sophy Documentation

Release 0.4.2

Charles Leifer

Nov 01, 2018

Contents:

Python binding for sophia embedded database, v2.2.

- Written in Cython for speed and low-overhead
- Clean, memorable APIs
- Comprehensive support for Sophia's features
- Supports Python 2 and 3.
- No 3rd-party dependencies besides Cython (for building).

About Sophia:

- Ordered key/value store
- Keys and values can be composed of multiple fieldsdata-types
- ACID transactions
- MVCC, optimistic, non-blocking concurrency with multiple readers and writers.
- Multiple databases per environment
- Multiple- and single-statement transactions across databases
- Prefix searches
- Automatic garbage collection and key expiration
- Hot backup
- Compression
- Multi-threaded compaction
- `mmap` support, direct I/O support
- APIs for variety of statistics on storage engine internals
- BSD licensed

Some ideas of where Sophia might be a good fit:

- Running on application servers, low-latency / high-throughput
- Time-series
- Analytics / Events / Logging
- Full-text search
- Secondary-index for external data-store

Limitations:

- Not tested on Windoze.

If you encounter any bugs in the library, please open an issue, including a description of the bug and any related traceback.

CHAPTER 1

Installing

Up-to-date source code for sophia is bundled with the `sophy` source code, so the only thing you need to build is Cython. If Cython is not installed, then the pre-generated C source files will be used.

sophy can be installed directly from the source or from pypi using `pip`.

1.1 Installing with pip

To install from PyPI:

```
$ pip install cython  # optional
$ pip install sophy
```

To install the very latest version, you can install with git:

```
$ pip install -e git+https://github.com/coleifer/sophy#egg=sophy
```

1.2 Obtaining the source code

The source code is hosted on github and can be obtained and installed:

```
$ git clone https://github.com/colefer/sophy
$ cd sophy
$ python setup.py build
$ python setup.py install
```

1.3 Running the tests

Unit-tests and integration tests are distributed with the source and can be run from the root of the checkout:

```
$ python tests.py
```

CHAPTER 2

Quick-start

Sophy is very simple to use. It acts like a Python `dict` object, but in addition to normal dictionary operations, you can read slices of data that are returned efficiently using cursors. Similarly, bulk writes using `update()` use an efficient, atomic batch operation.

Despite the simple APIs, Sophia has quite a few advanced features. There is too much to cover everything in this document, so be sure to check out the official Sophia storage engine documentation.

The next section will show how to perform common actions with `sophy`.

2.1 Using Sophy

Let's begin by importing `sophy` and creating an *environment*. The environment can host multiple *databases*, each of which may have a different *schema*. In this example our database will store UTF-8 strings as the key and value (though other data-types are supported). Finally we'll open the environment so we can start storing and retrieving data.

```
from sophy import Sophia, Schema, StringIndex

# Instantiate our environment by passing a directory path which will store
# the various data and metadata for our databases.
env = Sophia('/tmp/sophia-example')

# We'll define a very simple schema consisting of a single utf-8 string for
# the key, and a single utf-8 string for the associated value. Note that
# the key or value accepts multiple indexes, allowing for composite
# data-types.
schema = Schema([StringIndex('key')], [StringIndex('value')])

# Create a key/value database using the schema above.
db = env.add_database('example_db', schema)

if not env.open():
    raise Exception('Unable to open Sophia environment.')
```

In the above example we used *StringIndex* which stores UTF8-encoded string data. The following index types are available:

- *StringIndex* - UTF8-encoded string data (text, in other words).
- *BytesIndex* - bytestrings (binary data).
- *JsonIndex* - store value as UTF8-encoded JSON.
- *MsgPackIndex* - store arbitrary data using msgpack encoding.
- *PickleIndex* - store arbitrary data using python pickle module.
- *UUIDIndex* - store UUIDs.
- *SerializedIndex* - index that accepts serialize/deserialize functions and can be used for msgpack or pickled data, for example.
- *U64Index* - store 64-bit unsigned integers.
- *U32Index* - store 32-bit unsigned integers.
- *U16Index* - store 16-bit unsigned integers.
- *U8Index* - store 8-bit unsigned integers (or single bytes).
- There are also *U64RevIndex*, *U32RevIndex*, *U16RevIndex* and *U8RevIndex* for storing integers in reverse order.

