openleveldb

Release 0.1.5

Luca Moschella

Apr 15, 2021

CONTENTS

Openleveldb is a small pythonic wrapper around Plyvel

CHAPTER

ONE

FEATURES

1.1 Transparent object store

It works with python objects:

- Automatically **encodes objects** into bytes when saving to leveldb
- Automatically **decodes bytes** into their original type when retrieving objects from leveldb

Supported types include:

- `int`
- `str`
- `numpy.ndarray`
- Anything that is serializable by orjson

```
>>> db['key'] = {'key': [1, 2, 3]}
>>> db['key']
{'key': [1, 2, 3]}
```

1.2 Python dict-like protocol

It offers dict-like interface to LevelDB

```
>>> db["prefix", "key"] = np.array([1, 2, 3], dtype=np.int8)
>>> db["prefix", "key"]
array([1, 2, 3], dtype=int8)
```

```
>>> db = db["prefix", ...]
>>> db["key"]
array([1, 2, 3], dtype=int8)
```

1.3 String-only keys

The only possible type for the keys is `str`. It avoids several problems when working with prefixes.

1.4 Multiprocessing support

Experimental **multiprocessing** support using a background flask server, exposing the same API of a direct connection:

```
db = LevelDB(db_path="path_to_db", server_address="http://127.0.0.1:5000")
```

1.4.1 Installation

It is possible to install `openleveldb` with poetry:

```
poetry add openleveldb
```

or with pip:

```
pip install openleveldb
```

Verify installation

Verify that the installation has been successful and that `plyvel` correctly installed leveldb, if it is not already installed on the system:

```
python -c 'import openleveldb'
```

Verify that `openleveldb` using the tests

```
git clone git@github.com:lucmos/openleveldb.git
cd openleveldb
poetry run pytest .
```

1.4.2 Getting started

Open db connection

The connection to the db can be direct or pass through a REST server. The only change required in the code is how the `LevelDB` object is instantiated

Direct connection

The first thing to do is to instantiate a `LevelDB` object to open a connection to leveldb database:

```
from openleveldb import LevelDB
db = LevelDB(db_path="path_to_db")
```

REST connection

If it's required to have multiprocessing support, that is not provided by leveldb, it is possible to start a server and connect to the database through REST API. In order to start the server is enough to do:

```
cd openleveldb
make server
```

Then it's possible to instantiate a `LevelDB` object specifying the server:

```
from openleveldb import LevelDB
db = LevelDB(db_path="path_to_db", server_address="http://127.0.0.1:5000")
```

Basic access

Storing, reading and deleting an element follow the dict protocol:

```
>>> db["prefix", "key"] = np.array([1, 2, 3], dtype=np.int8)
>>> db["prefix", "key"]
array([1, 2, 3], dtype=int8)
>>> del db["prefix", "key"]
```

It is possible to use an arbitrary number of prefixes:

```
>>> db["prefix1", "prefix2", "key"] = np.array([1, 2, 3], dtype=np.int8)
>>> db["prefix1", "prefix2", "key"]
array([1, 2, 3], dtype=int8)
>>> del db["prefix1", "prefix2", "key"]
```

Iteration

Iteration over `(key, value)` pairs behaves accordingly:

```
>>> list(db)
[('a1', 'value1'), ('b1', 'value2'), ('b2', 'value3'), ('c1', 'value4')]
```

It's possible to perform advanced form of iteration using the `LevelDB.prefixed_iter` function:

```
>>> list(db)
[('a1', 'value1'), ('b1', 'value2'), ('b2', 'value3'), ('c1', 'value4')]
>>> list(db.prefixed_iter(prefixes=["b"]))
[('1', 'value2'), ('2', 'value3')]
>>> list(db.prefixed_iter(prefixes=["b", "1"]))
[('', 'value2')]
>>> list(db.prefixed_iter(starting_by="b"))
```

(continues on next page)

(continued from previous page)

```
[('b1', 'value2'), ('b2', 'value3')]
>>> list(db.prefixed_iter(starting_by=["b", "1"]))
[('b1', 'value2')]
```

Fancy indexing

When a local connection is available, it is possible to use fancy indexing to obtain a stateful `LevelDB` that remembers the prefixes:

```
>>> list(db)
[('a1', 'value1'), ('b1', 'value2'), ('b2', 'value3'), ('c1', 'value4')]
>>> db_b = db['b', ...]
>>> db_b["1"]
'value2'
>>> list(db_b)
[('1', 'value2'), ('2', 'value3')]
>>> list(db["c", ...])
[('1', 'value4')]
```

1.4.3 Public API

class openleveldb.**LevelDB**(*db_path: Optional[Union[str, pathlib.Path]]*, *server_address: Optional[str] = None*, *dbconnector: Optional[Union[openleveldb.backend.connectorlocal.LevelDBLocal, openleveldb.backend.connectorclient.LevelDBClient]] = None*, *read_only: bool = False*)

__init__(*db_path: Optional[Union[str, pathlib.Path]]*, *server_address: Optional[str] = None*, *dbconnector: Optional[Union[openleveldb.backend.connectorlocal.LevelDBLocal, openleveldb.backend.connectorclient.LevelDBClient]] = None*, *read_only: bool = False*) → None

Provide access to a leveldb database, it if does not exists one is created.

