MysqlSimpleQueryBuilder Manual

Release 0.4.0

saaj

May 13, 2018

Contents

1 Autocommit **3**

2 Nested transaction **5**

3 Implicit transaction completion **7**

4 Concurrency **9**

5 Persistent connection **11**

6 MySQL versus **13**

7 Install **15**
7.1 Adapter 15
7.2 Performance 16

8 Reference **17**
8.1 Facade 17
8.2 Cursor 20
8.3 Internal 21

9 Profiling **23**
9.1 Example 24

10 Mysql Simple Query Builder **27**
10.1 Scope 27

Briefly about the main module `myquerybuilder.builder`. It is about two hundred logical lines of code. It's mere string building that gives up most aspects of operation, like escaping, connectivity and types, to the underlying MySQL adapters. Nothing really technically fancy, but rather simple thin convenience wrapper.

For supported adapters see *Install*.

CHAPTER 1

Autocommit

PEP-249[1] proposes (it doesn't mandate if you still remember what the last P stands for), auto-commit to be initially off. To speak it outright in case of MySQL, it is **blatantly wrong**.

Not only it is counterintuitive, different to DBAPIs in other languages, breaks the Zen of Python, *explicit is better than implicit*, and leads to countless and constant surprise of beginners like *where is my data?*, *why does MyISAM work but InnoDB doesn't?*, *why does SELECT work but UPDATE doesn't?*, *Why doesn't MySQL INSERT into database?* and so on and so forth[2][3][4]. It also leads more established developers to scratch their heads, because InnoDB's default transaction isolation is *REPEATABLE READ*[5], and the reader will only receive its first snapshot[6].

Read-only transactions lead to many performance issues[7][8] and complicate application design when frameworks do weird twist pulling out consistency matters from their inherent domain model origin to request level. In that way even monstrous things like Django with huge codebases and the established practice rethink the decision and enable auto-commit[9].

Because decisions in domain model consistency should be conscious and elaborate, *MySQL Simple Query Builder* sets auto-commit to the server default setting, which is usually *ON*.

[1] https://www.python.org/dev/peps/pep-0249/
[2] http://stackoverflow.com/q/1028671/2072035
[3] http://stackoverflow.com/q/1451782/2072035
[4] http://stackoverflow.com/q/14445090/2072035
[5] http://dev.mysql.com/doc/refman/5.5/en/set-transaction.html#isolevel_repeatable-read
[6] http://stackoverflow.com/q/25991345/2072035
[7] https://blogs.oracle.com/mysqlinnodb/entry/better_scaling_of_read_only
[8] https://mariadb.com/blog/every-select-your-python-program-may-acquire-metadata-lock
[9] https://docs.djangoproject.com/en/1.7/topics/db/transactions/#autocommit

CHAPTER 2

Nested transaction

InnoDB supports transaction savepoints[10]. These are named sub-transactions that run in the scope of a normal transaction. Having `begin`, `commit` and `rollback` that always behave in a semantically correct way is usually beneficial. It's so for atomic domain object methods' that may call other atomic methods, for unit and integration tests and other cases. Here's how nested calls correspond to queries.

call	query
qb.begin()	BEGIN
qb.begin()	SAVEPOINT LEVEL1
qb.begin()	SAVEPOINT LEVEL2
qb.rollback()	ROLLBACK TO SAVEPOINT LEVEL2
qb.commit()	RELEASE SAVEPOINT LEVEL1
qb.rollback()	ROLLBACK
qb.begin()	BEGIN
qb.commit()	COMMIT

Warning: In MySQL you can query unmatched *BEGIN*, *COMMIT* and *ROLLBACK* without an error. For instance you can *ROLLBACK* without preceding *BEGIN*, or several *BEGIN* queries in row. MySQL just ignores the queries that make no sense. This is no longer the case with *myquerybuilder.builder.QueryBuilder* which maintains stack order transaction level counter, and raises `OperationalError` on unmatched calls.