2.2 CRUD operations

Sophy databases use the familiar `dict` APIs for CRUD operations:

```
>>> db['name'] = 'Huey'
>>> db['animal_type'] = 'cat'
>>> print(db['name'], 'is a', db['animal_type'])
Huey is a cat

>>> 'name' in db
True
>>> 'color' in db
False

>>> del db['name']
>>> del db['animal_type']
>>> print(db['name'])  # raises a KeyError.
KeyError: ('name',)
```

To insert multiple items efficiently, use the `Database.update()` method. Multiple items can be retrieved or deleted efficiently using *Database.multi_get()*, *Database.multi_get_dict()*, and *Database.multi_delete()*:

```
>>> db.update(k1='v1', k2='v2', k3='v3')
>>> for value in db.multi_get('k1', 'k3', 'kx'):
...     print(value)

v1
v3
None
```

(continues on next page)

(continued from previous page)

```
>>> db.multi_get_dict(['k1', 'k3', 'kx'])
{'k1': 'v1', 'k3': 'v3'}

>>> db.multi_delete('k1', 'k3', 'kx')
>>> 'k1' in db
False
```

2.3 Other dictionary methods

Sophy databases also provide efficient implementations of *keys()*, *values()* and *items()* for iterating over the data-set. Unlike dictionaries, however, iterating directly over a Sophy *Database* will return the equivalent of the *items()* method (as opposed to just the keys).

Note: Sophia is an ordered key/value store, so iteration will return items in the order defined by their index. So for strings and bytes, this is lexicographic ordering. For integers it can be ascending or descending.

```
>>> db.update(k1='v1', k2='v2', k3='v3')
>>> list(db)
[('k1', 'v1'),
 ('k2', 'v2'),
 ('k3', 'v3')]

>>> db.items()  # Returns a Cursor, which can be iterated.
<sophy.Cursor at 0x7f1dac231ee8>
>>> [item for item in db.items()]
[('k1', 'v1'),
 ('k2', 'v2'),
 ('k3', 'v3')]

>>> list(db.keys())
['k1', 'k2', 'k3']

>>> list(db.values())
['v1', 'v2', 'v3']
```

There are two ways to get the count of items in a database. You can use the `len()` function, which is not very efficient since it must allocate a cursor and iterate through the full database. An alternative is the `Database.index_count` property, which may not be exact as it includes transaction duplicates and not-yet-merged duplicates:

```
>>> len(db)
3
>>> db.index_count
3
```

2.4 Range queries

Because Sophia is an ordered data-store, performing ordered range scans is efficient. To retrieve a range of key-value pairs with Sophy, use the ordinary dictionary lookup with a `slice` as the index:

```
>>> db.update(k1='v1', k2='v2', k3='v3', k4='v4')
>>> db['k1':'k3']
<generator at 0x7f1db413bbf8>

>>> list(db['k1':'k3'])  # NB: other examples omit list() for clarity.
[('k1', 'v1'), ('k2', 'v2'), ('k3', 'v3')]

>>> db['k1.x':'k3.x']  # Inexact matches are OK, too.
[('k2', 'v2'), ('k3', 'v3')]

>>> db[:'k2']  # Omitting start or end retrieves from first/last key.
[('k1', 'v1'), ('k2', 'v2')]

>>> db['k3':]
[('k3', 'v3'), ('k4', 'v4')]

>>> db['k3':'k1']  # To retrieve a range in reverse, use the higher key first.
[('k3', 'v3'), ('k2', 'v2'), ('k1', 'v1')]
```

To retrieve a range in reverse order where the start or end is unspecified, you can pass in `True` as the `step` value of the slice to also indicate reverse:

```
>>> db[:'k2':True]  # Start-to-"k2" in reverse.
[('k2', 'v2'), ('k1', 'v1')]

>>> db['k3'::True]
[('k4', 'v4'), ('k3', 'v3')]

>>> db[::True]
[('k4', 'v4'), ('k3', 'v3'), ('k2', 'v2'), ('k1', 'v1')]
```

2.5 Cursors

For finer-grained control over iteration, or to do prefix-matching, Sophy provides a *Cursor* interface.

The *cursor()* method accepts five parameters:

- `order` (default=‘‘>=‘‘) - semantics for matching the start key and ordering results.
- `key` - the start key
- `prefix` - search for prefix matches
- `keys` - (default=‘‘True‘‘) – return keys while iterating
- `values` - (default=‘‘True‘‘) – return values while iterating

Suppose we were storing events in a database and were using an ISO-8601-formatted date-time as the key. Since ISO-8601 sorts lexicographically, we could retrieve events in correct order simply by iterating. To retrieve a particular slice of time, a prefix could be specified:

```
# Iterate over events for July, 2017:
cursor = db.cursor(key='2017-07-01T00:00:00', prefix='2017-07-')
for timestamp, event_data in cursor:
    process_event(timestamp, event_data)
```

2.6 Transactions

Sophia supports ACID transactions. Even better, a single transaction can cover operations to multiple databases in a given environment.

