PyDbLite Documentation

Release 3.0.2

Pierre Quentel & Bendik Rønning Opstad

April 27, 2015

Contents

PyDbLite is

- a fast, pure-Python, untyped, in-memory database engine, using Python syntax to manage data, instead of SQL
- a pythonic interface to SQLite using the same syntax as the pure-Python engine for most operations (except database connection and table creation because of each database specificities)

PyDbLite is suitable for a small set of data where a fully fledged DB would be overkill.

Supported Python versions: 2.6+

CHAPTER 1

Installation

1.1 PIP

```
pip install pydblite
```

1.2 Manually

Download the source and execute

```
python setup.py install
```

1.2.1 Pure-Python engine

The pure-Python engine consists of one module, pydblite.py. To use it, import the class `Base` from this module:

```
from pydblite import Base
```

Create or open a database

Create a database instance, passing it a path in the file system

```
db = Base('test.pdl')
```

For a new database, define the field names

```
db.create('name', 'age', 'size')
```

You don't have to define the field types. Any value will be accepted as long as it can be serialized by the cPickle module:

- strings
- Unicode strings
- integers
- floats
- dates and datetimes (instances of the date and datetime classes in the datetime module)

- user-defined classes

`db.exists()` indicates if the base exists.

if the base exists, open it

```
if db.exists():
    db.open()
```

You can pass a parameter "mode" to the `create()` method, to specify what you want to do if the base already exists in the file system

- mode = "open" : `db.create('name', 'age', 'size', mode="open")` opens the database and ignores the field definition
- mode = "override" : `db.create('name', 'age', 'size', mode="override")` erases the existing base and creates a new one with the field definition
- if `mode` is not specified and the base already exists, an `IOError` is raised

Insert, update, delete a record

insert a new record

by keywords

```
db.insert(name='homer', age=23, size=1.84)
```

If some fields are missing, they are initialized with the value None

by positional arguments

```
db.insert('homer', 23, 1.84)
```

The arguments must be provided in the same order as in the `create()` method

save the changes on disk

```
db.commit()
```

If you don't commit the changes, the insertion, deletion and update operations will not be saved on disk. As long as changes are not commited, use `open()` to restore the values as they are currently on disk (this is equivalent to rollback in transactional databases)

delete a record

```
db.delete(record)
```

or, if you know the record identifier

```
del db[rec_id]
```

to delete a list of records

```
db.delete(list_of_records)
```

where list_of_records can be any iterable (list, tuple, set, etc) yielding records

to update a record

```
db.update(record, age=24)
```

- besides the fields passed to the `create()` method, an internal field called `__id__` is added. It is an integer which is guaranteed to be unique and unchanged for each record in the base, so that it can be used as the record identifier
- another internal field called `__version__` is also managed by the database engine. It is an integer which is set to 0 when the record is created, then incremented by 1 each time the record is updated. This is used to detect concurrency control, for instance in a web application where 2 users select the same record and want to update it at the same time

Selection

The instance of Base is a Python iterator

to iterate on all the records

```
for r in db:
    do_something_with(r)
```

Direct access

A record can be accessed by its identifier

```
record = db[rec_id]
```

returns the record such that `record['__id__'] == rec_id`

Simple selections

- `db(key1=val1, key2=val2)` returns the list of records where the keys take the given values
- `db(key) >= val` returns an iterator on all records where the value of the field key is greater than or equal to val.

Example

```
for rec in (db("age") > 30):
     print rec["name"]
```

such "rich comparison" operations can be combined with & (AND) and | (OR)

```
for rec in (db("age") > 30) & (db("country") == "France"):
    print rec["name"]
```

List comprehension

The selection of records can use Python list comprehension syntax

```
recs = [r for r in db if 30 > r['age'] >= 18 and r['size'] < 2]
```

Returns the records in the base where the age is between 18 and 30, and size is below 2 meters. The record is a dictionary, where the key is the field name and value is the field value

Python generator expression syntax can also be used

```
for r in (r for r in db if r['name'] in ('homer', 'marge')):
    do_something_with(r)
```

iterates on the records where the name is one of 'homer' or 'marge'