If certain functionality doesn't need to be atomic or possible speedup from grouping statements into transactions is negligible, just rely in auto-commit and don't bother. Otherwise if you have decided to employ a transaction make sure you have the idea what ACID[11] is, and what impact InnoDB transaction isolation levels have[17]. It's important for your transactional code to be properly wrapped in `try...except` clause:

[10] http://dev.mysql.com/doc/refman/5.5/en/savepoint.html
[11] http://en.wikipedia.org/wiki/ACID
[17] http://dimitrik.free.fr/blog/archives/02-01-2015_02-28-2015.html

```
qb.begin()
try:
  # your workload
  qb.commit()
except:
  qb.rollback()
  raise
```

To reduce the boilerplate a simple decorator can be implemented as follows. Note that the example is applicable only to methods (or functions whose first argument has attribute `_db`):

```
import functools

def transaction(fn):
  '''Decorator that wraps a model method into transaction'''

  @functools.wraps(fn)
  def _transaction(self, *args, **kwargs):
    self._db.begin()
    try:
      result = fn(self, *args, **kwargs)
      self._db.commit()
    except:
      self._db.rollback()
      raise
    else:
      return result

  return _transaction
```

CHAPTER 3

Implicit transaction completion

MySQL can implicitly (i.e. without your direct command), commit or roll back a transaction. If you use nested transactions and have the following exception, it's very likely it is the case.

```
OperationalError: (1305, 'SAVEPOINT LEVEL1 does not exist')
```

MySQL implicitly commits a transaction in case of most DDLs, system tables modification, locking statements, data loading statements, administrative statements and more[12]. Implicit rollback occurs when[13]:

- Transaction deadlock is detected
- Lock wait is timed out (until 5.0.13 or if configured explicitly thereafter)
- Reconnection happened

The missing savepoint error is very likely originated from a deadlock. Take a look at `myquerybuilder.test.builder.TestQueryBuilderTransaction` for simulated implicit transaction completion.

[12] http://dev.mysql.com/doc/refman/5.5/en/implicit-commit.html
[13] http://dev.mysql.com/doc/refman/5.0/en/innodb-error-handling.html

CHAPTER 4

Concurrency

Underlying MySQL adapters are not thread-safe so isn't the library. If you will share the query builder instances, and thus the DBAPI connection object, among threads most likely outcome for a libmysqlclient-based adapter is segmentation fault and crash of whole Python process. For pure-Python adapter you will face weird behaviour, lost data and out-of-sync errors.

Suggested way is one of the simplest. Use thread-mapped persistent connections. It is perfectly fine to have several dozens of established MySQL connections[14]. This way any overhead related to establishing connection is eliminated and you get your data as soon as possible.

Though a transparent connection pooling or thread-mapping may appear in future releases as it simplifies things.

[14] http://stackoverflow.com/a/99565/2072035

CHAPTER 5

Persistent connection

Persistent connection to MySQL server has its performance benefits, which are paid off by the need to maintain the state and availability. For figuring out whether connection is alive while staying idle for a long time, *ping* MySQL API function is used. It is also the reconnection function. The library's counterpart is *ping()*.

Note: In case of web application and persistent connection, you need to call `qb.ping(True)` before processing a request to ensure that connection is alive and to reconnect if it was lost.

Alternatively, you can wrap your public methods or classes with decorator as follows:

```
def ping(clsOrFn):
  '''Class or function decorator that pings database connecting before execution
  of the function(s)'''

  if isinstance(clsOrFn, type):
    for name, member in clsOrFn.__dict__.items():
      if not name.startswith('_') and isinstance(member, types.FunctionType):
        setattr(clsOrFn, name, ping(member))

    return clsOrFn
  elif isinstance(clsOrFn, types.FunctionType):
    @functools.wraps(clsOrFn)
    def _ping(self, *args, **kwargs):
      self._db.ping()
      return clsOrFn(self, *args, **kwargs)

    return _ping
  else:
    raise TypeError
```

MySQL also has auto-reconnection feature[16]. It was even enabled by default from MySQL 5.0.0 to 5.0.3. It allows silent reconnection on any query when the client has found that the connection was lost. It may sound like a handy thing, but in fact it is a dangerous and discouraged one.

