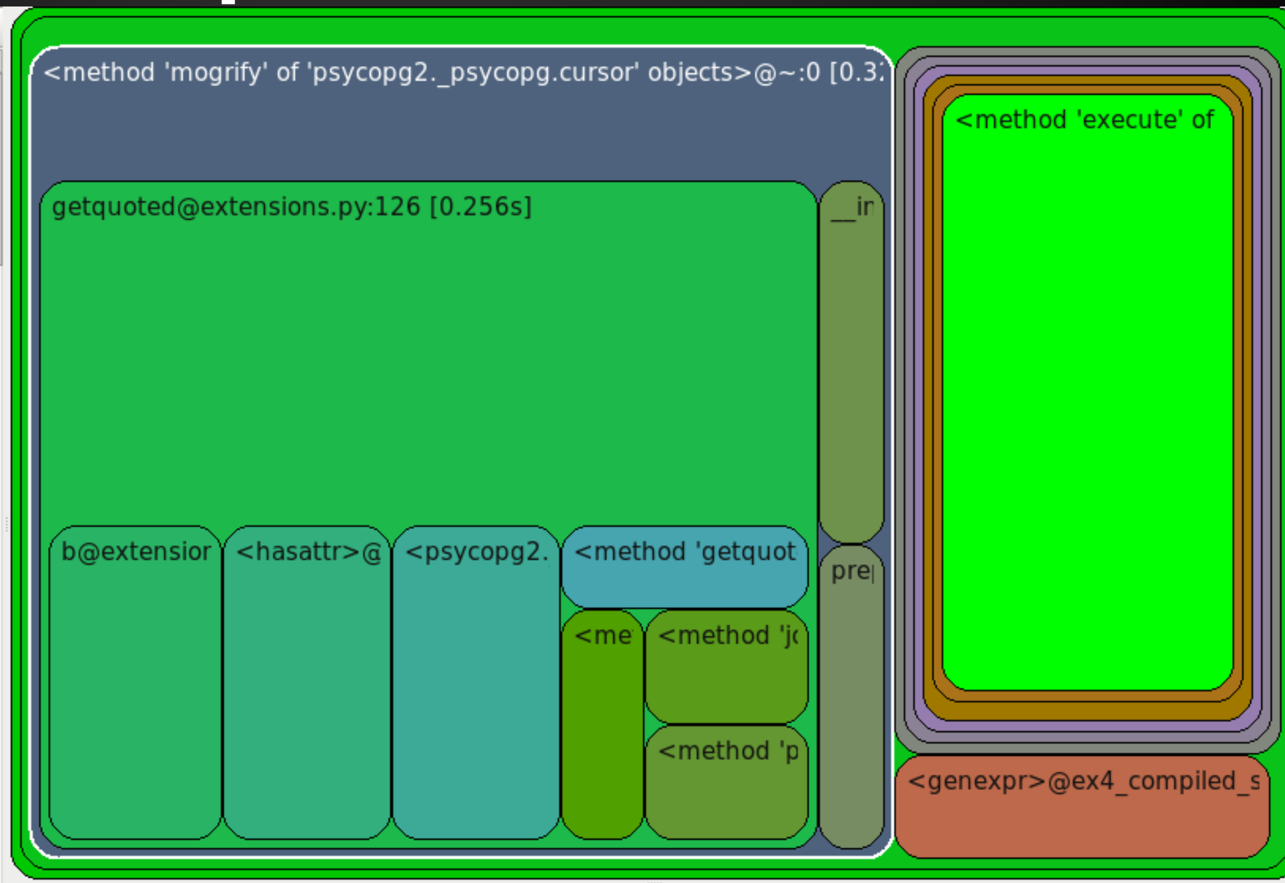
Directly accesses the DB API from SQLAlchemy

With ORM, use `s.connection().connection.cursor`

(from `ex4_compiled_sql.py`)

Profile for compiled SQL statement

Name	Calls	Cum
	0	0.48805
add_users_with_co...	1	0.48805
<method 'mogrify' ...	1	0.32914
getquoted	10000	0.25581
execute	1	0.12761
execute	1	0.12749
_execute_text	1	0.12748
_execute_context	1	0.12747
do_execute	1	0.12555
<method 'execute' ...	1	0.12554
b	30000	0.02864
<hasattr>	20009	0.02764
<psycopg2._psycop...	20000	0.02738
<genexpr>	10001	0.01846
__init__	10000	0.01167
<method 'getquote...	10000	0.01087
<method 'join' of 'st...	10002	0.01000
<method 'getquote...	10000	0.00996
prepare	10000	0.00980
<method 'prepare' ...	10000	0.00968
_commit_impl	1	0.00095
info	3	0.00093
_log	3	0.00090
do_commit	1	0.00064



- 0.49 s (> 6x faster again!)
- profile now looks *very* different
- note that compilation to SQL is ~ 67% of time!