Example of using *Sophia.transaction()*:

```
account_balance = env.add_database('balance', ...)
transaction_log = env.add_database('transaction_log', ...)

# ...

def transfer_funds(from_acct, to_acct, amount):
    with env.transaction() as txn:
        # To write to a database within a transaction, obtain a reference to
        # a wrapper object for the db:
        txn_acct_bal = txn[account_balance]
        txn_log = txn[transaction_log]

        # Transfer the asset by updating the respective balances. Note that we
        # are operating on the wrapper database, not the db instance.
        from_bal = txn_acct_bal[from_acct]
        txn_acct_bal[to_account] = from_bal + amount
        txn_acct_bal[from_account] = from_bal - amount

        # Log the transaction in the transaction_log database. Again, we use
        # the wrapper for the database:
        txn_log[from_account, to_account, get_timestamp()] = amount
```

Multiple transactions are allowed to be open at the same time, but if there are conflicting changes, an exception will be thrown when attempting to commit the offending transaction:

```
# Create a basic k/v store. Schema.key_value() is a convenience method
# for string key / string value.
>>> kv = env.add_database('main', Schema.key_value())

# Open the environment in order to access the new db.
>>> env.open()

# Instead of using the context manager, we'll call begin() explicitly so we
# can show the interaction of 2 open transactions.
>>> txn = env.transaction().begin()

>>> t_kv = txn[kv]  # Obtain reference to kv database in transaction.
>>> t_kv['k1'] = 'v1'  # Set k1=v1.

>>> txn2 = env.transaction().begin()  # Start a 2nd transaction.
>>> t2_kv = txn2[kv]  # Obtain a reference to the "kv" db in 2nd transaction.
>>> t2_kv['k1'] = 'v1-x'  # Set k1=v1-x

>>> txn2.commit()  # ERROR !!
SophiaError
...
SophiaError('transaction is not finished, waiting for concurrent transaction to␣
↪finish.')

>>> txn.commit()  # OK
```

(continues on next page)

(continued from previous page)

```
>>> txn2.commit()  # Retry committing 2nd transaction. ERROR !!
SophiaError
...
SophiaError('transasction rolled back by another concurrent transaction.')
```

Sophia detected a conflict and rolled-back the 2nd transaction.

2.7 Index types, multi-field keys and values

Sophia supports multi-field keys and values. Additionally, the individual fields can have different data-types. Sophy provides the following field types:

- *StringIndex* - UTF8-encoded string data (text, in other words).
- *BytesIndex* - bytestrings (binary data).
- *JsonIndex* - store value as UTF8-encoded JSON.
- *MsgPackIndex* - store arbitrary data using msgpack encoding.
- *PickleIndex* - store arbitrary data using python pickle module.
- *UUIDIndex* - store UUIDs.
- *SerializedIndex* - index that accepts serialize/deserialize functions and can be used for custom serialization formats.
- *U64Index* - store 64-bit unsigned integers.
- *U32Index* - store 32-bit unsigned integers.
- *U16Index* - store 16-bit unsigned integers.
- *U8Index* - store 8-bit unsigned integers (or single bytes).
- There are also *U64RevIndex*, *U32RevIndex*, *U16RevIndex* and *U8RevIndex* for storing integers in reverse order.

To store arbitrary data encoded using msgpack, for example:

```
schema = Schema(StringIndex('key'), MsgPackIndex('value'))
db = sophia_env.add_database('main', schema)
```

If you have a custom serialization library you would like to use, you can use *SerializedIndex*, passing the serialize/deserialize callables:

```
# Equivalent to previous msgpack example.
import msgpack

schema = Schema(StringIndex('key'),
                SerializedIndex('value', msgpack.packb, msgpack.unpackb))
db = sophia_env.add_database('main', schema)
```

To declare a database with a multi-field key or value, you will pass the individual fields as arguments when constructing the *Schema* object. To initialize a schema where the key is composed of two strings and a 64-bit unsigned integer, and the value is composed of a string, you would write:

```
# Declare a schema consisting of a multi-part key and a string value.
key_parts = [StringIndex('last_name'),
             StringIndex('first_name'),
             U64Index('area_code')]
value_parts = [StringIndex('address_data')]
schema = Schema(key_parts, value_parts)

# Create a database using the above schema.
address_book = env.add_database('address_book', schema)
env.open()
```

