How can I profile a SQLAlchemy powered application?

Does anyone have experience profiling a Python/SQLAlchemy app? And what are the best way to find bottlenecks and design flaws?

We have a Python application where the database layer is handled by SQLAlchemy. The application uses a batch design, so a lot of database requests is done sequentially and in a limited timespan. It currently takes a bit too long to run, so some optimization is needed. We don't use the ORM functionality, and the database is PostgreSQL.

python sqlalchemy profiler

asked Jul 23 '09 at 11:33

goxe
276 ●5 ●7

3 Answers

Sometimes just plain SQL logging (enabled via python's logging module or via the `echo=True` argument on `create_engine()`) can give you an idea how long things are taking. For example if you log something right after a SQL operation, you'd see something like this in your log:

```
17:37:48,325 INFO  [sqlalchemy.engine.base.Engine.0x...048c] SELECT ...
17:37:48,326 INFO  [sqlalchemy.engine.base.Engine.0x...048c] {<params>}
17:37:48,660 DEBUG [myapp.somemessage]
```

if you logged `myapp.somemessage` right after the operation, you know it took 334ms to complete the SQL part of things.

Logging SQL will also illustrate if dozens/hundreds of queries are being issued which could be better organized into much fewer queries via joins. When using the SQLAlchemy ORM, the "eager loading" feature is provided to partially (`contains_eager()`) or fully (`eagerload()`, `eagerload_all()`) automate this activity, but without the ORM it just means to use joins so that results across multiple tables can be loaded in one result set instead of multiplying numbers of queries as more depth is added (i.e. `r + r*r2 + r*r2*r3` ...)

If logging reveals that individual queries are taking too long, you'd need a breakdown of how much time was spent within the database processing the query, sending results over the network, being handled by the DBAPI, and finally being received by SQLAlchemy's result set and/or ORM layer. Each of these stages can present their own individual bottlenecks, depending on specifics.

For that you need to use profiling, such as cProfile or hotshot. Here is a decorator I use:

```
import cProfile as profiler
import gc, pstats, time

def profile(fn):
    def wrapper(*args, **kw):
        elapsed, stat_loader, result = _profile("foo.txt", fn, *args, **kw)
        stats = stat_loader()
        stats.sort_stats('cumulative')
        stats.print_stats()
        # uncomment this to see who's calling what

        # stats.print_callers()
        return result
    return wrapper

def _profile(filename, fn, *args, **kw):
    load_stats = lambda: pstats.Stats(filename)
    gc.collect()

    began = time.time()
    profiler.runctx('result = fn(*args, **kw)', globals(), locals(),
                    filename=filename)
```

```python
    ended = time.time()

    return ended - began, load_stats, locals()['result']
```

To profile a section of code, place it in a function with the decorator:

```python
@profile
def go():
    return Session.query(FooClass).filter(FooClass.somevalue==8).all()
myfoos = go()
```

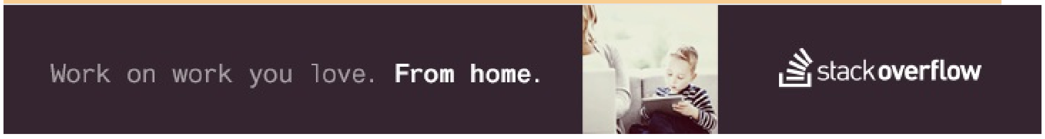
The output of profiling can be used to give an idea where time is being spent. If for example you see all the time being spent within `cursor.execute()` , that's the low level DBAPI call to the database, and it means your query should be optimized, either by adding indexes or restructuring the query and/or underlying schema. For that task I would recommend using pgadmin along with its graphical EXPLAIN utility to see what kind of work the query is doing.

If you see many thousands of calls related to fetching rows, it may mean your query is returning more rows than expected - a cartesian product as a result of an incomplete join can cause this issue. Yet another issue is time spent within type handling - a SQLAlchemy type such as `Unicode` will perform string encoding/decoding on bind parameters and result columns, which may not be needed in all cases.

The output of a profile can be a little daunting but after some practice they are very easy to read. There was once someone on the mailing list claiming slowness, and after having him post the results of profile, I was able to demonstrate that the speed problems were due to network latency - the time spent within cursor.execute() as well as all Python methods was very fast, whereas the majority of time was spent on socket.receive().

