quma Documentation

Release 0.1.0

ebene fünf GmbH

Aug 18, 2020

Navigation

quma is a small SQL database library for **Python** and **PyPy** version 3.5 and higher. It maps object methods to SQL script files and supports **SQLite**, **PostgreSQL**, **MySQL** and **MariaDB**.

Unlike ORMs, it allows to write SQL as it was intended and to use all features the DBMS provides. As it uses plain SQL files you can fully utilize your database editor or IDE tool to author your queries.

It also provides a simple connection pool and templating for dynamic SQL like conditional WHEREs.

CHAPTER 1

Motivation

Unlike ORMs, it allows to write SQL as it was intended and to use all features the DBMS provides. As it uses plain SQL files you can fully utilize your database editor or IDE to author your queries.

If you know how to best design your DDL and already have a SELECT in your mind when data needs to be retrieved, welcome, this is for you.

It gives you back your powers you so carelessly gave away to ORMs.

CHAPTER 2

Learn more

- If you want to know how to install quma and its dependencies, see *Installation*.
- To get started read *How to use quma* from start to finish.
- To see what you can do with query objects read *The Query class*.
- In *Connecting* you learn how to connect to SQLite, PostgreSQL and MySQL/MariaDB databases.
- *Connection pool*.
- Learn what a *changling cursor* is and how it enables you to access result data in three different ways.
- Database management systems have different ways of parameter binding. *Passing parameters* shows how it works in quma.
- SQL doesn't support every kind of dynamic queries. If you reach its limits you can circumvent this by using *Templates*.
- If you pass more than one directory to the constructor, quma shadows duplicate files. See how this works in *Shadowing*.
- You can add custom methods to namespaces. Learn how to do it in *Custom namespaces*. You will also learn about aliasing.
- If you like to work on quma itself, *Testing* has the information on how to run its tests.

CHAPTER 3

License

quma is released under the MIT license.

CHAPTER 4

Installation

4.1 Prerequisites

You may need to install the Python and database library develompment headers.

On Debian/Ubuntu derivates for example like this:

```
# PostgreSQL
sudo apt install python3-dev libpq-dev
# MySQL/MariaDB
sudo apt install python3-dev default-libmysqlclient-dev
```

4.2 Installing quma

If you like to use quma with **SQLite** Python has everything covered and you only need to install quma itself:

```
pip install quma
```

To connect to a **PostgreSQL** or **MySQL/MariaDB** database you need to install the matching driver:

```
# PostgreSQL
pip install quma psycopg2
# or
pip install quma psycopg2cffi


# MySQL/MariaDB
pip install quma mysqlclient
```

4.3 Templates for dynamic SQL

You need to install the Mako template library if you want to use dynamic sql scripts using *templates*.

```
pip install mako
```

4.4 Development

```
git clone https://github.com/ebenefuenf/quma
cd quma
pip install -e '.[test,docs,templates,postgres,mysql]'
```

CHAPTER 5

How to use quma

quma reads sql files from the file system and makes them accessible as script objects. It passes the content of the files to a connected database management system (DBMS) when these objects are called.

Throughout this document we assume a directory with the following structure:

```
Scripts                  |  Content
-------------------------|-------------------------------------------
path/to/sql/scripts      |
├─ users                 |
│   ├─ all.sql           |  SELECT * FROM users
│   ├─ by_city.sql       |  SELECT * FROM users WHERE city = :city
│   ├─ by_id.sql         |  SELECT * FROM users WHERE id = :id
│   ├─ remove.sql        |  DELETE FROM users WHERE id = :id
│   └─ rename.sql        |  UPDATE TABLE users
│                        |      SET name = :name WHERE id = :id
└─ get_admin.sql         |  SELECT * FROM users WHERE admin = 1
```

After initialization you can run these scripts by calling members of a `Database` or a `Cursor` instance. Using the example above the following members are available:

```
# 'cur' is a cursor instance
cur.users.all(...
cur.users.by_city(...
cur.users.by_id(...
cur.users.rename(...
cur.users.remove(...
cur.get_admin(...
```

Read on to see how this works.

