MonSQL Documentation

Release 0.1.5

firstprayer

January 07, 2015

Contents

MongoDB style of using Relational Database

Examples

```
>>> from monsql import MonSQL, DB_TYPES
>>> monsql = MonSQL(host, port, username, password, dbname, DB_TYPES.MYSQL)
>>> user_table = monsql.get('user')
>>> activated_users = user_table.find({'state': 2})
>>> user_ids = user_table.insert([{'username': ...}, {'username': ...}, ...])
>>> user_table.commit() # OR monsql.commit()
```

`monsql.MonSQL`(*host=None*, *port=None*, *username=None*, *password=None*, *dbname=None*, *dbpath=None*, *dbtype=None*)

Initialize and return a Database instance

Contents:

CHAPTER 1

API reference for MonSQL

There're several basic concepts in MonSQL:

The MonSQL API is organised as follows:

Database: The `Database` manages operations such as create or drop tables, truncate table, return a table object. It has several subclasses implemented for different databases. A Database object should always be obtained through :py:function:'MonSQL':

```
>>> db = MonSQL(host='localhost', port=3306, dbtype=DB_TYPES.MYSQL)
```

Table: The `monsql.table.Table` class is the main class for interacting with data in tables. This class offers methods for data retrieval and data manipulation. Instances of this class can be obtained using the `Database.get()` method.

QuerySet: The `QuerySet` class is a lazy-load object for representing data. A QuerySet object can turn into a new QuerySet object by adding filter, setting limits, etc. Read all rows in a query set is simple, just:

```
>>> queryset = table.find(...)
>>> for row in queryset:
>>>     # do something
```

DataRow: A wrapper class for more readable codes. It works like this:

```
>>> queryset = table.find(filter={'id': {'$gt': 10}}, fields=['id', 'name'])
>>> for row in queryset:
>>>     print row.id, print row.name
```

1.1 Database

class `monsql.db.`**`Database`**(*db*, *mode='manual'*)

Database wrapper for interaction with specific database

`close`()

Close the connection to the server

`commit`()

Commit the current session

`create_table`(*tablename*, *columns*, *primary_key=None*, *force_recreate=False*)

Parameters

- tablename: string

- columns: list or tuples, with each element be a string like 'id INT NOT NULL UNIQUE'
- primary_key: list or tuples, with elements be the column names
- force_recreate: When table of the same name already exists, if this is True, drop that table; if False, raise exception

Return Nothing

drop_table(*tablename*, *silent=False*)

Drop a table

Parameters

- tablename: string
- slient: boolean. If false and the table doesn't exists an exception will be raised; Otherwise it will be ignored

Return Nothing

get(*name*)

Return a Table object to perform operations on this table.

Note that all tables returned by the samle Database instance shared the same connection.

Parameters

- name: A table name

Returns A Table object

get_table_obj(*name*)

This is used internally inside the class Implemented by subclasses, because different database may use different table class

is_table_existed(*tablename*)

Check whether the given table name exists in this database. Return boolean.

list_tables()

Return a list of lower case table names. Different databases have their own ways to do this, so leave the implementation to the subclasses

raw(*sql*)

Execute raw sql :Parameters:

- sql: string, sql to be executed

Return the result of this execution

If it's a select, return a list with each element be a DataRow instance

Otherwise return raw result from the cursor (Should be insert or update or delete)

set_foreign_key_check(*to_check*)

Enable/disable foreign key check. Disabling this is especially useful when deleting from a table with foreign key pointing to itself

truncate_table(*tablename*)
Delete all rows in a table. Not all databases support built-in truncate, so implementation is left to subclasses. For those don't support truncate, 'delete from ...' is used

1.2 Table

class monsql.table.**Table**(*db*, *name*, *mode=None*)
A collection of related data entries in columns and rows. This class should not be directly constructed. Instead use Database.get('table_name') to have a table returned.

columns
The columns property.

commit()
Ends current transaction, making permanent any changes made.

count(*query=None*, *distinct=False*, *distinct_fields=None*)
Returns the number of rows satisying a criteria, if provided.

Parameters

- query: specify the WHERE clause
- distinct : boolean, whether to use DISTINCT()
- distinct_fields : list or tuple or strings. Each string is a column name used inside COUNT(). If none, '*' will be used.

Return int, the number of rows

find(*filter={}*, *fields=None*, *skip=0*, *limit=None*, *sort=None*)
Searches the table using the filters provided.

Examples

```
>>> users = user_table.find({'id': {'$in': [10, 20]}, 'age': {'$gt': 20}}) # Complex query
>>> user_count = len(users)
>>> for user in users:
>>>     # Do something...
>>>     print user.id
>>>
>>> users = user_table.find({}, sort=[('age', monsql.ASCENDING)]) # sort by age
```

Also support complex operators:

```
>>> {a: 1}                                  # a == 1
>>> {a: {$gt: 1}}                           # a > 1
>>> {a: {$gte: 1}}                          # a >= 1
>>> {a: {$lt: 1}}                           # a < 1
>>> {a: {$lte: 1}}                          # a <= 1
>>> {a: {$eq: 1}}                           # a == 1
>>> {a: {$in: [1, 2]}}                      # a == 1 or a == 2
>>> {a: {$contains: '123'}}                 # a like %123%
>>> {$not: condition}                       # !(condition)
>>> {$and: [condition1, condition2, ...]}   # condition1 and condition2
>>> {$or: [condition1, condition2, ...]}    # condition1 or condition2
```

Parameters

- query(dict): specify the WHERE clause. One example is {"name": "...", "id": ...}
- fields: specify what fields are needed
- skip, limit: both integers, skip without defining limit is meaningless
- sort: A list, each element is a two-item tuple, with the first item be the column name and the second item be either monsql.ASCENDING or monsql.DESCENDING

Return a QuerySet object

find_one (*filter=None*, *fields=None*, *skip=0*, *sort=None*)
Similar to find. This method will only retrieve one row. If no row matches, returns None

insert (*data_or_list_of_data*)
Insert data into the table.

Examples

```
>>> user_table.insert({'username': 'Jude'}) # Insert one row
>>> user_table.insert([{'username': 'Andy'}, {'username': 'Julia'}, ...]) # Insert multiple
```

Parameters

- data_or_list_of_data: Either a dict or a list of dict

Return id or list of ids of inserted row

remove (*filter=None*)
Removes rows from the table.

Parameters

- query(dict), specify the WHERE clause

Return Number of rows deleted

update (*query*, *attributes*, *upsert=False*)
Updates data in the table.

Parameters

- query(dict), specify the WHERE clause
- attributes(dict), specify the SET clause
- upsert: boolean. If True, then when there's no row matches the query, insert the values

Return Number of rows updated or inserted

1.3 QuerySet

class monsql.queryset.**QuerySet** (*cursor*, *query*)
Lazy load data

distinct ()
Only return distinct row. Return a new query set with distinct mark

filter (*filter*)
Add new filter to the query set. Note that since QuerySet is lazy, it would just combine this filter with the original filter with an 'AND' :Parameters: - filter: a dictionary :Return: a new QuerySet object

limit (*n*, *skip=None*)
Limit the result set. However when the query set already has limit field before, this would raise an exception :Parameters: - n : The maximum number of rows returned - skip: how many rows to skip :Return: a new QuerySet object so we can chain operations

1.4 DataRow

class monsql.queryset.**DataRow** (*keyvalue_map*)

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*
- *search*

m