[16] http://dev.mysql.com/doc/refman/5.0/en/auto-reconnect.html

When reconnection does occur, no matter silent or deliberate, the connection's state is severely affected, including but not limited to:

- Active transaction is rolled back
- All table locks are released
- All temporary tables are closed and dropped
- User variable settings are lost

In worst case the server may not yet know that the client has lost connection, thus the client that has just reconnected may find itself in a separate to its old transaction.

The subtle point with auto-reconnection and pinging is that MySQL-python (and mysqlclient as a fork) accepts `reconnect` argument for connection's method `ping`, which has a side-effect that sets connection-wide auto-reconnection behaviour. In other words:

```
import MySQLdb
conn = MySQLdb.connect(user = 'guest')
conn.ping(True)
```

And your connection works in auto-reconnection mode and you should expect the unexpected later on. It is subtle because ping's docstring is the only source and it's defined in `_mysql.c`, so you see one only if you ask it, `help(conn.ping)`. There's just a brief remark:

> Accepts an optional reconnect parameter. If True, then the client will attempt reconnection. Note that **this setting is persistent**.

Therefore, to avoid the persistent side-effect when called with `True` reconnection argument *`ping()`* calls underlying ping method with `False` second time.

For simulated behaviour, see `myquerybuilder.test.builder.TestQueryBuilder.testPing`.

CHAPTER 6

MySQL versus

Here goes a note about what makes MySQL (which usually should be read as InnoDB) "special" and why it has been chosen for the library in the first place. Or in other words *why not PostgreSQL?* which you may hear claimed superior and "true" database (as opposite to MySQL) here and there.

I can't say anything good or bad about PostgreSQL. It is just cut off. That is when one submits to Occam's razor to gain mastery with one thing, rather what acquaintance with many. Thus nothing makes it special in the first place but the historic choice. However I will quote a translation of one good related comment I read long ago, which has only survived in my personal correspondence:

> You may notice that Google hasn't overgrown MySQL and still uses it in replicated setup with hundreds of geographically distributed servers. Makes sense? Why do Facebook, Yahoo and other giants nearly the same thing? So what makes your project so special that you say you have overgrown it?
>
> There is no doubt that true scalability is sharding. Full-featured SQL, when you never shy to *JOIN* any table, unfortunately, doesn't scale. In no way. Just forget about it. Eventually, you will be sharding your data and your business logic will retrace back to middleware. Where it belongs, though.
>
> Of course it disturbs "true" RDBMS vendors who wish as most business logic as possible to reside inside the database. Otherwise they have nothing to sell you. So far as business logic lives in middleware, only a reliable storage is needed. This is what MySQL can cope with. Don't trust me? Ask Google.

CHAPTER 7

Install

MySQL Simple Query Builder can employ three MySQL client adapters which are either equivalent or compatible by design:

- PyMySQL
- mysqlclient
- MySQL-python

The library works on Python 2, Python 3 and PyPy. At server side all MySQL-family should work, including MySQL, MariaDB, Percona Server, Sphinx Search and probably others. Though CI is made only for MySQL[1].

7.1 Adapter

For actual combination of supported Python versions and MySQL adapters, see tox file[2].

7.1.1 PyMySQL

This is a pure-Python adapter, so it is straightforward:

```
pip install MysqlSimpleQueryBuilder[pymysql]
```

7.1.2 mysqlclient

This one made recent appearance, and in fact it is a fork of MySQL-python. But the fork has remarkable qualities. Unlike long-stale MySQL-python and its possible future version called *moist*, it also supports Python 3 and PyPy.

[1] https://drone.io/bitbucket.org/saaj/mysql-simple-query-builder
[2] https://bitbucket.org/saaj/mysql-simple-query-builder/src/default/tox.ini

Moreover it's being actively developed by the same PyMySQL people[3], who port bugs from the MySQL-python bug tracker, and overall making a great job!