5.1 Opening a connection

To connect to a DBMS you need to instantiate an object of the `Database` class and provide a connection string and either a single path or a list of paths to your SQL scripts.

```
from quma import Database
db = Database('sqlite:///:memory:', '/path/to/sql-scripts')
```

Note: `Database` instances are threadsafe.

For more connection examples (e. g. PostgreSQL or MySQL/MariaDB) and parameters see *Connecting*. quma also supports *connection pooling*.

From now on we assume an established connection in the examples.

5.2 Creating a cursor

DBAPI libs like *psycopg2* or *sqlite3* have the notion of a cursor, which is used to manage the context of a fetch operation. quma is similar in that way. To execute queries you need to create a cursor instance.

quma provides two ways to create a cursor object. Either by using a context manager:

```
with db.cursor as cur:
    ...
```

Or by calling the `cursor` method of the `Database` instance:

```
try:
    cur = db.cursor()
finally:
    cur.close()
```

5.3 Running queries

To run the query in a sql script from the path(s) you passed to the `Database` constructor you call members of the Database instance or the cursor (*db* and *cur* from now on).

Scripts and directories at the root of the path are translated to direct members of *db* or *cur*. After initialisation of our example dir above, the script `/get_admin.sql` is available as `Script` instance `db.get_admin` or `cur.get_admin` and the directory `/users` as instance of `Namespace`, i. e. `db.users` or `cur.users`. Scripts in subfolders will create script objects as members of the corresponding namespace: `/users/all` will be `db.users.all` or `cur.users.all`.

When you call a `Script` object, as in `cur.user.all()` where `all` is the mentioned object, you get back a `Query` instance. The simplest use is to iterate over it (see below for more information about the `Query` class):

```
with db.cursor as cur:
    all_users = cur.users.all()
    for user in all_users:
        print(user['name'])
```

The same using the *db* API:

```
with db.cursor as cur:
    all_users = db.users.all(cur)
```

To learn what you can do with `Query` objects see *The Query class*.

Note: As you can see *cur* provides a nicer API where you don't have to pass the cursor when you call a script or a method. Then again the *db* API has the advantage of being around 30% faster. But this should only be noticeable if you run hundreds or thousands of queries in a row for example in a loop.

If you have cloned the quma repository from github you can see the difference when you run the script `bin/cursor_vs_db.py`.

5.4 Committing changes and rollback

quma does not automatically commit by default. You have to manually commit all changes as well as rolling back if an error occurs using the `commit()` and `rollback()` methods of the cursor.

```
try:
    cur.users.remove(id=13).run()
    cur.users.rename(id=14, name='New Name').run()
    cur.commit()
except Exception:
    cur.rollback()
```

If *db* is initialized with the flag `contextcommit` set to `True` and a context manager is used, quma will automatically commit when the context manager ends. So you don't need to call `cur.commit()`.

```
db = Database('sqlite:///:memory:', contextcommit=True)

with db.cursor as cur:
    cur.users.remove(id=13).run()
    cur.users.rename(id=14, name='New Name').run()
    # no need to call cur.commit()
```

Note: If you are using MySQL or SQLite some statements will automatically cause a commit. See the MySQL docs and SQLite docs

5.4.1 Autocommit

If you pass `autocommit=True` when you initialize a cursor, each query will be executed in its own transaction that is implicitly committed.

```
with db(autocommit=True).cursor as cur:
    cur.users.remove(id=13).run()
```

```
try:
    cur = db.cursor(autocommit=True)
    cur.users.remove(id=13).run()
finally:
    cur.close()
```

5.5 Executing literal statements

`Database` instances provide the method `execute()`. You can pass arbitrary sql strings. Each call will be automatically committed. If there is a result it will be returned otherwise it returns `None`.

```
db.execute('CREATE TABLE users ...')
users = db.execute('SELECT * FROM users')
for user in users:
    print(user.name)
```

If you want to execute statements in the context of a transaction use the `execute()` method of the cursor:

```
with db.cursor as cur:
    cur.execute('DELETE FROM users WHERE id = 13');
    cur.commit()
```