To store data, we use the same dictionary methods as usual, just passing tuples instead of individual values:

```
address_book['kitty', 'huey', 66604] = '123 Meow St'
address_book['puppy', 'mickey', 66604] = '1337 Woof-woof Court'
```

To retrieve our data:

```
>>> address_book['kitty', 'huey', 66604]
'123 Meow St.'
```

To delete a row:

```
>>> del address_book['puppy', 'mickey', 66604]
```

Indexing and slicing works as you would expect, with tuples being returned instead of scalar values where appropriate.

Note: When working with a multi-part value, a tuple containing the value components will be returned. When working with a scalar value, instead of returning a 1-item tuple, the value itself is returned.

2.8 Configuring and Administering Sophia

Sophia can be configured using special properties on the `Sophia` and `Database` objects. Refer to the *settings configuration document* for the details on the available options, including whether they are read-only, and the expected data-type.

For example, to query Sophia's status, you can use the `Sophia.status` property, which is a readonly setting returning a string:

```
>>> print(env.status)
online
```

Other properties can be changed by assigning a new value to the property. For example, to read and then increase the number of threads used by the scheduler:

```
>>> env.scheduler_threads
6
>>> env.scheduler_threads = 8
```

Database-specific properties are available as well. For example to get the number of GET and SET operations performed on a database, you would write:

```
>>> print(db.stat_get, 'get operations')
24 get operations
>>> print(db.stat_set, 'set operations')
33 set operations
```

Refer to the *settings configuration table* for a complete list of available settings.

2.9 Backups

Sophia can create a backup the database while it is running. To configure backups, you will need to set the path for backups before opening the environment:

```
env = Sophia('/path/to/data')
env.backup_path = '/path/for/backup-data/'

env.open()
```

At any time while the environment is open, you can call the `backup_run()` method, and a backup will be started in a background thread:

```
env.backup_run()
```

Backups will be placed in numbered folders inside the `backup_path` specified during environment configuration. You can query the backup status and get the ID of the last-completed backup:

```
env.backup_active  # Returns 1 if running, 0 if completed/idle
env.backup_last  # Get ID of last-completed backup
env.backup_last_complete  # Returns 1 if last backup succeeded
```

CHAPTER 3

Sophy API

class SophiaError

General exception class used to indicate error returned by Sophia database.

3.1 Environment

class Sophia(*path*)

Parameters path (*str*) – Directory path to store environment and databases.

Environment object providing access to databases and for controlling transactions.

Example of creating environment, attaching a database and reading/writing data:

```
from sophy import *


# Environment for managing one or more databases.
env = Sophia('/tmp/sophia-test')

# Schema describes the indexes that comprise the key and value portions
# of a database.
kv_schema = Schema([StringIndex('key')], [StringIndex('value')])
db = env.add_data('kv', kv_schema)

# We need to open the env after configuring the database(s), in order
# to read/write data.
assert env.open(), 'Failed to open environment!'

# We can use dict-style APIs to read/write key/value pairs.
db['k1'] = 'v1'
assert db['k1'] == 'v1'
```

(continues on next page)

(continued from previous page)

```
# Close the env when finished.
assert env.close(), 'Failed to close environment!'
```

open()

Returns Boolean indicating success.

Open the environment. The environment must be opened in order to read and write data to the configured databases.

close()

Returns Boolean indicating success.

Close the environment.

add_database(*name*, *schema*)

Parameters

- **name** (`str`) – database name
- **schema** (`Schema`) – schema for keys and values.

Returns a database instance

Return type *Database*

Add or declare a database. Environment must be closed to add databases. The *Schema* will declare the data-types and structure of the key- and value-portion of the database.

```
env = Sophia('/path/to/db-env')

# Declare an events database with a multi-part key (ts, type) and
# a msgpack-serialized data field.
events_schema = Schema(
    key_parts=[U64Index('timestamp'), StringIndex('type')],
    value_parts=[MsgPackIndex('data')])
db = env.add_database('events', events_schema)

# Open the environment for read/write access to the database.
env.open()

# We can now write to the database.
db[current_time(), 'init'] = {'msg': 'event logging initialized'}
```

remove_database(*name*)

Parameters **name** (`str`) – database name

Remove a database from the environment. Environment must be closed to remove databases. This method does really not have any practical value but is provided for consistency.

get_database(*name*)

Returns the database corresponding to the provided name

Return type *Database*

Obtain a reference to the given database, provided the database has been added to the environment by a previous call to *add_database()*.