If you're feeling ambitious, there's also a more involved example of SQLAlchemy profiling within the SQLAlchemy unit tests, if you poke around http://www.sqlalchemy.org/trac/browser/sqlalchemy/trunk/test/aaa_profiling . There, we have tests using decorators that assert a maximum number of method calls being used for particular operations, so that if something inefficient gets checked in, the tests will reveal it (it is important to note that in Python, function calls have the highest overhead of any operation, and the count of calls is more often than not nearly proportional to time spent). Of note are the the "zoomark" tests which use a fancy "SQL capturing" scheme which cuts out the overhead of the DBAPI from the equation - although that technique isn't really necessary for garden-variety profiling.

answered Jul 24 '09 at 3:54
zzzeek
33.7k ● 7 ● 90 ● 113



There's an extremely useful profiling recipe on the SQLAlchemy wiki

With a couple of minor modifications,

```python
from sqlalchemy import event
from sqlalchemy.engine import Engine
import time
import logging

logging.basicConfig()
logger = logging.getLogger("myapp.sqltime")
logger.setLevel(logging.DEBUG)

@event.listens_for(Engine, "before_cursor_execute")
def before_cursor_execute(conn, cursor, statement,
                        parameters, context, executemany):
    context._query_start_time = time.time()
    logger.debug("Start Query:\n%s" % statement)

    # Modification for StackOverflow answer:
    # Show parameters, which might be too verbose, depending on usage..
    logger.debug("Parameters:\n%r" % (parameters,))


@event.listens_for(Engine, "after_cursor_execute")
def after_cursor_execute(conn, cursor, statement,
                        parameters, context, executemany):
    total = time.time() - context._query_start_time
    logger.debug("Query Complete!")

    # Modification for StackOverflow: times in milliseconds
    logger.debug("Total Time: %.02fms" % (total*1000))

if __name__ == '__main__':
    from sqlalchemy import *

    engine = create_engine('sqlite://')

    m1 = MetaData(engine)
    t1 = Table("sometable", m1,
            Column("id", Integer, primary_key=True),
            Column("data", String(255), nullable=False),
            Column('data', String(255), nullable=False),
        )

    conn = engine.connect()
    m1.create_all(conn)

    conn.execute(
        t1.insert(),
        [{"data":"entry %d" % x} for x in xrange(100000)]
    )

    conn.execute(
        t1.select().where(t1.c.data.between("entry 25", "entry
7800")).order_by(desc(t1.c.data))
    )
```

Output is something like:

```
DEBUG:myapp.sqltime:Start Query:
SELECT sometable.id, sometable.data
FROM sometable
WHERE sometable.data BETWEEN ? AND ? ORDER BY sometable.data DESC
DEBUG:myapp.sqltime:Parameters:
('entry 25', 'entry 7800')
DEBUG:myapp.sqltime:Query Complete!
DEBUG:myapp.sqltime:Total Time: 410.46ms
```

Then if you find an oddly slow query, you could take the query string, format in the parameters (can be done the `%` string-formatting operator, for psycopg2 at least), prefix it with "EXPLAIN ANALYZE" and shove the query plan output into http://explain.depesz.com/ (found via this good article on PostgreSQL performance)

answered Dec 8 '11 at 9:06
dbr
88.7k ●43 ●216 ●289

I have had some success in using cprofile and looking at the results in runsnakerun. This at least told me what functions and calls where taking a long time and if the database was the issue. The documentation is here. You need wxpython. The presentation on it is good to get you started.
Its as easy as

```
import cProfile
command = """foo.run()"""
cProfile.runctx( command, globals(), locals(), filename="output.profile" )
```

Then

python runsnake.py output.profile

If you are looking to optimise your queries you will need postgrsql profiling.

It is also worth putting logging on to record the queries, but there is no parser for this that I know of to get the long running queries (and it wont be useful for concurrent requests).

```
sqlhandler = logging.FileHandler("sql.log")
sqllogger = logging.getLogger('sqlalchemy.engine')
sqllogger.setLevel(logging.info)
sqllogger.addHandler(sqlhandler)
```

and making sure your create engine statement has echo = True.
When I did it is was actually my code that was the main issue, so the cprofile thing helped.

edited Jul 23 '09 at 12:14

answered Jul 23 '09 at 12:00
David Raznick
7,317 ●1 ●18 ●25