5.6 Accessing the DBAPI cursor and connection

The underlying DBAPI connection and cursor objects are available as members of the cursor instance. The connection object as `raw_conn` and the cursor as `raw_cursor.cursor`.

```
# The connection
cur.raw_conn.autocommit = True
dbapi_cursor = cur.raw_conn.cursor()

# The cursor
cur.raw_cursor.cursor.execute('SELECT * FROM users;')
users = cur.raw_cursor.cursor.fetchall()
# raw_cursor wraps the real cursor. This would work as well
cur.raw_cursor.execute('SELECT * FROM users;')
users = cur.raw_cursor.fetchall()
```

All members of the `raw_cursor.cursor` object are also available as members of *cur*. Hence there should be no need to use it directly:

```
cur.execute('SELECT * FROM users;')
users = cur.fetchall()
```

CHAPTER 6

The Query class

When you call a script object it returns an instance of the `Query` class which holds the code from the script file and the parameters passed to the script call.

Queries are executed lazily. This means you have to either call a method of the query object or iterate over it to cause the execution of the query against the DBMS.

6.1 Getting multiple rows from a query

You can either iterate directly over the query object or call its `all()` method to get a list of all the rows.

```
with db.cursor as cur:
    result = cur.users.by_city(city='City 1')
    for row in result:
        print(row.name)

    # calling the .all() method to get a materialized list/tuple
    user_list = cur.users.by_city(city='City 1').all()
    # is a bit faster than
    user_list = list(cur.users.by_city(city='City 1'))
```

When calling `all()` **MySQL** and **MariaDB** will return a tuple, **PostgreSQL** and **SQLite** will return a list.

Note: If you are using **PyPy** with the *sqlite3* driver the cast using the `list()` function does not currently work and will always result in an empty list.

6.2 Getting a single row

If you know there will be only one row in the result of a query you can use the `one()` method to get it. quma will raise a `DoesNotExistError` error if there is no row in the result and a `MultipleRowsError` if there is more

than one row.

```
from quma import (
    DoesNotExistError,
    MultipleRowsError,
)
...

with db.cursor as cur:
    try:
        user = cur.users.by_id(id=13).one()
    except DoesNotExistError:
        print('The user does not exist')
    except MultipleRowsError:
        print('There are multiple users with the same id')
```

`DoesNotExistError` and `MultipleRowsError` are also attached to the `Database` class so you can access it from the *db* instance. For example:

```
with db.cursor as cur:
    try:
        user = cur.users.by_id(id=13).one()
    except db.DoesNotExistError:
        print('The user does not exist')
    except db.MultipleRowsError:
        print('There are multiple users with the same id')
```

It is also possible to get a single row by accessing its index on the result set:

```
user = cur.users.by_id(id=13)[0]
# or
users = cur.users.by_id(id=13)
user = users[0]
```

If you want the first row of a result set which may have more than one row or none at all you can use the `first()` method:

```
# "user" will be None if there are no rows in the result.
user = cur.users.all().first()
```

The method `value()` invokes the `one()` method, and upon success returns the value of the first column of the row (i. e. `fetchall()[0][0]`). This comes in handy if you are using a `RETURNING` clause, for example, or return the last inserted id after an insert.

```
last_inserted_id = cur.users.insert().value()
```

6.3 Execute only

To simply execute a query without needing its result you call the `run()` method:

```
with db.cursor as cur:
    cur.user.add(name='User',
                 email='user@example.com',
                 city='City').run()
```

(continues on next page)

(continued from previous page)

```
    # or
    query = cur.user.add(name='User',
                         email='user@example.com',
                         city='City')
    query.run()
```

This is handy if you only want to execute the query, e. g. DML statements like `INSERT`, `UPDATE` or `DELETE` where you don't need a fetch call.

6.4 Getting data in chunks

quma supports the `fetchmany` method of Python's DBAPI by providing the `many()` method of `Query`. `many()` returns an instance of `ManyResult` which implements the `get()` method which internally calls the `fetchmany` method of the underlying cursor.

```
many_users = cur.users.by_city(city='City').many()
first_two = manyusers.get(2) # the first call of get executes the query
next_three = manyusers.get(3)
next_two = manyusers.get(2)
```

Another example:

```
def users_generator()
    with db.cursor as cur:
        many_users = cur.users.all().many()
        batch = many_users.get(3) # the first call of get executes the query
        while batch:
            for result in batch:
                yield result
            batch = many_users.get(3)

for user in users_generator():
    print(user.name)
```