Scoresheet

Method	Total time	DB API time
Basic ORM	13.7 s	3.1 s (23%)
PK preload	6.0 s	3.0 s (50%)
SQLAlchemy expr. lang.	3.0 s	2.9 s (95%)
Compiled SQL	0.49 s	~100% *

We're done for sure now.

We *could* go further... eg: COPY vs INSERT (postgres only), Cython that mogrify call, etc.

But the profiling point is proven.

Who am I kidding?

2/3 time generating a SQL string?!

We're not done yet!

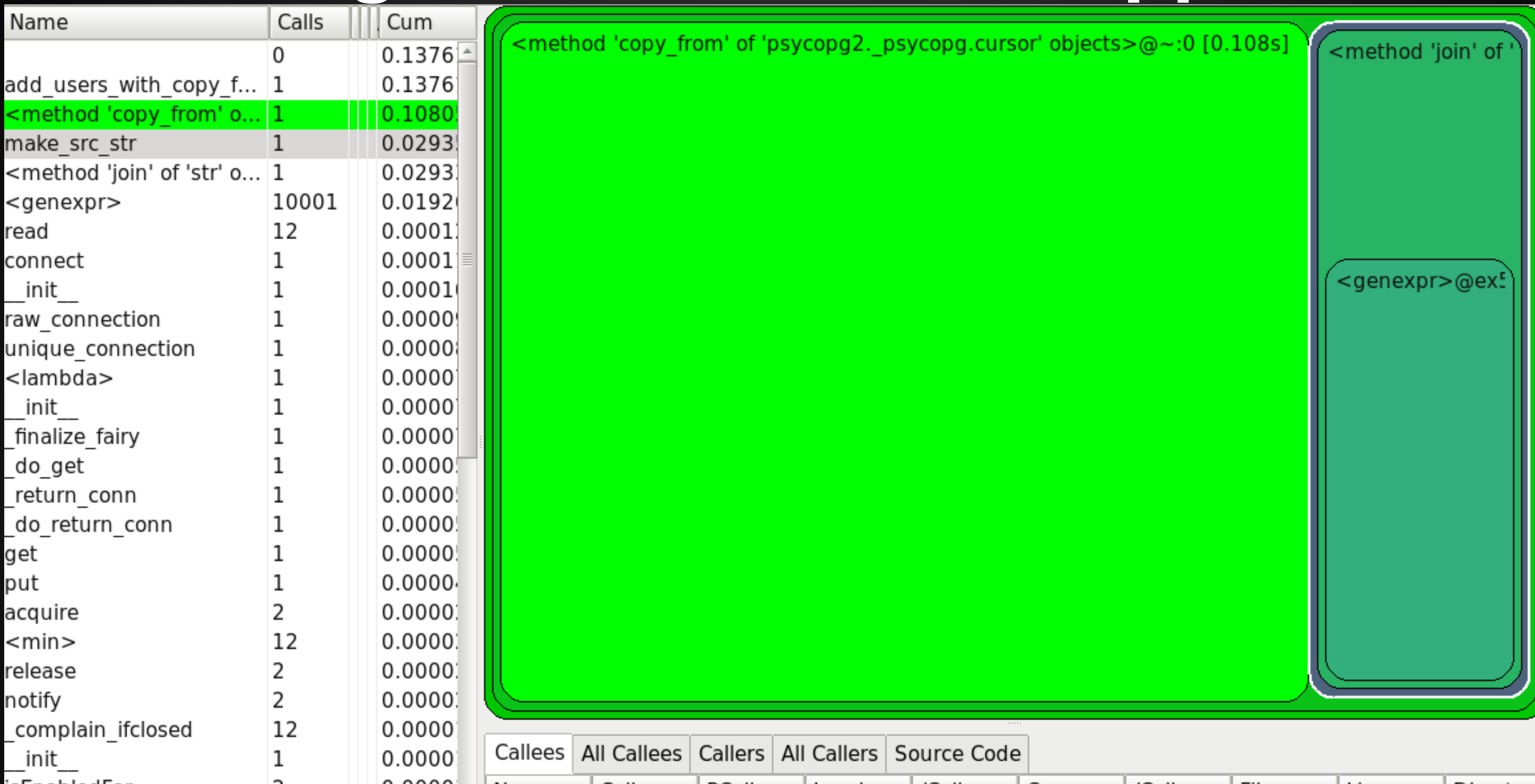
Using PostgreSQL's COPY FROM

```python
from StringIO import StringIO #no unicode for cStringIO :(
@profile_this
def add_users_with_copy_from(n):
    user_names = ("USER%04d" % i for i in xrange(n))
    def make_src_str(): #for profile visibility
        return StringIO("\n".join(user_names) + "\n")
    src = make_src_str()
    cursor = engine.connect().connection.cursor()
    cursor.copy_from(src, "users", columns = ("name", ))

add_users_with_copy_from(10000)
```

Notice the (seemingly) pointless local function

(from `ex5_copy_from.py`)

Profiling the COPY FROM approach



0.14 s (> 3x faster!) - StringIO formation is ~20%!

The local function trick was cheap and improves the profile. Use functions to organize profiling!