Index

To speed up selections, an index can be created on a field using `create_index('field')`

```
db.create_index('age')
```

When an index is created, the database instance has an attribute (here `_age` : note the heading underscore, to avoid name conflicts with internal names). This attribute is a dictionary-like object, where keys are the values taken by the field, and values are the records whose field values are egal to the key :

`records = db._age[23]` returns the list of records with `age == 23`

If no record has this value, lookup by this value returns an empty list

The index supports iteration on the field values, and the `keys()` method returns all existing values for the field

Other attributes and methods

- `add_field('new_field'[,default=v])`: adds a new field to an existing base. `default` is an optional default value ; set to `None` if not specified
- `drop_field('field')`: drops an existing field
- `db.path`: the path of the database in the file system
- `db.name`: the database name : the basename of the path, stripped of its extension
- `db.fields`: the list of the fields (does not include the internal fields `__id__` and `__version__`)
- `len(db)` : number of records in the base

1.2.2 SQLite adapter

The main difference with the pure-Python module is the syntax to identify a database and a table, and the need to specify field types on base creation

For compliance with SQLite vocabulary, the module defines two classes, `Database` and `Table`

Database

`Database(db_path[,**kw])` : db_path is the database path in the file system. The keyword arguments are the same as for the method `connect()` of the Python built-in module sqlite3

Instances of Database are dictionary-like objects, where keys are the table names and values are instances of the `Table` class

- `db["foo"]` returns the instance of the Table class for table "foo"
- `db.keys()` returns the table names

- `if "foo" in db` tests if table "foo" exists in the database
- `del db["foo"]` drops the table "foo"

To create a new table

```
table = db.create(table_name, *fields[,mode])
```

The fields must be 2-element tuples `(field_name, field_type)` where field_type is an SQLite field type

- INTEGER
- REAL
- TEXT
- BLOB

```
db.create('test', ('name', 'TEXT'), ('age', 'INTEGER'), ('size', 'REAL'))
```

If other information needs to be provided, put it in the second argument, using the SQL syntax for SQLite

```
db.create('test', ('date', 'BLOB DEFAULT CURRENT_DATE'))
```

The optional keyword argument `mode` specifies what you want to do if a table of the same name already exists in the database

- `mode="open"` opens the table and ignores the field definition
- `mode="override"` erases the existing table and creates a new one with the field definition
- if `mode` is not specified and the table already exists, an `IOError` is raised

Table

For record insertion, updating, deletion and selection the syntax is the same as for the *pure-Python module*. The SQLite primary key rowid is used like the key `__id__` to identify records

To insert many records at a time,

```
table.insert(list_of_values)
```

will be much faster than

```
for values in list_of_values:
    table.insert(values)
```

Note that you can't use the `drop_field()` method, since dropping fields is not supported by SQLite

Type conversion

Conversions between Python types and SQLite field types use the behaviour of the Python SQLite module. `datetime.date`, `datetime.time` and `datetime.datetime` instances are stored as ISO dates/datetimes

Selection methods return dictionaries, with SQLite types converted to Python types like this

SQLite type	Python type
NULL	None
TEXT	unicode
BLOB	str
INTEGER	int
REAL	float

If you want fields to be returned as instances of datetime.date, datetime.time or datetime.datetime instances, you can specify it when creating or opening the table, using methods `is_date(field_name)`, `is_time(field_name)` or `is_datetime(field_name)`.

```
db = Database('test.sqlite')
table = db['dummy']
table.is_date('birthday')
```

cursor and commit

Instances of `Database` and `Table` have the attribute `cursor`, the SQLite connections cursor, so you can also execute SQL expressions by

```
db.cursor.execute(some_sql)
```

and get the result by

```
results = db.cursor.fetchall()
```

the method `commit()` saves the changes to a database after a transaction

```
db.commit()
```

1.2.3 API

PyDbLite.PyDbLite API

class pydblite.pydblite.**Index**(*db*, *field*)
Class used for indexing a base on a field. The instance of Index is an attribute of the Base instance

class pydblite.pydblite.**_Base**(*path*, *protocol=2*, *save_to_file=True*, *sqlite_compat=False*)
protocol as defined in pickle / pickle. Defaults to the highest protocol available. For maximum compatibility use protocol = 0

__init__(*path*, *protocol=2*, *save_to_file=True*, *sqlite_compat=False*)
protocol as defined in pickle / pickle. Defaults to the highest protocol available. For maximum compatibility use protocol = 0

add_field(*field*, *column_type='ignored'*, *default=None*)
Adds a field to the database

commit()
Write the database to a file

create(**fields*, ***kw*)
Create a new base with specified field names.