Note: In contrast to all other fetching methods of the query object, like `all()`, `first()`, or `one()`, a call of `many()` will not execute the query. Instead, the first call of the `get()` method of a *many* result object will cause the execution. Also, results of *many* calls are not cached and if a query was already executed the *many* mechanism will execute it again anyway. So keep in mind that already executed queries will be re-executed when `many()` is called after the first execution, as in:

```
all_users = cur.users.all()
first_user = allusers.first() # query executed the first time
many_users = allusers.many()
first_two = manyusers.get(2) # query executed a second time
```

Additionally, the cache of `all_users` from the last example will be invalidated after the first call of `get()`. So you should avoid to mix *many* queries with "normal" queries.

6.5 A simpler version of `many()`

If your expected result set is too large for simply iterating over the query object or calling `all()` (as they call `fetchall` internally) but you like to work with the result in a single simple loop instead of using `many()`, you can use the method `unbunch()`. It is a convenience method which internally calls `fetchmany` with the given size. Using `unbunch()` we can simplify the `many()` example with the `users_generator` from the last section:

```
with db.cursor as cur:
    for user in cur.users.all().unbunch(3):
        print(user.name)
```

`unbunch()` re-excutes the query and invalidates the cache on each call, just like `many()`.

6.6 Getting the number of rows

If you are only interested in the number of row in a result you can pass a `Query` object to the `len()` function. quma also includes a convenience method called `count()`. Some drivers (like pycopg2) support the `rowcount` property of PEP249 which specifies the number of rows that the last execute produced. If it is available it will be used to determine the number of rows, otherwise a fetchall will be executed and passed to `len()` to get the number.

```
number_of_users = len(cur.users.all())
number_of_users = cur.users.all().count()
number_of_users = db.users.all(cur).count()
```

Note: `len()` or `count()` calls must occure before fetch calls like `one()` or `all()`. This has to do with the internals of the DBAPI drivers. A fetch would overwrite the value of `rowcount` which would return `-1` afterwards.

6.7 Checking if a result exists

To check if a query has a result or not call the `exists()` method.

```
has_users = cur.users.all().exists()
```

You can also use the query object itself for truth value testing:

```
all_users = cur.users.all()
if all_users:
    user1 = allusers.first()
```

6.8 Ad hoc queries

To run an ad hoc query you can use the `query` method of the cursor:

```
with db.cursor as cur:
    sql = 'SELECT name, city FROM users WHERE email = ?;'
    user = cur.query(sql, 'user.1@example.com').one()
```

6.9 Prepared statements

quma does not have a special API for prepared statements, but you can easily use them. In the following example we use PostgreSQL's syntax. Given a SQL script `sqlscripts/users/prepare.sql` with the content below ...

```
PREPARE prep (varchar(128), int) AS
SELECT name, city FROM users WHERE email = $1 AND 1 = $2;
```

... you can use it as shown here:

```
with db.cursor as cur:
    cur.users.pgsql_prepare().run()
    for i in range(1, 5):
        q = cur.query(f"EXECUTE prep('user.{i}@example.com', 1);")
        assert q.value() == f'User {i}'
    cur.query(f"DEALLOCATE PREPARE prep;").run()
```

6.10 Results are cached

As described above, quma executes queries lazily. Only after the first call of a method or when an iteration over the query object is started, the data will be fetched. The fetched result will be cached in the query object. This means you can perform more than one operation on the object while the query will not be re-executed. If you want to re-execute it, you need to call `run()` manually.

```
with db.cursor as cur:
    all_users = cur.users.all()

    for user in all_users:
        # the result is fetched and cached on the first iteration
        print(user.name)

    # get a list of all users from the cache
    all_users.all()
    # get the first user from the cache
    all_users.first()

    # re-execute the query
    all_users.run()

    # fetch and cache the new result of the re-executed query
    all_users.all()
```

6.11 Accessing the underlying cursor

You can access the attributes of the cursor which is used to execute the query directly on the query object.

```
with db.cursor as cur:
    added = cur.users.add(name='User', email='user.1@example.com').run()
    if added.lastrowid:
        user = cur.user.by_id(id=added.lastrowid).run()
        user.fetchone()
```