__getitem__(*name*)
Short-hand for *get_database()*.

transaction()

Returns a transaction handle.

Return type *Transaction*

Create a transaction handle which can be used to execute a transaction on the databases in the environment. The returned transaction can be used as a context-manager.

Example:

```
env = Sophia('/tmp/sophia-test')
db = env.add_database('test', Schema.key_value())
env.open()

with env.transaction() as txn:
    t_db = txn[db]
    t_db['k1'] = 'v1'
    t_db.update(k2='v2', k3='v3')

# Transaction has been committed.
print(db['k1'], db['k3'])  # prints "v1", "v3"
```

See *Transaction* for more information.

3.2 Database

class Database

Database interface. This object is not created directly, but references can be obtained via *Sophia.add_database()* or *Sophia.get_database()*.

For example:

```
env = Sophia('/path/to/data')

kv_schema = Schema(StringIndex('key'), MsgPackIndex('value'))
kv_db = env.add_database('kv', kv_schema)

# Another reference to "kv_db":
kv_db = env.get_database('kv')

# Same as above:
kv_db = env['kv']
```

set(*key*, *value*)

Parameters

- **key** – key corresponding to schema (e.g. scalar or tuple).
- **value** – value corresponding to schema (e.g. scalar or tuple).

Returns No return value.

Store the value at the given key. For single-index keys or values, a scalar value may be provided as the key or value. If a composite or multi-index key or value is used, then a `tuple` must be provided.

Examples:

```
simple = Schema(StringIndex('key'), StringIndex('value'))
simple_db = env.add_database('simple', simple)

composite = Schema(
    [U64Index('timestamp'), StringIndex('type')],
    [MsgPackIndex('data')])
composite_db = env.add_database('composite', composite)

env.open()  # Open env to access databases.

# Set k1=v1 in the simple key/value database.
simple_db.set('k1', 'v1')

# Set new value in composite db. Note the key is a tuple and, since
# the value is serialized using msgpack, we can transparently store
# data-types like dicts.
composite_db.set((current_time, 'evt_type'), {'msg': 'foo'})
```

get(*key*[, *default=None*])

Parameters

- **key** – key corresponding to schema (e.g. scalar or tuple).
- **default** – default value if key does not exist.

Returns value of given key or default value.

Get the value at the given key. If the key does not exist, the default value is returned.

If a multi-part key is defined for the given database, the key must be a tuple.

Example:

```
simple_db.set('k1', 'v1')
simple_db.get('k1')  # Returns "v1".

simple_db.get('not-here')  # Returns None.
```

delete(*key*)

Parameters **key** – key corresponding to schema (e.g. scalar or tuple).

Returns No return value

Delete the given key, if it exists. If a multi-part key is defined for the given database, the key must be a tuple.

Example:

```
simple_db.set('k1', 'v1')
simple_db.delete('k1')  # Deletes "k1" from database.

simple_db.exists('k1')  # False.
```

exists(*key*)

Parameters **key** – key corresponding to schema (e.g. scalar or tuple).

Returns Boolean indicating if key exists.

Return type bool

Return whether the given key exists. If a multi-part key is defined for the given database, the key must be a tuple.

multi_set([*__data=None*[, ***kwargs*]])

Parameters

- **__data** (*dict*) – Dictionary of key/value pairs to set.
- **kwargs** – Specify key/value pairs as keyword-arguments.

Returns No return value

Set multiple key/value pairs efficiently.

multi_get(**keys*)

Parameters **keys** – key(s) to retrieve

Returns a list of values associated with the given keys. If a key does not exist a `None` will be indicated for the value.

Return type list

Get multiple values efficiently. Returned as a list of values corresponding to the `keys` argument, with missing values as `None`.

Example:

```
db.update(k1='v1', k2='v2', k3='v3')
db.multi_get('k1', 'k3', 'k-nothere')
# ['v1', 'v3', None]
```

multi_get_dict(*keys*)

Parameters **keys** (*list*) – list of keys to get

Returns a list of values associated with the given keys. If a key does not exist a `None` will be indicated for the value.

Return type list

Get multiple values efficiently. Returned as a dict of key/value pairs. Missing values are not represented in the returned dict.