To be able to build C part of the client on Debian you, of course need `build-essential`, corresponding development version of Python, for instance `python-dev` for Python 2, and `libmysqlclient-dev`:

```
apt-get install build-essential python-dev libmysqlclient-dev
```

Then:

```
pip install MysqlSimpleQueryBuilder[mysqlclient]
```

7.1.3 MySQL-python

This is our old default Python 2-only thing. The same OS packages are requires as for mysqlclient:

```
pip install MysqlSimpleQueryBuilder[mysqldb]
```

7.2 Performance

Here goes a quick benchmark of the supported environments to give you a clue of expectation. I set `myquerybuilder.test.benchmark.Benchmark.repeat = 10240`, commented out `py27-mysqldb` commands in tox, which also serves a role of code coverage environment and run:

```
for i in {1..8}; do tox -- -s myquerybuilder.test.benchmark.Benchmark.
↪testSelectManyFields; done
```

As it wasn't the point to benchmark MySQL per se, all queries are served from query cache. For PyMySQL it was ~95% CPU utilization by Python process, ~8% by mysqld. For libmysqlclient ~90% and ~15%, respectively.

It was run on Linux Mint Olivia with Intel Core i5 1.8GHz x 2 with HT. MySQL 5.5.34, Python 2.7.4 and 3.3.1 were from Ubuntu repository. Python 3.4.2 was from *deadspakes* PPA, PyPy 2.4.0 (implements equivalent of Python 2.7.8) from PyPy PPA (see tox *setupdrone* environment[2]).

There's no surprise, but I will comment on a few observations anyway:

- PyPy is good for pure-Python libraries
- PyPy is bad for libraries with native C calls
- deadsnakes is test-only
- mysqlclient and MySQL-python are neck and neck

[3] https://github.com/PyMySQL/

CHAPTER 8

Reference

This page is reference manual for the builder functionality. For more examples take a look at the project's test suite[1]. As well as the following snippets, the test suite is built against Sakila test database[2].

8.1 Facade

class myquerybuilder.builder.**QueryBuilder**(*clauseBuilder=<class 'myquerybuilder.builder.ClauseBuilder'>*, ***kwargs*)

The package's facade. The initializer sets *autocommit* to server's default and establishes connection.

```
from myquerybuilder import QueryBuilder

qb = QueryBuilder(user = 'guest', passwd = 'pass', db = 'sakila')
```

cursorType = {<type 'tuple'>: <class 'myquerybuilder.builder.NamedCursor'>, <type 'di

The alias mapping that *cursor()* will look up for actual cursor class.

select(*fields*, *table*, *where=None*, *order=None*, *limit=None*)

Executes a *SELECT* query with the specified clauses.

param fields `str` sequence of fields to fetch. When it is an one-element sequence, return value is tuple of scalars, otherwise return value is tuple of `dict` values. If no row is matched, empty tuple is returned.

param table `str` of table name.

param where `dict` of conditions which is applied as a conjunction and whose values can be scalar or vector. These values are values that can be escaped by the underlying MySQL diver. Note that `set` has correspondence to MySQL *ENUM* type.

- `None`
- number: `int`, `long`, `float`, `Decimal`

[1] https://bitbucket.org/saaj/mysql-simple-query-builder/src/default/myquerybuilder/test/
[2] http://dev.mysql.com/doc/sakila/en/index.html

- date: `datetime`, `date`
- string: `str`, `unicode`, `bytes`
- number or string sequence

E.g. `{'a': 'foo', 'b': (21, 9), 'c': None}` results in `WHERE (\`a\` = %(a)s AND \`b\` IN %(b)s) AND \`c\` IS %(c)s` which in turn is interpolated by the driver library.

param order `tuple` sequence of field and sort order, e.g. `[('a', 'asc'), ('b', 'desc')]`.

param limit `int` of row limit or `tuple` with offset and row limit, e.g. `10` or `(100, 10)`.

```
fields = 'film_id', 'title'
table  = 'film'
where  = {'rating': ('R', 'NC-17'), 'release_year': 2006}
order  = [('release_year', 'asc'), ('length', 'desc')]
limit  = 2

rows = qb.select(fields, table, where, order, limit)
print(rows)
# (
#   {'film_id': 872, 'title': 'SWEET BROTHERHOOD'},
#   {'film_id': 426, 'title': 'HOME PITY'},
# )

fields = 'film_id',
rows   = qb.select(fields, table, where, order, limit)
print(rows)
# (872, 426)
```

one (*fields*, *table*, *where=None*, *order=None*)