6.12 Overview

6.12.1 Class Query

class quma.query.**Query**(*script*, *cursor*, *args*, *kwargs*, *prepare_params*)

The query object is the value you get when you run a query, i. e. call a `Script` object.

all()
Return a list of all results

count()
Return the length of the result.

exists()
Return if the query's result has rows.

first()
Get exactly one row and return None if there is no row present in the result.

many()
Return a ManyResult object initialized with this query object.

one()
Get exactly one row and check if only one exists, otherwise raise an error.

run()
Execute the query using the DBAPI driver.

unbunch(*size=None*)
Return a generator that simplifies the use of fetchmany.

Parameters **size** – The number of rows to be fetched per fetchmany call. If not given use the default value of the driver.

value(*key=0*)
Call *one()* and return the first column by default.

Parameters **key** – Return the value at `key`'s position instead of the first column.

6.12.2 Class ManyResult

class quma.query.**ManyResult**(*query*)

get(*size=None*)
Call the `fetchmany()` method of the raw cursor.

Parameters **size** – The number of rows to be returned. If not given use the default value of the driver.

CHAPTER 7

Connecting

You connect to a server/database by creating an instance of the class `quma.Database`. You have to at least provide a valid database URL and the path to your sql scripts. See below for the details.

7.1 Connection Examples

```
sqldirs = '/path/to/sql/scripts'

# can also be a list of paths:
sqldirs = [
  '/path/to/sql/scripts',
  '/another/path/to/sql/scripts',
]

# SQLite
db = Database('sqlite:////path/to/db.sqlite', sqldirs)
# SQLite in memory db
db = Database('sqlite:///:memory:', sqldirs)

# PostgreSQL localhost
db = Database('postgresql://username:password@/db_name', sqldirs)
# PostgreSQL network server
db = Database('postgresql://username:password@10.0.0.1:5432/db_name', sqldirs)

# MySQL/MariaDB localhost
db = Database('mysql://username:password@/db_name', sqldirs)
# MySQL/MariaDB network server
db = Database('mysql://username:password@192.168.1.1:5432/db_name', sqldirs)

# You can pass driver specific parameters e. g. MySQL's charset
db = Database('mysql://username:password@192.168.1.1:5432/db_name',
              sqldirs,
              charset='utf8')
```

7.2 The Database class

class quma.database.**Database**(*dburi*, **args*, ***kwargs*)

The database object acts as the central object of the library.

Parameters

- **dburi** – The connection string. See section "Connection Examples"
- **sqldirs** – One or more filesystem paths pointing to the sql scripts. `str` or `pathlib.Path`.
- **persist** – If `True` quma immediately opens a connection and keeps it open throughout the complete application runtime. Setting it to `True` will raise an error if you try to initialize a connection pool. Defaults to `False`.
- **pessimistic** – If `True` quma emits a test statement on a persistent SQL connection every time it is accessed or at the start of each connection pool checkout (see section "Connection Pool"), to test that the database connection is still viable. Defaults to `False`.
- **contextcommit** – If `True` and a context manager is used quma will automatically commit all changes when the context manager exits. Defaults to `False`.
- **prepare_params** – A callback function which will be called before every query to prepare the params which will be passed to the query. Defaults to None.
- **file_ext** – The file extension of sql files. Defaults to '`sql`'.
- **tmpl_ext** – The file extension of template files (see *Templates*). Defaults to `'msql'`.
- **echo** – Print the executed query to stdout if `True`. Defaults to `False`. *PostgreSQL* and *MySQL/MariaDB* connections will print the query after argument binding. This means placeholders will be substituted with the parameter values. *SQLite* will always print the query without substitutions.
- **cache** – cache the scripts in memory if `True`, otherwise re-read each script when the query is executed. Defaults to `False`.