Example:

```
db.update(k1='v1', k2='v2', k3='v3')
db.multi_get_dict(['k1', 'k3', 'k-nothere'])
# {'k1': 'v1', 'k3': 'v3'}
```

multi_delete(**keys*)

Parameters **keys** – key(s) to delete

Returns No return value

Efficiently delete multiple keys.

get_range(*start=None*, *stop=None*, *reverse=False*)

Parameters

- **start** – start key (omit to start at first record).
- **stop** – stop key (omit to stop at the last record).

- **reverse** (*bool*) – return range in reverse.

Returns a generator that yields the requested key/value pairs.

Fetch a range of key/value pairs from the given start-key, up-to and including the stop-key (if given).

keys()
Return a cursor for iterating over the keys in the database.

values()
Return a cursor for iterating over the values in the database.

items()
Return a cursor for iterating over the key/value pairs in the database.

__getitem__(*key_or_slice*)

Parameters **key_or_slice** – key or range of keys to retrieve.

Returns value of given key, or an iterator over the range of keys.

Raises KeyError if single key requested and does not exist.

Retrieve a single value or a range of values, depending on whether the key represents a single row or a slice of rows.

Additionally, if a slice is given, the start and stop values can be omitted to indicate you wish to start from the first or last key, respectively.

__setitem__(*key*, *value*)
Equivalent to `set()`.

__delitem__(*key*)
Equivalent to `delete()`.

__contains__(*key*)
Equivalent to `exists()`.

__iter__()
Equivalent to `items()`.

__len__()
Equivalent to iterating over all keys and returning count. This is the most accurate way to get the total number of keys, but is not very efficient. An alternative is to use the `Database.index_count` property, which returns an approximation of the number of keys in the database.

cursor(*order='>='*, *key=None*, *prefix=None*, *keys=True*, *values=True*)

Parameters

- **order** (*str*) – ordering semantics (default is ">=")
- **key** – key to seek to before iterating.
- **prefix** – string prefix to match.
- **keys** (*bool*) – return keys when iterating.
- **values** (*bool*) – return values when iterating.

Create a cursor with the given semantics. Typically you will want both `keys=True` and `values=True` (the defaults), which will cause the cursor to yield a 2-tuple consisting of `(key, value)` during iteration.

3.3 Transaction

class Transaction

Transaction handle, used for executing one or more operations atomically. This class is not created directly - use *Sophia.transaction()*.

The transaction can be used as a context-manager. To read or write during a transaction, you should obtain a transaction-specific handle to the database you are operating on.

Example:

```
env = Sophia('/tmp/my-env')
db = env.add_database('kv', Schema.key_value())
env.open()

with env.transaction() as txn:
    tdb = txn[db]  # Obtain reference to "db" in the transaction.
    tdb['k1'] = 'v1'
    tdb.update(k2='v2', k3='v3')

# At the end of the wrapped block, the transaction is committed.
# The writes have been recorded:
print(db['k1'], db['k3'])
# ('v1', 'v3')
```

begin()

Begin a transaction.

commit()

Raises SophiaError

Commit all changes. An exception can occur if:

1. The transaction was rolled back, either explicitly or implicitly due to conflicting changes having been committed by a different transaction. **Not recoverable**.
2. A concurrent transaction is open and must be committed before this transaction can commit. **Possibly recoverable**.

rollback()

Roll-back any changes made in the transaction.

__getitem__(*db*)

Parameters **db** (Database) – database to reference during transaction

Returns special database-handle for use in transaction

Return type DatabaseTransaction

Obtain a reference to the database for use within the transaction. This object supports the same APIs as *Database*, but any reads or writes will be made within the context of the transaction.

3.4 Schema Definition

class Schema(*key_parts*, *value_parts*)

Parameters

- **key_parts** (*list*) – a list of Index objects (or a single index object) to use as the key of the database.
- **value_parts** (*list*) – a list of Index objects (or a single index object) to use for the values stored in the database.

The schema defines the structure of the keys and values for a given *Database*. They can be comprised of a single index-type or multiple indexes for composite keys or values.

Example:

```
# Simple schema defining text keys and values.
simple = Schema(StringIndex('key'), StringIndex('value'))

# Schema with composite key for storing timestamps and event-types,
# along with msgpack-serialized data as the value.
event_schema = Schema(
    [U64Index('timestamp'), StringIndex('type')],
    [MsgPackIndex('value')])
```

Schemas are used when adding databases using the *Sophia.add_database()* method.

add_key (*index*)

Parameters **index** (BaseIndex) – an index object to add to the key parts.