Returns first matched row's field values. When `fields` is one-element sequence, it returns the field's value, otherwise returns value is a `dict`. If no row is matched, `None` is returned.

```
fields = 'username', 'email'
table  = 'staff'
where  = {'active': True}
order  = [('last_name', 'asc')]

row = qb.one(fields, table, where, order)
print(row)
# {'username': 'Mike', 'email': 'Mike.Hillyer@sakilastaff.com'}

fields = 'username',
value  = qb.one(fields, table, where, order)
print(value)
# Mike
```

count (*table*, *where=None*)

Returns matched row count.

```
count = qb.count('payment', {'customer_id': (1, 2, 3), 'staff_id': 1}
print(count)
# 46
```

insert (*table*, *values*)
Inserts a row with the values into the table. Last inserted id is returned.

```
actor = {
  'first_name' : 'John',
  'last_name'  : 'Doe'
}
id = self.testee.insert('actor', actor)
```

update (*table*, *values*, *where*)
Updates the matched rows in the table. Affected row count is returned. If `where` is `None` it updates every row in the table.

```
values = {'title': 'My New Title', 'length': 99}
where  = {'film_id': 10}

affected = qb.update('film', values, where)
```

delete (*table*, *where*)
Deletes the matched rows from the table. Affected row count is returned. If `where` is `None` it deletes every row in the table.

```
where = {'release_year': 2000}

affected = qb.delete('film', where)
```

cursor (*type=<type 'tuple'>*)
Return a cursor instance that corresponds to the provided type. Type can be either an actual cursor class, or an alias that is looked up in *cursorType*.

quote (*value*)
Returns literal representation comprised of UTF-8 bytes, `str` for Python 2 and `bytes` with *surrogateescape* encoding for Python3, for the value. It doesn't necessarily quotes the value, when it's an `int`, or, specifically in case of `pymysql decimal.Decimal`.

query (*sql*, *where=None*, *order=None*, *limit=None*)
Executes the SQL query and returns its cursor. Returned cursor is the cursor aliased by `dict`. The method is an aid for complex query construction when its *WHERE*, *ORDER*, *LIMIT* are yet simple. If there is no clause placeholder in the query, but clause values are provided, its representation is appended to the query. If there is a placeholder, but no values, it is replaced with empty string.

```
sql = '''
  SELECT address, district
  FROM (
    SELECT ad.*
    FROM country cn
    JOIN city    ct USING(country_id)
    JOIN address ad USING(city_id)
    {where}
    {order}
 ) AS `derived`
 {limit}
'''
where = {
  'ad.address2'    : None,
  'ct.city_id'     : 300,
  'ad.address_id' : (1, 2, 3)
```

(continues on next page)

(continued from previous page)

```
}
order = [('ad.address_id', 'desc')]
limit = 10

cursor = qb.query(sql, where, order, limit)
```

ping(*reconnect=True*)

Checks connection to the server.

param reconnect Controls whether reconnection should be performed in case of lost connection.

raises OperationalError Is raised when ping has failed.

> **Warning:** When reconnection occurs, implicit rollback, lock release and other resets are performed! In worst case the server may not yet know that the client has lost connection, thus the client may find itself in a separate to its old transaction. For more details read *Persistent connection*.

begin()
Starts a transaction on the first call (in stack order), or creates a save point on a consecutive call. Increments the transaction level. Read *Nested transaction*.

commit()
Commits a transaction if it's the only pending one. Otherwise releases the savepoint. Decrements the transaction level. Read *Nested transaction*.

rollback()
Rolls back a transaction if it's the only pending one. Otherwise rolls back the savepoint. Decrements the transaction level. Read *Nested transaction*.