Additional connection pool parameters (see *Connection pool*):

Parameters

- **size** – The size of the pool to be maintained. This is the largest number of connections that will be kept persistently in the pool. The pool begins with no connections. Defaults to 5.
- **overflow** – The maximum overflow size of the pool. When the number of checked-out connections reaches the size set in *size*, additional connections will be returned up to this limit. Set to -1 to indicate no overflow limit. Defaults to 10.
- **timeout** – The number of seconds to wait before giving up on returning a connection. Defaults to None.

exception DoesNotExistError

exception MultipleRowsError

close()

Close (all) open connections. If you want to reconnect you need to create a new `quma.Database` instance.

cursor

Open a connection and return a cursor.

execute(*query*, ***kwargs*)
Execute the statements in `query` and commit immediately.

Parameters **query** – The sql query to execute.

release(*carrier*)
If the `carrier` holds a connection close it or return it to the pool.

Parameters **carrier** – An object holding a quma connection. See *Reusing connections*

CHAPTER 8

Connection pool

quma provides a connection pool implementation (PostgreSQL and MySQL only) similar to sqlalchemy's and even borrows code and ideas from it.

Setup a pool:

```
# PostgreSQL pool (keeps 5 connections open and allows 10 more)
db = Database('postgresql+pool://username:password@/db_name', sqldir,
              size=5, overflow=10)

# MySQL/MariaDB pool
db = Database('mysql+pool://username:password@/db_name', sqldir,
              size=5, overflow=10)
```

For a description of the parameters see *Connecting*.

CHAPTER 9

Reusing connections

To reuse connections you can pass an object - the so-called carrier - to *db* when you create a cursor. quma then remembers the carrier object's identity and and returns the same connection which was returned the first time when you pass the same carrier again.

A good example is the request object in web applications:

```
from pyramid.view import view_config
from quma import Database

db = Database('sqlite:////path/to/db.sqlite', sqldir)


def do_more(request, user_id):
    # reuses the same connection which was opened
    # in user_view.
    with db(request).cursor as cur:
        cur.user.remove(id=user_id)


@view_config(route_name='user')
def user_view(request):
    with db(request).cursor as cur:
        user = cur.user.by_name(name='Username').one()

    do_more(request, user['id'])

    with db(request).cursor as cur:
        # reuses the connection
        user = cur.user.rename(id=13, name='New Username')
        # commit every statement previously executed
        cur.commit()

        # either exlicitly close the cursor as last step
        cur.close()
```

(continues on next page)

(continued from previous page)

```
    # or release the carrier using the database object
    db.release(request)
```

Note: It is always a good idea to close a connection if you're done. If you are using a carrier and a *connection pool* it is absolutely necessary and you have to explicitly close the cursor or release the carrier. You can do it using `cur.close()` or by passing the carrier to `db.release(carrier)`, otherwise the connection would not be returned to the pool.

CHAPTER 10

Changling cursor

If you are using **SQLite** or **PostgreSQL** you can access result object attributes using three different methods if you pass `changling=True` on *db* initialization. (**MySQL** does not support it. See below)

```
db = Database('sqlite:///:memory:', sqldir, changeling=True)

with db.cursor as c:
    user = db.users.by_id(c, 13).one()
    name = user[0]        # by index
    name = user['name']   # by key
    name = user.name      # by attribute
```

10.1 Shadowed superclass members

If a query result has a field with the same name as a member of the superclass of the changeling (*sqlite*: `sqlite3.Row`, *psycopg2*: `psycopg2.extras.DictRow`) it shadows the original member. This means the original member isn't accessible. You can access it anyway if you prefix it with an underscore '_'.

The `sqlite3.Row`, for example, has a method `keys()` which lists all field names. If a query returns a field with the name 'keys' the method is shadowed:

```
-- /path/to/sql/scripts/users/by_id.sql
SELECT name, email, 'the keys' AS keys FROM users WHERE id = :id;
```

```
row = cur.users.by_id(13).one()
assert row.keys == 'the keys'

# If you want to call the keys method of row prefix it with _
print(row._keys()) # ['name', 'email', 'keys']
```

10.2 Performance

By default, changling is *False* which is slightly faster. Then SQLite supports access by index only. PostgreSQL by key and index (we use `psycopg.extras.DictCursor` internally).

10.3 MySQL/MariaDB

MySQL/MariaDB supports access by index only, except you pass `dict_cursor=True` on initialization. Then it supports access by key only.