Add an index to the key. Allows *Schema* to be built-up programmatically.

add_value (*index*)

Parameters **index** (BaseIndex) – an index object to add to the value parts.

Add an index to the value. Allows *Schema* to be built-up programmatically.

classmethod key_value ()

Short-hand for creating a simple text schema consisting of a single *StringIndex* for both the key and the value.

class BaseIndex (*name*)

Parameters **name** (*str*) – Name for the key- or value-part the index represents.

Indexes are used to define the key and value portions of a *Schema*. Traditional key/value databases typically only supported a single-value, single-datatype key and value (usually bytes). Sophia is different in that keys or values can be comprised of multiple parts with differing data-types.

For example, to emulate a typical key/value store:

```
schema = Schema([BytesIndex('key')], [BytesIndex('value')])
db = env.add_database('old_school', schema)
```

Suppose we are storing time-series event logs. We could use a 64-bit integer for the timestamp (in micro-seconds) as well as a key to denote the event-type. The value could be arbitrary msgpack-encoded data:

```
key = [U64Index('timestamp'), StringIndex('type')]
value = [MsgPackIndex('value')]
events = env.add_database('events', Schema(key, value))
```

class SerializedIndex (*name*, *serialize*, *deserialize*)

Parameters

- **name** (*str*) – Name for the key- or value-part the index represents.

- **serialize** – a callable that accepts data and returns bytes.
- **deserialize** – a callable that accepts bytes and deserializes the data.

The *SerializedIndex* can be used to transparently store data as bytestrings. For example, you could use a library like `msgpack` or `pickle` to transparently store and retrieve Python objects in the database:

```
key = StringIndex('key')
value = SerializedIndex('value', pickle.dumps, pickle.loads)
pickled_db = env.add_database('data', Schema([key], [value]))
```

Note: `sophy` already provides indexes for *JsonIndex*, *MsgPackIndex* and *PickleIndex*.

class BytesIndex(*name*)
Store arbitrary binary data in the database.

class StringIndex(*name*)
Store text data in the database as UTF8-encoded bytestrings. When reading from a *StringIndex*, data is decoded and returned as unicode.

class JsonIndex(*name*)
Store data as UTF8-encoded JSON. Python objects will be transparently serialized and deserialized when writing and reading, respectively.

class MsgPackIndex(*name*)
Store data using the msgpack serialization format. Python objects will be transparently serialized and deserialized when writing and reading.

Note: Requires the `msgpack-python` library.

class PickleIndex(*name*)
Store data using Python's pickle serialization format. Python objects will be transparently serialized and deserialized when writing and reading.

class UUIDIndex(*name*)
Store UUIDs. Python `uuid.UUID()` objects will be stored as raw bytes and decoded to `uuid.UUID()` instances upon retrieval.

class U64Index(*name*)

class U32Index(*name*)

class U16Index(*name*)

class U8Index(*name*)
Store unsigned integers of the given sizes.

class U64RevIndex(*name*)

class U32RevIndex(*name*)

class U16RevIndex(*name*)

class U8RevIndex(*name*)
Store unsigned integers of the given sizes in reverse order.

3.5 Cursor

class Cursor
Cursor handle for a *Database*. This object is not created directly but through the *Database.cursor()* method or one of the database methods that returns a row iterator (e.g. *Database.items()*).

Cursors are iterable and, depending how they were configured, can return keys, values or key/value pairs.

3.6 Settings

Sophia supports a wide range of settings and configuration options. These settings are also documented in the Sophia documentation.

3.6.1 Environment settings

The following settings are available as properties on *Sophia*:

Setting	Type	Description
version	string, ro	Get current Sophia version
version_storage	string, ro	Get current Sophia storage version
build	string, ro	Get git commit hash of build
status	string, ro	Get environment status (eg online)
errors	int, ro	Get number of errors
error	string, ro	Get last error description
path	string, ro	Get current Sophia environment directory
Backups		
backup_path	string	Set backup path
backup_run	method	Start backup in background (non-blocking)
backup_active	int, ro	Show if backup is running
backup_last	int, ro	Show ID of last-completed backup
backup_last_complete	int, ro	Show if last backup succeeded
Scheduler		
scheduler_threads	int	Get or set number of worker threads
scheduler_trace(thread_id)	method	Get a worker trace for given thread
Transaction Manager		
transaction_online_rw	int, ro	Number of active read/write transactions
transaction_online_ro	int, ro	Number of active read-only transactions
transaction_commit	int, ro	Total number of completed transactions
transaction_rollback	int, ro	Total number of transaction rollbacks
transaction_conflict	int, ro	Total number of transaction conflicts
transaction_lock	int, ro	Total number of transaction locks
transaction_latency	string, ro	Average transaction latency from start to end
transaction_log	string, ro	Average transaction log length
transaction_vlsn	int, ro	Current VLSN
transaction_gc	int, ro	SSI GC queue size
Metrics		
metric_lsn	int, ro	Current log sequential number
metric_tsn	int, ro	Current transaction sequential number
metric_nsn	int, ro	Current node sequential number
metric_dsn	int, ro	Current database sequential number
metric_bsn	int, ro	Current backup sequential number
metric_lfsn	int, ro	Current log file sequential number
Write-ahead Log		
log_enable	int	Enable or disable transaction log
log_path	string	Get or set folder for log directory

Continued on next page

Table 1 – continued from previous page

Setting	Type	Description
log_sync	int	Sync transaction log on every commit
log_rotate_wm	int	Create a new log after "rotate_wm" updates
log_rotate_sync	int	Sync log file on every rotation
log_rotate	method	Force Sophia to rotate log file
log_gc	method	Force Sophia to garbage-collect log file pool
log_files	int, ro	Number of log files in the pool

3.6.2 Database settings

The following settings are available as properties on *Database*. By default, Sophia uses `pread(2)` to read from disk. When `mmap`-mode is on (by default), Sophia handles all requests by directly accessing memory-mapped node files.

Setting	Type	Description
database_name	string, ro	Get database name
database_id	int, ro	Database sequential ID
database_path	string, ro	Directory for storing data
mmap	int	Enable or disable mmap-mode
direct_io	int	Enable or disable `O_DIRECT` mode.
sync	int	Sync node file on compaction completion
expire	int	Enable or disable key expiration
compression	string	Specify compression type: lz4, zstd, none (default)
limit_key	int, ro	Scheme key size limit
limit_field	int	Scheme field size limit
Index		
index_memory_used	int, ro	Memory used by database for in-memory key indexes
index_size	int, ro	Sum of nodes size in bytes (e.g. database size)
index_size_uncompressed	int, ro	Full database size before compression
index_count	int, ro	Total number of keys in db, includes unmerged dupes
index_count_dup	int, ro	Total number of transactional duplicates
index_read_disk	int, ro	Number of disk reads since start
index_read_cache	int, ro	Number of cache reads since start
index_node_count	int, ro	Number of active nodes
index_page_count	int, ro	Total number of pages
Compaction		
compaction_cache	int	Total write cache size used for compaction
compaction_checkpoint	int	
compaction_node_size	int	Set a node file size in bytes.
compaction_page_size	int	Set size of page
compaction_page_checksum	int	Validate checksum during compaction
compaction_expire_period	int	Run expire check process every `N` seconds
compaction_gc_wm	int	GC starts when watermark value reaches `N` dupes
compaction_gc_period	int	Check for a gc every `N` seconds
Performance		
stat_documents_used	int, ro	Memory used by allocated document
stat_documents	int, ro	Number of currently allocated documents
stat_field	string, ro	Average field size
stat_set	int, ro	Total number of Set operations

Continued on next page

Table 2 – continued from previous page

Setting	Type	Description
stat_set_latency	string, ro	Average Set latency
stat_delete	int, ro	Total number of Delete operations
stat_delete_latency	string, ro	Average Delete latency
stat_get	int, ro	Total number of Get operations
stat_get_latency	string, ro	Average Get latency
stat_get_read_disk	string, ro	Average disk reads by Get operation
stat_get_read_cache	string, ro	Average cache reads by Get operation
stat_pread	int, ro	Total number of pread operations
stat_pread_latency	string, ro	Average pread latency
stat_cursor	int, ro	Total number of cursor operations
stat_cursor_latency	string, ro	Average cursor latency
stat_cursor_read_disk	string, ro	Average disk reads by Cursor operation
stat_cursor_read_cache	string, ro	Average cache reads by Cursor operation
stat_cursor_ops	string, io	Average number of keys read by Cursor operation
Scheduler		
scheduler_gc	int, ro	Show if GC operation is in progress
scheduler_expire	int, ro	Show if expire operation is in progress
scheduler_backup	int, ro	Show if backup operation is in progress
scheduler_checkpoint	int, ro	

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

B

C

D

E

G

I

J

K

M

O

P

R

S

T

U

V