8.2 Cursor

class myquerybuilder.builder.**NamedCursor**(*connection*)
Default cursor type of *QueryBuilder*. It converts *named* paramstyle into *pyformat*, so it can work with MySQLdb-family as it uses client-side query parametrisation with string interpolation via `%` operator. *Named* paramstyle is easy to read and write. Though if you don't need it you can opt-put with setting desired *cursorType* mapping as *QueryBuilder* internally doesn't rely on *named* paramstyle.

```
sql = '''
  SELECT c.first_name `firstName`, c.last_name `lastName`
  FROM customer c
  JOIN store    s USING(store_id)
  JOIN staff    t ON s.manager_staff_id = t.staff_id
  WHERE c.active = :active AND t.email LIKE :email
  LIMIT 0, 1
'''
cursor = self.testee.cursor(dict)
cursor.execute(sql, {'active': True, 'email': '%@sakilastaff.com'})

print(cursor.fetchall())
# ({'firstName': u'MARY', 'lastName': u'SMITH'},)
```

execute(*query*, *args=None*)
Executes a query.

Parameters

- **query** – Query to execute on server.
- **args** – Optional sequence or mapping, parameters to use with query. If `args` is a sequence, then *format* paramstyle, `%s`, must be used in the query. If a mapping is used, then it should be either *named* or *pyformat*, `:foo` and `%(bar)s` respectively.

Returns Number of affected rows.

class myquerybuilder.builder.**NamedDictCursor**(*connection*)
The same as *NamedCursor* but with records represented by a `dict`.

8.3 Internal

class myquerybuilder.builder.**ClauseBuilder**
The class is responsible for handling SQL clause strings.

where = '{where}'
SQL *WHERE* clause placeholder to be used with *query()*.

order = '{order}'
SQL *ORDER* clause placeholder to be used with *query()*.

limit = '{limit}'
SQL *LIMIT* clause placeholder to be used with *query()*.

getReference(*name*, *alias=None*)
Returns a valid reference for a field, with optional alias, or a table, e.g.:

name →‘name‘
a.name → a.‘name‘

getPlaceholder(*name*, *nameOnly=False*, *postfix=''*)
Returns *pyformat* paramstyle placeholder, which can optionally be postfixed.

getExpressions(*fields*, *postfix=''*, *op='='*)
Returns a tuple of boolean binary expressions for provided field names, where the entry looks like *field operator placeholder*. Default operator is equality, =. Placeholder can be postfixed.

getSelectClause(*fields*)
Returns SELECT clause string with provided fields.

getFromClause(*table*)
Returns FROM clause string with provided table name.

getWhereClause(*where*)
Returns WHERE clause string with conjunction of provided conditions. Empty string when no condition is provided.

getOrderClause(*order*)
Returns ORDER clause string with provided order sequence. Empty string when no order is provided.

getLimitClause(*limit*)
Returns LIMIT clause string with provided limit. Empty string when no limit is provided. The parameter can be either entry limiting `int` or two-element sequence *(start, offset)*.

replaceClause(*sql*, *name*, *values*)
Handles the clause in the SQL query. If the query has a placeholder, *{where}* for instance, and corresponding values are provided, the placeholder is replaced with return value of the clause callable. If there's no placeholder, return value is appended to the query. If a placeholder presents, but no values are provided, it's replaced with empty string.

Parameters

- **sql** – SQL query to process.
- **name** – Clause name: *where*, *order*, *limit*.
- **values** – Values to pass to the clause callable.

Returns SQL query with processed clause.

class myquerybuilder.builder.**ClauseBuilderCamelCase**
The subclass that makes it possible to reference fields and tables with *camelCase* (also called mixed case when it starts from lowercase letter), when MySQL side, which is dominating convention, uses underscore convention and Python side uses camelCase. *QueryBuilder* can be instantiated with custom clause builder like this:

```
qb = QueryBuilder(clauseBuilder = ClauseBuilderCamelCase, **config)
```

getReference(*name*, *alias=None*)
Returns a valid reference for a field, with optional alias, or a table, e.g.:

nameToName →'name_to_name'
a.nameToName → a.'name_to_name'

CHAPTER 9

Profiling

MySQL Simple Query Builder provides profiling[1] automation and status variable[2] collection for a query or series of queries. Specifically what it does is:

1. Turn off query cache
2. Fetch status variables before the target queries
3. Wrap target queries in `SET profiling = 1` and `SET profiling = 0`
4. Fetch status variables after the target queries and calculate a difference

`myquerybuilder.profiler.QueryProfiler` only collects profiles and status changes. If you need to profile a method or a query subclass `myquerybuilder.test.ProfilerTestCase`. It is a subclass of `unittest.TestCase` and is intended to be used with compatible test runner.