Scoresheet

Method	Total time	DB API time
Basic ORM	13.7 s	3.1 s (23%)
PK preload	6.0 s	3.0 s (50%)
SQLAlchemy expr. lang.	3.0 s	2.9 s (95%)
Compiled SQL	0.49 s	~100% *
COPY FROM (postgres)	0.14 s	80%

Scoresheet

Method	Total time	DB API time
Basic ORM	13.7 s	3.1 s (23%)
PK preload	6.0 s	3.0 s (50%)
SQLAlchemy expr. lang.	3.0 s	2.9 s (95%)
Compiled SQL	0.49 s	~100% *
COPY FROM (postgres)	0.14 s	80%

We're done optimizing for real now. Honest.

Scoresheet

Method	Total time	DB API time
Basic ORM	13.7 s	3.1 s (23%)
PK preload	6.0 s	3.0 s (50%)
SQLAlchemy expr. lang.	3.0 s	2.9 s (95%)
Compiled SQL	0.49 s	~100% *
COPY FROM (postgres)	0.14 s	80%

We're done optimizing for real now. Honest.

But we *could* go further... that copy_from block could use the BINARY option which would shrink it, and copy_from's use of StringIO (pure python) is bound to be slow so that could likely be improved... what's REALLY happening inside psycopg2_copy_from, anyway? I bet I could optimize the heck out of that massive join/genexp combo with Cython, too... We could... etc etc

Scoresheet

Method	Total time	DB API time
Basic ORM	13.7 s	3.1 s (23%)
PK preload	6.0 s	3.0 s (50%)
SQLAlchemy expr. lang.	3.0 s	2.9 s (95%)
Compiled SQL	0.49 s	~100% *
COPY FROM (postgres)	0.14 s	80%

We're done optimizing for real now. Honest.

But we *could* go further... that copy_from block could use the BINARY option which would shrink it, and copy_from's use of StringIO (pure python) is bound to be slow so that could likely be improved... what's REALLY happening inside psycopg2_copy_from, anyway? I bet I could optimize the heck out of that massive join/genexp combo with Cython, too... We could... etc etc

BE CAREFUL OF DIMINISHING RETURNS!!!

It isn't that you *shouldn't* try and make your code be as fast as possible.

Having a responsive application is *definitely* a noble goal.

It's that your time is probably better spent on other code where you can have greater *overall* impact.

Use profiling to tell you where.

Now... how much impact did the profiling itself have?

At this level, and with large numbers of function calls, the 'instrumenting' that the profiler does to make the measurements can be large.

Be aware of this.

Let's use good ol' time.time() to remeasure...

A crude timing decorator

```
def time_this(fn):
    def timed_fn(*args, **kwargs):
        start_s = time.time()
        ret = fn(*args, **kwargs)
        elapsed_s = time.time() - start_s
        print "%s(...) took %f s" % (fn.__name__, elapsed_s)
        return ret
    return timed_fn
```

This could be a *lot* better but it's fine for now.

Let's re-run all profiles with `@time_this` instead of `@profile_this`...

Scoresheet - cProfile vs time.time()

Method	Total time (cProfile)	Total time (time.time)	Ratio (~)
Basic ORM	13.7 s	7.4 s	1.9 x
PK preload	6.0 s	3.8 s	1.6 x
SQLAlchemy expr. lang.	3.0 s	3.0 s	1.0 x
Compiled SQL	0.49 s	0.20 s	2.4 x
COPY FROM (postgres)	0.14 s	0.10 s	1.4 x

Scoresheet - cProfile vs time.time()

Method	Total time (cProfile)	Total time (time.time)	Ratio (~)
Basic ORM	13.7 s	7.4 s	1.9 x
PK preload	6.0 s	3.8 s	1.6 x
SQLAlchemy expr. lang.	3.0 s	3.0 s	1.0 x
Compiled SQL	0.49 s	0.20 s	2.4 x
COPY FROM (postgres)	0.14 s	0.10 s	1.4 x

NOTE: This timing data is a bit sloppy!

Timing short runs (esp. with I/O) is not reliable or repeatable. More runs+records is better.

Still, it's clear that profiling can have overhead.

Final words on profiling

- Profile before optimizing
  - Only optimize when/where you need to.
  - Make it work first!
  - "Premature optimization is the root of all evil"
- Be smart
  - Beware of diminishing returns! Bigger gains can probably be made elsewhere. *Profile!*
  - Profiles can be deceiving (and be aware of overhead!)
  - Use all of your tools (timing, logs, etc)
  - Pick your battles. Sometimes enough is enough.
- Make profiling easier to use in your workflow
  - A decorator makes it very convenient
  - `ServerProxy.ProfileNextCall()`
- Use a viewer like RunSnakeRun

Links and versions

cProfile

- docs: http://goo.gl/X7rHv

RunSnakeRun

- http://goo.gl/rW7sV
- version used: 2.0.2b1

SqlAlchemy

- www.sqlalchemy.org
- version used: 0.8.0b2