Args:

- *fields (str): The field names to create.
- mode (str): the mode used when creating the database.

•if mode = 'create' : create a new base (the default value)

•if mode = 'open' : open the existing base, ignore the fields

•if mode = 'override' : erase the existing base and create a new one with the specified fields

Returns:

- the database (self).

create_index(**fields*)
Create an index on the specified field names

An index on a field is a mapping between the values taken by the field and the sorted list of the ids of the records whose field is equal to this value

For each indexed field, an attribute of self is created, an instance of the class Index (see above). Its name it the field name, with the prefix _ to avoid name conflicts

Args:

- fields (list): the fields to index

delete(*remove*)
Remove a single record, or the records in an iterable

Before starting deletion, test if all records are in the base and don't have twice the same __id__

Args:

- remove (record or list of records): The record(s) to delete.

Returns:

- Return the number of deleted items

delete_index(**fields*)
Delete the index on the specified fields

drop_field(*field*)
Removes a field from the database

exists()

Returns:

- bool: if the database file exists

fields = None
The list of the fields (does not include the internal fields __id__ and __version__)

get_indices()
Returns the indices

get_unique_ids(*id_value*, *db_filter=None*)
Returns a set of unique values from column

group_by(*column*, *torrents_filter*)
Returns the records grouped by column

insert(**args*, ***kw*)
Insert one or more records in the database.

Parameters can be positional or keyword arguments. If positional they must be in the same order as in the create() method If some of the fields are missing the value is set to None

Args:

- args (values, or a list/tuple of values): The record(s) to insert.
- kw (dict): The field/values to insert

Returns:

- Returns the record identifier if inserting one item, else None.

name = None
The basename of the path, stripped of its extension

open()
Open an existing database and load its content into memory

path = None
The path of the database in the file system

update(*records*, ***kw*)
Update one record or a list of records with new keys and values and update indices

Args:

- records (record or list of records): The record(s) to update.

PyDbLite.SQLite API

Main differences from `pydblite.pydblite`:

- pass the connection to the `SQLite db` as argument to `Table`
- in `create()` field definitions must specify a type.
- no *drop_field* (not supported by SQLite)
- the `Table` instance has a `cursor` attribute, so that raw SQL requests can be executed.

exception pydblite.sqlite.**SQLiteError**
SQLiteError

class pydblite.sqlite.**Database**(*filename*, ***kw*)
To create an in-memory database provide ':memory:' as filename

Args:

- filename (str): The name of the database file, or ':memory:'
- kw (dict): Arguments forwarded to sqlite3.connect

commit()
Save any changes to the database

conn = None
The SQLite connection

cursor = None
The SQLite connections cursor

class pydblite.sqlite.**Table**(*table_name*, *db*)
Args:

•table_name (str): The name of the SQLite table.

•db (`Database`): The database.

__call__(**args*, ***kw*)
Selection by field values.

db(key=value) returns the list of records where r[key] = value

Args:

- args (list): A field to filter on.

- kw (dict): pairs of field and value to filter on.

Returns:

- When args supplied, return a `Filter` object that filters on the specified field.
- When kw supplied, return all the records where field values matches the key/values in kw.

__delitem__(*record_id*)
Delete by record id

__getitem__(*record_id*)
Direct access by record id.

__init__(*table_name*, *db*)
Args:

•table_name (str): The name of the SQLite table.

•db (`Database`): The database.

__iter__()
Iteration on the records

add_field(*name*, *column_type='TEXT'*, *default=None*)
Add a new column to the table.