Note: `myquerybuilder.profiler.QueryProfiler` overrides `cursorType` with aliases to profilable cursors. Thus if you use non-alias or `None` `type` argument for `cursor()`, the queries won't be handled.

By default, `myquerybuilder.test.ProfilerTestCase` prints only non-zero changes in status variables and performance entries which are greater than or equal to 5% of the query total execution time. You can adjust these by changing its `skipZeroStatus` and `durationThreshold` attributes, respectively. Also it prints queries, query arguments and their execution plans.

`QueryProfiler.groups` defines status groups which should be collected. By default it is `('select', 'sort', 'handler', 'created', 'innodb_buffer')`. See MySQL server status variable description[3].

Note: MySQL variable `profiling_history_size`[4] defines the number of statements for which to maintain profiling information if profiling is enabled, which by default is `15`. If you need to profile a group of more than 15 queries, you need to increase it.

---

[1] http://dev.mysql.com/doc/refman/5.5/en/show-profile.html
[2] http://dev.mysql.com/doc/refman/5.5/en/show-status.html
[3] http://dev.mysql.com/doc/refman/5.5/en/server-status-variables.html
[4] http://dev.mysql.com/doc/refman/5.5/en/server-system-variables.html#sysvar_profiling_history_size

9.1 Example

```
from myquerybuilder import test, profiler


class TestExample(test.ProfilerTestCase):

  def setUp(self):
    test.ProfilerTestCase.profiler = profiler.QueryProfiler(**config)

    super(TestQueryProfiler, self).setUp()

  @test.ProfilerTestCase.profile
  def testProfile(self):
    sql = '''
      SELECT country_id, city_id,  city
      FROM city
      JOIN country USING(country_id )
      {where}
    '''
    self.profiler.query(sql, {'country': 'Japan'})
```

The output for this test case looks like this:

```
********* profile **********
  #1 0.000717 - 100.0%
    starting        0.000072
    sending data    0.000389
    freeing items   0.000075

    *********** plan ***********
      ***************** id:1 *****************
      select_type     SIMPLE
      table           country
      type            ALL
      possible_keys   PRIMARY
      key
      key_len
      ref
      rows            109
      extra           Using where
      ***************** id:1 *****************
      select_type     SIMPLE
      table           city
      type            ref
      possible_keys   idx_fk_country_id
      key             idx_fk_country_id
      key_len         2
      ref             sakila.country.country_id
      rows            3
      extra

    ********** query ***********

      SELECT country_id, city_id,  city
      FROM city
      JOIN country USING(country_id )
```

(continues on next page)

(continued from previous page)

```
      WHERE (`country` = %(country)s)


    ******** arguments *********
    {'country': 'Japan'}


****** status handler ******
  handler_commit                        1.0
  handler_read_first                    1.0
  handler_read_key                      2.0
  handler_read_next                     31.0
  handler_read_rnd_next                 110.0
*** status innodb_buffer ***
  innodb_buffer_pool_read_requests      116.0
****** status select *******
  select_scan                           1.0
```

For more examples you can look at the profiler test module[5].

[5] https://bitbucket.org/saaj/mysql-simple-query-builder/src/default/myquerybuilder/test/profiler.py

CHAPTER 10

Mysql Simple Query Builder

Following the rule *simple easy, complex possible*, the package provides API for simple queries, nested transactions and aid for complex query building and profiling. It's a small wrapper around a Python MySQL driver.

The package is written on the following assumptions:

1. SQL is feasible and representative DSL
2. Simple SQL is simple but tedious to write by hand
3. Complex SQL is possible and should be written by hand or constructed elaborately
4. Unit/integration testing of domain logic against database is necessary
5. Database abstraction in the age of SaaS is a waste

10.1 Scope

What is simple SQL? Here are signatures for the query methods:

```
select(self, fields, table, where = None, order = None, limit = None)
one(self, fields, table, where = None, order = None)
count(self, table, where = None)
insert(self, table, values)
update(self, table, values, where)
delete(self, table, where)
```

What is complex SQL? Here are signatures of query construction:

```
cursor(self, type = tuple)
quote(self, value)
query(self, sql, where = None, order = None, limit = None)
```

Short argument description follows. For detailed description read *Reference*.

