sostore Documentation

Release 0.4

Jeffrey Armstrong

June 14, 2014

Contents

sostore, or the SQLite Object Store, is a straightforward storage engine for storing and retrieving dictionaries from an SQLite database. Much of the terminology is taken from MongoDB as this engine was originally designed to replace a PyMongo implementation.

This library may seem trivial because it very much is. However, some others may need a super-lightweight dictionary store. Linking of objects within the database is not supported unless explicitly handled by the developer.

sostore is almost certainly not performant. It can be thread-safe as long as `Collection` objects aren't passed around between threads as they contain `sqlite3.Connection` objects.

CHAPTER 1

Requirements

sostore should work with Python 2.6 and higher, including Python 3.x. Because it uses only SQLite, there are no further requirements.

CHAPTER 2

Using sostore

sostore works with `Collection` objects for storing and retrieving dictionaries. A `Collection` is instantiated in the following manner:

```
>>> import sostore
>>> collection = sostore.Collection("peoples")
>>>
```

The above will instantiate an in-memory connection. For a more permanent solution, the database (a file on disk) can alternatively be specified:

```
>>> import sostore
>>> collection = sostore.Collection("peoples", db="balance.db")
>>>
```

Because sostore uses SQLite as a backend, you can also specify an existing connection to an SQLite database rather than a database name:

```
>>> import sostore
>>> import sqlite3
>>> con = sqlite3.connect(":memory:")
>>> collection = sostore.Collection("peoples", connection=con)
>>>
```

The above can be useful for reusing database connections. Note, however, that connections (for SQLite at least) cannot be shared across threads.

Finally, the database's dictionary identifiers are normally chosen by SQLite's automatic, sequential assignment mechanism. Alternatively, you can specify random identifiers (non-sequential) if necessary:

```
>>> import sostore
>>> collection = sostore.Collection("peoples", randomized=True)
>>>
```

2.1 Inserting

Inserting a dictionary into a collection is relatively simple. All ids of dictionaries are automatically generated.

```
>>> import sostore
>>> collection = sostore.Collection("peoples")
>>> d = {"name": "Margaux LaFleur", "hair color":"black"}
>>> d = collection.insert(d)
>>>
```

The returned dictionary will have a new key/value pair with the key *_id*. Depending on options passed into the `Collection` constructor, the *_id* is either sequentially chosen by SQLite or randomly selected.

Dictionaries can be inserted once and only once. If a dictionary has already been inserted, a `ValueError` will be thrown.

2.2 Retrieval

If an id of a dictionary is known, retrieval is rather simple:

```
>>> d = collection.get(1)
>>> print(d)
{'_id': 1, 'name': 'Margaux LaFleur', 'hair color': 'black'}
>>>
```

If an id doesn't exist in the database, this method will simply return `None` without any errors.

Multiple dictionaries can be retrieved in one call via a list of ids:

```
>>> a = collection.get_many((1, 2))
>>> print(a)
[{'_id': 1, 'name': 'Margaux LaFleur', 'hair color': 'black'},  {'_id': 2, 'name': 'Henry McCallum'}]
>>>
```

The fields returned can be restricted when retrieving multiple dictionaries if desired:

```
>>> a = collection.get_many((1, 2), fields=('name',))
>>> print(a)
[{'_id': 1, 'name': 'Margaux LaFleur'},  {'_id': 2, 'name': 'Henry McCallum'}]
>>>
```

All dictionaries can also be retrieved if desired:

```
>>> d = collection.all()
>>> print(d)
[{'_id': 1, 'name': 'Margaux LaFleur', 'hair color': 'black'},  {'_id': 2, 'name': 'Henry McCallum'}]
>>>
```

Similarly, the fields can be restricted in this call as well:

```
>>> d = collection.all(fields=('name',))
>>> print(d)
[{'_id': 1, 'name': 'Margaux LaFleur'},  {'_id': 2, 'name': 'Henry McCallum'}]
>>>
```

2.3 Retrieval by Field

A dictionary can also be retrieved by a known field. The `find_one` method accepts a key and a value for which to search, returning the first matching dictionary or `None` if no matches exist:

```
>>> d = collection.find_one("name", "Margaux LaFleur")
>>> print(d)
{'_id': 1, 'name': 'Margaux LaFleur', 'hair color': 'black'}
>>> d = collection.find_one("occupation", "magician")
>>> print(d)
None
>>>
```

Because the dictionaries are schemaless, keys that don't exist in any dictionary can be searched for without errors being thrown.

Similarly, the `find` method performs a similar, but it returns an array of matching dictionaries:

```
>>> d = collection.find("name", "Margaux LaFleur")
>>> print(d)
[{'_id': 1, 'name': 'Margaux LaFleur', 'hair color': 'black'}]
>>>
```

Finally, in either `find` method, you may restrict the fields returned. For example. if only the name is of interest, the following command can be used:

```
>>> d = collection.find_one("name", "Margaux LaFleur", fields=('hair color',))
>>> print(d)
{'_id': 1, 'hair color': 'black'}
>>>
```

2.4 Random Retrieval

In some specific cases, it may be advantageous to retreive a random entry or entries from a collection. To retrieve one random entry, simply call:

```
>>> d = collection.random_entry()
>>> print(d)
{'_id': 7, 'name': 'Erin', 'magic': False}
>>>
```

Multiple random entries can be retrieved in a list as well:

```
>>> d = collection.random_entries(count=2)
>>> print(d)
[{'_id': 7, 'name': 'Erin', 'magic': False}, {'_id':13, 'name': 'Stephen', 'occupation': 'inn keeper'
>>>
```

2.5 Updating

Existing stored dictionaries can be easily updated. In the example below, a dictionary is retrieved, a field is added, and the stored dictionary is updated:

```
>>> d = collection.get(1)
>>> d['occupation'] = "witch"
>>> collection.update(d)
>>>
```

If a dictionary has not yet been stored, the method will raise a `ValueError`.

2.6 Cleanup

Once work with a `Collection` is complete, the `done` method should be called to close any SQLite connections if necessary.

```
>>> collection.done()
>>>
```

If a connection was specified in the constructor rather than a database name, this method will close the connection regardless. It performs no other tasks at this time.

CHAPTER 3

Behind the Scenes

sostore uses a ridiculously simple and naive backend. Each `Collection` the user creates generates a new table in the database with the name of that `Collection` as the name of the table. The table will have two columns, "_id" and "_data." The names of these columns, however, are not particularly important to the user.

The "_id" column is an auto-incrementing primary key (unless a random id setting is enabled). The "_data" column is a text blob that contains the JSON-ified Python dictionary to be stored *without the _id key*. The "_id" key is always removed on store and added back into the dictionary upon retrieval to ensure consistency.

The SQLite commands within this library use safe prepare statements and, therefore, can be assumed safe from SQLite injection attacks. However, the `Collection` names are *not* safe. You should not allow dirty `Collection` names to be specified under any circumstance.

That's about all there is to it.

CHAPTER 4

Licensing

CHAPTER 5

Indices and tables

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