Args:

- name (string): The name of the field
- column_type (string): The data type of the column (Defaults to TEXT)
- default (datatype): The default value for this field (if any)

commit()
Save any changes to the database

conv(*field_name*, *conv_func*)
When a record is returned by a SELECT, ask conversion of specified field value with the specified function.

create(**fields*, ***kw*)
Create a new table.

Args:

- fields (list of tuples): The fields names/types to create. For each field, a 2-element tuple must be provided:
  - the field name
  - a string with additional information like field type + other information using the SQLite syntax eg ('name', 'TEXT NOT NULL'), ('date', 'BLOB DEFAULT CURRENT_DATE')
- **mode (str): The mode used when creating the database.** mode is only used if a database file already exists.
  - if mode = 'open' : open the existing base, ignore the fields
  - if mode = 'override' : erase the existing base and create a new one with the specified fields

Returns:

- the database (self).

cursor = None
The SQLite connections cursor

delete(*removed*)
Remove a single record, or the records in an iterable.

Before starting deletion, test if all records are in the base and don't have twice the same __id__.

Returns:

- int: the number of deleted items

insert(**args*, ***kw*)
Insert a record in the database.

Parameters can be positional or keyword arguments. If positional they must be in the same order as in the `create()` method.

Returns:

- The record identifier

is_date(*field_name*)
Ask conversion of field to an instance of datetime.date

is_datetime(*field_name*)
Ask conversion of field to an instance of datetime.date

is_time(*field_name*)
Ask conversion of field to an instance of datetime.date

open()
Open an existing database.

update(*record*, ***kw*)
Update the record with new keys and values.

PyDbLite.common API

class pydblite.common.**Filter**(*db*, *key*)
A filter to be used to filter the results from a database query. Users should not have to use this class.

__and__(*other_filter*)
Returns a new filter that combines this filter with other_filter using AND.

__eq__(*value*)
Perform EQUALS operation When input value is an iterable, but not a string, it will match for any of the values on the iterable

__ge__(*value*)
Perform GREATER THAN OR EQUALS operation

__gt__(*value*)
Perform GREATER THAN operation

__iter__()
Returns in iterator over the records for this filter

__le__(*value*)
Perform LESS THAN OR EQUALS operation

__len__()
Returns the number of records that matches this filter

__lt__(*value*)
Perform LESS THAN operation

__ne__(*value*)
Perform NOT EQUALS operation

__or__(*other_filter*)
Returns a new filter that combines this filter with other_filter using OR.

__str__()
Returns a string representation of the filter

filter()
Returns the filter

ilike(*value*)
Perform ILIKE operation

is_filtered()
If the filter contains any filters

like(*value*)
Perform LIKE operation

1.2.4 Example code

Pure Python

```
#!/usr/bin/env python
    table.commit()


def pydblite():
    from pydblite.pydblite import Base
    db = Base('dummy', save_to_file=False)
    # create new base with field names
    db.create('name', 'age', 'size')
    # insert new record
    db.insert(name='homer', age=23, size=1.84)
    # records are dictionaries with a unique integer key __id__
    # simple selection by field value
    records = db(name="homer")
    # complex selection by list comprehension
    res = [r for r in db if 30 > r['age'] >= 18 and r['size'] < 2]
    print "res:", res
    # delete a record or a list of records
    r = records[0]
    db.delete(r)

    list_of_records = []
    r = db.insert(name='homer', age=23, size=1.84)
    list_of_records.append(db[r])
    r = db.insert(name='marge', age=36, size=1.94)
    list_of_records.append(db[r])

    # or generator expression
    for r in (r for r in db if r['name'] in ('homer', 'marge')):
        # print "record:", r
        pass
```

```
    db.delete(list_of_records)

    rec_id = db.insert(name='Bart', age=15, size=1.34)
    record = db[rec_id]  # the record such that record['__id__'] == rec_id

    # delete a record by its id
    del db[rec_id]

    # create an index on a field
    db.create_index('age')
    # update
    rec_id = db.insert(name='Lisa', age=13, size=1.24)

    # direct access by id
    record = db[rec_id]

    db.update(record, age=24)
    # add and drop fields
    db.add_field('new_field', default=0)
    db.drop_field('name')
    # save changes on disk
    db.commit()
if __name__ == "__main__":
```