- *fields* is `str` sequence, e.g. `('film_id', 'title')`

- *table* is `str`, e.g. `'film'`
- *where* is `dict`, e.g. `{'rating': ('R', 'NC-17'), 'release_year': 2006}`
- *order* is `tuple` sequence of field and sort order, e.g. `[('release_year', 'asc'), ('length', 'desc')]`
- *limit* is `int` of row limit or `tuple` with offset and row limit, e.g. `10` or `(100, 10)`
- *values* is `dict`, e.g. `{'title': 'My New Title', 'length': 99}`

Accordingly, simple *SELECT* query is a query to single table with conjunction of conditions whose values are literal, set of fields to order by and row offset and limit. Methods for *INSERT*, *UPDATE* and *DELETE* queries are similar. Complex SQL is everything else.

That is to say the package doesn't even try to abstract away from SQL. But rather it takes away the need of writing really simple queries by hand. These method calls have obvious corresponding SQL, so obvious are their execution plans.

The library was written for web application use case when dealing with domain objects' representations is more natural order of things than classical domain simulation with domain objects per se. Thus if you work on a classical environment with long-living in-memory domain objects, taking philosophical problem of object-relational mismatch[1] serious, or just developing a shrink-wrap, unlikely you will find the library useful. If that's the case it's worth looking at SQLAlchemy[2], peewee[3] or the like.

[1] http://en.wikipedia.org/wiki/Object-relational_impedance_mismatch
[2] http://www.sqlalchemy.org/
[3] https://peewee.readthedocs.org/

Index

B

begin() (myquerybuilder.builder.QueryBuilder method), 20

C

ClauseBuilder (class in myquerybuilder.builder), 21
ClauseBuilderCamelCase (class in myquerybuilder.builder), 22
commit() (myquerybuilder.builder.QueryBuilder method), 20
count() (myquerybuilder.builder.QueryBuilder method), 18
cursor() (myquerybuilder.builder.QueryBuilder method), 19
cursorType (myquerybuilder.builder.QueryBuilder attribute), 17

D

delete() (myquerybuilder.builder.QueryBuilder method), 19

E

execute() (myquerybuilder.builder.NamedCursor method), 20

G

getExpressions() (myquerybuilder.builder.ClauseBuilder method), 21
getFromClause() (myquerybuilder.builder.ClauseBuilder method), 21
getLimitClause() (myquerybuilder.builder.ClauseBuilder method), 21
getOrderClause() (myquerybuilder.builder.ClauseBuilder method), 21
getPlaceholder() (myquerybuilder.builder.ClauseBuilder method), 21
getReference() (myquerybuilder.builder.ClauseBuilder method), 21
getReference() (myquerybuilder.builder.ClauseBuilderCamelCase method), 22
getSelectClause() (myquerybuilder.builder.ClauseBuilder method), 21
getWhereClause() (myquerybuilder.builder.ClauseBuilder method), 21

I

insert() (myquerybuilder.builder.QueryBuilder method), 18

L

limit (myquerybuilder.builder.ClauseBuilder attribute), 21

N

NamedCursor (class in myquerybuilder.builder), 20
NamedDictCursor (class in myquerybuilder.builder), 21

O

one() (myquerybuilder.builder.QueryBuilder method), 18
order (myquerybuilder.builder.ClauseBuilder attribute), 21

P

ping() (myquerybuilder.builder.QueryBuilder method), 20

Q

query() (myquerybuilder.builder.QueryBuilder method), 19
QueryBuilder (class in myquerybuilder.builder), 17
quote() (myquerybuilder.builder.QueryBuilder method), 19

R

replaceClause() (myquerybuilder.builder.ClauseBuilder method), 22

rollback() (myquerybuilder.builder.QueryBuilder method), 20

S

select() (myquerybuilder.builder.QueryBuilder method), 17

U

update() (myquerybuilder.builder.QueryBuilder method), 19

W

where (myquerybuilder.builder.ClauseBuilder attribute), 21