CHAPTER 11

Passing Parameters to SQL Queries

SQLite supports two kinds of placeholders: question marks (*qmark* style) and named placeholders (named style). PostgreSQL/MySQL/MariaDB support simple (*%s*) and named (*%(name)s*) *pyformat* placeholders:

```
-- SQLite qmark
SELECT name, email FROM users WHERE id = ?
-- named
SELECT name, email FROM users WHERE id = :id

-- PostgreSQL/MySQL/MariaDB pyformat
SELECT name, email FROM users WHERE id = %s
-- named
SELECT name, email FROM users WHERE id = %(id)s
```

```
# simple style (? or %s)
cur.users.by_id(1)
db.users.by_id(cur, 1)

# named style (:name or %(name)s)
cur.users.by_id(id=1)
db.users.by_id(cur, id=1)
```

CHAPTER 12

Dynamic SQL using Templates

quma supports rendering templates using the Mako template library. By default, template files must have the file extension `*.msql`, which can be changed.

Using this feature you are able to write dynamic queries which would not be possible with SQL alone.

A very simple example:

```
-- sql/users/by_group.msql
SELECT
    name,
% if admin:
    birthday,
% endif
    city
FROM users
% if not admin:
WHERE
    group = 'public'
% endif
```

In Python you call it the same way like any other SQL query:

```
cur.users.by_group(admin=True)
```

12.1 Beware of SQL injections!

Never use templates to do a form of string concatenation as this would open the door to SQL injections. So never write queries like so:

You should always use the parameter substitution mechanism of the underlying driver and restrict Mako features to control structures:

```
SELECT * FROM
% if table_name == 'admins':
    admins
% else:
    users
% endif
WHERE
    is_active = %(is_active)s;
```

See:

- https://en.wikipedia.org/wiki/SQL_injection
- https://xkcd.com/327/ (You've seen this far too often? https://xkcd.com/1053/)

12.2 The problem with the %

The Mako template engine uses the %-sign to indicate control structures like `if` and `for`. Unfortunately *psycopg2* as well as *mysqlclient* use `%s` for query placeholders and the `%(variable)s` syntax for named placeholders. Mako does not allow the %-sign to be the first non whitespace character in a line. As per documentation Mako should allow to escape `%` using `%%`, but it seems it does not work. So you should simply avoid it in template scripts.

Wrong:

```
SELECT * FROM
    users
WHERE
    %(is_active)s = is_active;
```

Correct:

```
SELECT * FROM
    users
WHERE
    is_active = %(is_active)s;
```

See:

- https://docs.makotemplates.org/en/latest/syntax.html#control-structures
- https://github.com/sqlalchemy/mako/issues/323

12.3 Template files lookup

The resolution of included or imported template files is accomplished by mako's class `TemplateLookup`, which you can learn more about in the mako docs: Using TemplateLookup

It is initialized with the the same sql directories which are used on `Database` initialization.

CHAPTER 13

Shadowing

If you pass a list of two or more directories to the `Database` constructor, the order is important. Files from subsequent directories in the list with the same relative path will shadow (or overwrite) files from preceding directories.

Let's say you have two different directories with SQL scripts you like to use with quma. For example directory *first*:

```
/path/to/sql/scripts/first
├── addresses
│   ├── all.sql
│   └── remove.sql
├── users
│   ├── all.sql
│   └── remove.sql
├── get_admin.sql
└── remove_admin.sql
```

and directory *second*:

```
/path/to/sql/scripts/second
├── users
│   ├── all.sql
│   └── rename.sql
├── create_admin.sql
└── get_admin.sql
```

When you initialize quma with both directories like this:

```
from quma import Database

db = Database('sqlite:///:memory:', [
    '/path/to/sql/scripts/first',
    '/path/to/sql/scripts/second',
])
```

quma creates the following members:

```
                          # From directory:
cur.addresses.all         #   first
cur.addresses.remove      #   first
cur.users.all             #   second (shadows all.sql from dir first)
cur.users.remove          #   first
cur.users.rename          #   second
cur.create_admin          #   second
cur.get_admin             #   second (shadows get_admin.sql from dir first)
cur.remove_admin          #   first
```

CHAPTER 14

Custom namespaces

quma automatically creates namespace objects when it reads in your sql scripts. Each subfolder in the script directory will result in a namespace object as a direct member of *db* or *cur*.