SQLite

```
#!/usr/bin/env python
#


def sqlite():
    from pydblite.sqlite import Database, Table
    # connect to SQLite database "test"
    db = Database(":memory:")
    # pass the table name and database path as arguments to Table creation
    table = Table('dummy', db)
    # create new base with field names
    table.create(('name', 'TEXT'), ('age', 'INTEGER'), ('size', 'REAL'))
    # existing base
    table.open()
    # insert new record
    table.insert(name='homer', age=23, size=1.84)
    table.insert(name='marge', age=36, size=1.94)
    rec_id = table.insert(name='Lisa', age=13, size=1.24)

    # records are dictionaries with a unique integer key __id__
    # selection by list comprehension
    res = [r for r in table if 30 > r['age'] >= 18 and r['size'] < 2]
    print "res:", res
    # or generator expression
    for r in (r for r in table if r['name'] in ('homer', 'marge')):
        pass
    # simple selection (equality test)
    records = table(age=23)

    # delete a record by its id
```

```
del table[rec_id]

rec_id = records[0]['__id__']

# direct access by id
record = table[rec_id]  # the record such that record['__id__'] == rec_id
# update
table.update(record, age=24)
# add a field
table.add_field('new_field')  # Defaults to type 'TEXT'
# save changes on disk
db.commit()
```

1.2.5 Unit tests

Check the test coverage.

class tests.common_tests.**Generic**

Generic class for unit testing pydblite.pydblite and pydblite.sqlite

reset_status_values_for_filter()

setup_db_for_filter()

test_add_field()

test_add_field_with_default_value()

test_create_index()

test_del()

test_delete()

test_delete_index()

test_fetch()

test_filter_and()

Test AND

test_filter_equals()

test_filter_get_unique_ids()

test_filter_greater()

test_filter_greater_equals()

test_filter_ilike()

Test text case sensitive

test_filter_in()

Test IN (== with a list

test_filter_len()

test_filter_less()

test_filter_less_equals()

test_filter_like()

Test text case insensitive

test_filter_not_equals()

test_filter_or()
Test OR

test_get_group_count()

test_insert()

test_insert_values_in_order()

test_iter()

test_select()

test_select_unicode()

test_update()

class tests.test_pydblite.**PyDbLiteTestCase**(*methodName='runTest'*)
Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

setUp()

setup_db_for_filter()

tearDown()

test_insert_list()

test_open()

test_open_existing()

test_open_file_with_existing_dir()

test_open_memory()

test_open_memory_with_existing_filename()

test_sqlite_compat()

test_sqlite_compat_insert_list()

class tests.test_pydblite_sqlite.**SQLiteTestCase**(*methodName='runTest'*)
Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

setUp()

tearDown()

test_open_existing()

class tests.test_pydblite_sqlite.**TestSQLiteFunctions**(*methodName='runTest'*)
Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

insert_test_data()

setUp()

test_00_create()

test_02_iterate_on_database()

test_10_insert_one()

test_11_insert_many()

test_12_insert_kw_unicode_missing()

test_20_select()

test_40_delete()

1.2.6 Benchmarks

PyDbLite has been measured by the time taken by various operations for 3 pure-Python database modules (PyDbLite, buzhug and Gadfly) and compared them with SQLite.

The tests are those described on the SQLite comparisons pages, which compares performance of SQLite to that of MySQL and PostGreSQL

insert

create the base and insert n elements (n= 1000, 25,000 or 100,000) in it

The database has 3 fields:

- a (integer, from 1 to n)
- b (random integer between 1 and 100000)
- c (a string, value = 'fifty nine' if b=59)

For PyDbLite, gadfly and SQLite two options are possible : with an index on field a, or without index

The values of a, b, c are stored in a list recs

SQL statements

```
cursor.execute("CREATE TABLE t1(a INTEGER, b INTEGER, c VARCHAR(100))")
if make_index:
    cursor.execute("CREATE INDEX i3 ON t1(a)")
for a, b, c in recs:
    cursor.execute("INSERT INTO t1 VALUES(%s,%s,'%s')" %(a, b, c))
conn.commit()
```

PyDbLite code

```
db = PyDbLite.Base(name).create('a','b','c')
if index:
    db.create_index('a')
for a,b,c in recs:
   db.insert(a=a,b=b,c=c)
db.commit()
```

buzhug code

```
db=Base('t1').create(('a', int), ('b', int), ('c', str))
for rec in recs:
    db.insert(*rec)
db.commit()
```

gadfly code

```
conn = gadfly.gadfly()
conn.startup(folder_name,folder_name)
cursor = conn.cursor()
cursor.execute("CREATE TABLE t1(a INTEGER, b INTEGER, c VARCHAR(100))")
if make_index:
    cursor.execute("CREATE INDEX i3 ON t1(a)")
insertstat = "INSERT INTO t1 VALUES(?,?,?)"
for a, b, c in recs:
    cursor.execute(insertstat,(a, b, c))
conn.commit()
```

select1

100 selections to count the number of records and the average of field b for values of b between 10*n and 10*n + 1000 for n = 1 to 100

SQL statements

```
for i in range(100):
    sql = 'SELECT count(*), avg(b) FROM t1 WHERE b>=%s AND b<%s' % (100 * i, 1000 + 100 * i)
    cursor.execute(sql)
    nb,avg = cursor.fetchall()[0]
```

buzhug code

```
for i in range(100):
    recs = db.select(['b'], b=[100 * i, 999 + 100 * i])
    nb = len(recs)
    if nb:
        avg = sum([r.b for r in recs]) / nb
```

select2

100 selections to count the number of records and the average of field b for values of c with the string 'one', 'two', ...,'ninety nine' inside. It uses the keyword LIKE for SQL database (I couldn't do the test for Gadfly which doesn't support LIKE) ; for buzhug I use regular expressions. The strings for each number between 0 and 99 are stored in the list num_strings

SQL statements

```
for num_string in num_strings:
    sql = "SELECT count(*), avg(b) FROM t1 WHERE c LIKE '%%%s%%'" %num_string
    cursor.execute(sql)
    nb,avg = cursor.fetchall()[0]
```

buzhug code

```
for num_string in num_strings:
    pattern = re.compile(".*"+num_string+".*")
    recs = db.select(['b'], 'p.match(c)', p=pattern)
    nb = len(recs)
    if nb:
        avg = sum([r.b for r in recs]) / nb
```

delete1

delete all the records where the field c contains the string 'fifty'. There again I couldn't do the test for gadfly

SQL statements

```
sql = "DELETE FROM t1 WHERE c LIKE '%fifty%';"
cursor.execute(sql)
conn.commit()
```

buzhug code

```
db.delete(db.select(['__id__'], 'p.match(c)', p=re.compile('.*fifty.*')))
```

delete2

delete all the records for which the field a is > 10 and < 20000

SQL statements

```
sql="DELETE FROM t1 WHERE a > 10 AND a < 20000;"
cursor.execute(sql)
conn.commit()
```

buzhug code

```
db.delete(db.select(['__id__'], 'x < a < y', x=10, y=20000))
```

update1

1000 updates, multiply b by 2 for records where 10*n <= a < 10 * (n + 1) for n = 0 to 999

SQL statements

```
for i in range(100):
    sql="UPDATE t1 SET b = b * 2 WHERE a>=%s AND a<%s;" % (10 * i, 10 * (i + 1))
    cursor.execute(sql)
conn.commit()
```

buzhug code

```
for i in range(100):
    for r in db.select(a=[10 * i, 10 * i + 9]):
        db.update(r, b=r.b * 2)
```

update2

1000 updates to set c to a random value where a = 1 to 1000 New values of field c are stored in a list new_c

SQL statements

```
for i in range(0, 1000):
    sql="UPDATE t1 SET c='%s' WHERE a=%s" % (new_c[i], i)
    cursor.execute(sql)
conn.commit()
```

buzhug code

```
recs = db.select_for_update(['a','c'], a=[1,999])
for r in recs:
    db.update(r, c=new_c[r.a])
```

The tests were made on a Windows XP machine, with Python 2.5 (except gadfly : using the compile kjbuckets.pyd requires Python 2.2)