You can add custom methods to these objects by putting a `__init__.py` file into the subfolder which is your namespace and by adding a subclass of `quma.Namespace` to it. The class must have the same name as the folder with the first letter uppercase.

```
path/to/sql/scripts
├─ users
│    ├─ __init__.py
│    ├─ all.sql
│    ├─ by_city.sql
..
```

```
from quma import Namespace

# If the subfolder's name is 'users' the
# class must be named 'Users'.
class Users(Namespace):
    # the method must accept the cursor as its first parameter
    def get_test(self, cur):
        return 'Test'

    def get_address(self, cur, username):
        user = cur.users.by_username(username=username)
        return cur.address.by_user(user.id)
```

Public methods of the namespace **must** be definied with the cursor as second parameter. It will automatically be passed when you use the *cur* api.

Now you can call the method the same way as you would call scripts:

```
db.users.get_test(cur)
cur.users.get_test() # no need to pass cur
address = cur.users.get_address('username')
```

14.1 Root members

If you want to add root level methods you need to add `__init__.py` to the root of your script directory and name the subclass *Root*.

```
path/to/sql/scripts
├─ __init__.py
├─ users
│   ├─ all.sql
..
```

```
class Root(Namespace):
    def root_method(self, cursor):
        return 'Test'
```

```
db.root_method()
cur.root_method()
```

14.2 Aliasing

If you add the class level attribute `alias` to your custom namespace, you can call it by that name too.

```
from quma import Namespace

class Users(Namespace):
    alias = 'user'
```

```
cur.user.all()
# This is the same as.
cur.users.all()
```

CHAPTER 15

Importable database

Sometimes it isn't enough to create a global `Database` instance and import it into other modules. For example, if you read the database credentials from a configuration file at runtime and then initialize the instance while the uninitialized global is already imported elsewhere. The following code shows a way to keep the quma API in place and allows to import the `db` wrapper class even if the connection is not established yet.

```
#####  my_db_module.py

import quma

_db = None

class MetaDB(type):
    def __getattr__(cls, attr):
        return getattr(_db, attr)

class db(object, metaclass=MetaDB):
    def __init__(self, carrier=None, autocommit=None):
        self.carrier = carrier
        self.autocommit = autocommit

    def __getattr__(self, attr):
        return getattr(_db(carrier=self.carrier,
                           autocommit=self.autocommit), attr)

def connect(uri):
    global _db
    sqldir = '/path/to/sql/scripts'

    _db = quma.Database(uri, sqldir)
```

Create the instance in your main module:

```
#####  main.py
```

(continues on next page)

(continued from previous page)

```
import my_db_module

my_db_module.connect('sqlite:///:memory:')
```

Now you can import the class `my_db_module.db` from everywhere and use it the same way as a usual instance of `quma.Database`.

```
#####  e. g. model.py

from my_db_module import db

with db.cursor as cur:
    cur.users.all()
```

CHAPTER 16

Testing

16.1 Prerequisites

In order to run the tests for *MySQL* or *PostgreSQL* you need to create a test database:

PostgreSQL:

```
CREATE USER quma_test_user WITH PASSWORD 'quma_test_password';
CREATE DATABASE quma_test_db;
GRANT ALL PRIVILEGES ON DATABASE quma_test_db to quma_test_user;
```

MySQL/MariaDB:

```
CREATE DATABASE quma_test_db CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
CREATE USER quma_test_user@localhost IDENTIFIED BY 'quma_test_password';
GRANT ALL ON quma_test_db.* TO quma_test_user@localhost;
```

16.2 How to run the tests

Run `pytest` or `py.test` to run all tests. `pytest -m "not postgres and not mysql"` for all general tests. And `pytest -m "postgres"` or `pytest -m "mysql"` for DBMS specific tests.

16.3 Overwrite credentials

If you like to use your own test database and user you can overwrite the default credentials by setting environment variables

PostgreSQL:

- `QUMA_PGSQL_USER`

- QUMA_PGSQL_PASS
- QUMA_PGSQL_DB

MySQL/MariaDB:

- QUMA_MYSQL_USER
- QUMA_MYSQL_PASS
- QUMA_MYSQL_DB

Index