Versions : PyDbLite 2.5, buzhug 1.6, gadfly 1.0.0, SQLite 3.0 embedded in Python 2.5 Results

Here are the results

1000 records

	PyDbLite		sqlite		gadfly		buzhug
	no index	index	no index	index	no index	index	
size (kO)	79	91	57	69	60		154
create	0.04	0.03	1.02	0.77	0.71	2.15	0.29
select1	0.06	0.07	0.09	0.09	1.50	1.49	0.21
select2	0.04	0.04	0.16	0.16	-	-	0.51
delete1	0.01	0.02	0.49	0.50	-	-	0.04
delete2	0.08	0.01	0.56	0.26	0.04	0.05	0.17
update1	0.07	0.07	0.52	0.37	1.91	1.91	0.49
update2	0.20	0.03	0.99	0.45	7.72	0.54	0.72

```
25,000 records

                PyDbLite              sqlite           gadfly         buzhug
           no index     index    no index   index   no index  index

size         2021       2339        1385    1668      2948              2272

create       0.73       1.28        2.25    2.20    117.04              7.04
select1      2.31       2.72        2.69    2.67    153.05              3.68
select2      1.79       1.71        4.53    4.48    -                  12.33
delete1      0.40       0.89        1.88    0.98    -           (1)     0.84
delete2      0.22       0.35        0.82    0.69      1.78              2.88
update1      2.85       3.55        2.65    0.45    183.06              1.23
update2     18.90       0.96       10.93    0.47    218.30              0.81

100,000 records

                PyDbLite              sqlite       buzhug
           no index     index    no index   index

size         8290       9694        5656      6938     8881

create       4.07       7.94        5.54      7.06    28.23
select1      9.27      13.73        9.86      9.99    14.72
select2      7.49       8.00       16.86     16.64    51.46
delete1      2.97       4.10        2.58      3.58     3.48
delete2      3.00       4.23        0.98      1.41     3.31
update1     13.72      15.80        9.22      0.99     1.87
update2     24.83       5.95       69.61      1.21     0.93


(1) not tested with index, creation time is +INF
```

Conclusions PyDblite is as fast, and even faster than SQLite for small databases. It is faster than gadfly in all cases. buzhug is faster on most operations when size grows

1.2.7 Changelog

3.0.2 (2015-04-18)

- Fixed bug (#1) in `Base` where calling `db.create()` without supplying a value for the "mode" parameter with give an error.

3.0.1 (2015-02-23)

- Fixed bug in `Base` where opening existing database would fail.

3.0 (2014-09-18)

Note: Some changes in this release are not backwards compatible with 2.X versions.

- pydblite and sqlite are rewritten to use a common `Filter` object.
- Tests have been improved and standardised in *Unit tests*.

- Updated *Example code*.
- Renamed module and package names to lower case according to **PEP 8**
- Converted to UNIX line endings and follow **PEP 8** code style.
- MySQL adapter has been dropped until it can be tested with unit tests.

2.6

- if db exists, read field names on instance creation
- allow add_field on an instance even if it was not open()
- attribute path is the path of the database in the file system (was called "name" in previous versions)
- attribute name is the base name of the database, without the extension
- adapt code to run on Python 2 and Python 3

2.5

- test is now in folder "test"
- SQLite changes:
  - many changes to support "legacy" SQLite databases
    - no control on types declared in CREATE TABLE or ALTER TABLE
    - no control on value types in INSERT or UPDATE
    - no version number in records
  - add methods to specify a conversion function for fields after a SELECT
  - change names to be closer to SQLite names
    - a class Database to modelise the database
    - a class Table (not Base) for each table in the database

2.4

- add BSD Licence
- raise exception if unknown fields in insert

2.3

- introduce syntax (db('name')>'f') & (db('age') == 30)

2.2

- add __contains__

1.2.8 License

PyDbLite is an open Source software, published under the BSD licence :

1.2.9 Documentation coverage

Pierre Quentel: at pierre(dot)quentel(at)gmail(dot)com

Bendik Rønning Opstad: bro.development gmail.com

CHAPTER 2

Indices and tables

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

p

t

Symbols

A

C

D

E

F

G

I

L

N

O

P

R

S

T

U