Local databases do not support multiprocessing. It is possible to access a local db with:

```
>>> db = LevelDB(db_path)
```

Remote databases support multiprocessing. Once the a leveldb server is running, it is possible to access the remote db with:

```
>>> # db = LevelDB(remote_db_path, server_address)
```

Parameters

- **db_path** – the path in the filesystem to the database
- **server_address** – the address of the remote server
- **read_only** – if true the db can not be modified
- **dbconnector** – provide directly an existing dbconnector

prefixed_iter(*prefixes: Optional[Union[str, Iterable[str]]] = None*, *starting_by: Optional[Union[str, Iterable[str]]] = None*, *include_key=True*, *include_value=True*) → Iterable

Builds a custom iterator.

The parameters `include_key` and `include_value` define what should be yielded:

```
>>> list(db)
[('a1', 'value1'), ('b1', 'value2'), ('b2', 'value3'), ('c1', 'value4')]
>>> list(db.prefixed_iter(include_key=False, include_value=False))
[None, None, None, None]
>>> list(db.prefixed_iter(include_key=True, include_value=False))
['a1', 'b1', 'b2', 'c1']
>>> list(db.prefixed_iter(include_key=False, include_value=True))
['value1', 'value2', 'value3', 'value4']
```

The `prefixes` and `starting_by` parameters have a similar meaning. They determine over which keys it should iterate. The difference is that `starting by` preserves the prefix in the returned key. The iterations stops when all the available keys with the given prefixes have been yielded

```
>>> list(db)
[('a1', 'value1'), ('b1', 'value2'), ('b2', 'value3'), ('c1', 'value4')]
>>> list(db.prefixed_iter(prefixes=["b"]))
[('1', 'value2'), ('2', 'value3')]
>>> list(db.prefixed_iter(prefixes=["b", "1"]))
[('', 'value2')]
>>> list(db.prefixed_iter(starting_by="b"))
[('b1', 'value2'), ('b2', 'value3')]
>>> list(db.prefixed_iter(starting_by=["b", "1"]))
[('b1', 'value2')]
```

Parameters

- **include_key** – if False do not yield the keys
- **include_value** – if False do not yield the values
- **prefixes** – prefixes of the desired keys The prefixes will be removed from the keys returned
- **starting_by** – prefixes of the desired keys The prefixes will be preserved from the keys returned

Returns the iterable over the keys and/or values

prefixed_len(*prefixes: Optional[Union[str, Iterable[str]]] = None*, *starting_by: Optional[str] = None*) → int

Utility function to compute the number of keys with a given prefix, see :py:meth:~database.LevelDB.prefixed_iter for more details.

```
>>> list(db)
[('a1', 'value1'), ('b1', 'value2'), ('b2', 'value3'), ('c1', 'value4')]
>>> db.prefixed_len(prefixes=["b"])
2
>>> db.prefixed_len(prefixes=["b", "1"])
1
```

Parameters

- **prefixes** – prefixes of the desired keys The prefixes will be removed from the keys returned
- **starting_by** – prefixes of the desired keys The prefixes will be preserved from the keys returned

Returns the number of matching keys

__iter__() → Iterator
Iterator over (key, value) sorted by key

```
>>> list(db)
[('a1', 'value1'), ('b1', 'value2'), ('b2', 'value3'), ('c1', 'value4')]
```

Returns the iterator over the items

__len__() → int
Computes the number of element in the database.

```
>>> list(db)
[('a1', 'value1'), ('b1', 'value2'), ('b2', 'value3'), ('c1', 'value4')]
>>> len(db)
4
```

Returns number of elements in the database

__setitem__(*key: Union[str, Iterable[str]]*, *value: Any*) → None
Store the couple (key, value) in leveldb. The key and the value are automatically encoded together with the obj type information, in order to be able to automate the decoding.

```
>>> import numpy as np
>>> db["array"] = np.array([1, 2, 3], dtype=np.int8)
>>> db["array"]
array([1, 2, 3], dtype=int8)
>>> del db["array"]
```

The key may be one or more strings to specify prefixes. The last element is always the key:

```
>>> db["prefix1", "prefix2", "key"] = "myvalue"
>>> db["prefix1", "prefix2", "key"]
'myvalue'
>>> del db["prefix1", "prefix2", "key"]
```

Parameters **key** – one or more strings to specify prefixes

Returns the value associated to the key in leveldb

__getitem__(*key: Union[str, Iterable[Union[str, ellipsis]]]*) → Any
Retrieve the couple (key, value) from leveldb.

```
>>> db["a1"]
'value1'
>>> db["a", "1"]
'value1'
```

The value is automatically decoded into its original type. The value must have been stored with :py:~:py:meth:~database.LevelDB.__setitem__

```
>>> import numpy as np
>>> db["array"] = np.array([1, 2, 3], dtype=np.int8)
>>> db["array"]
array([1, 2, 3], dtype=int8)
>>> del db["array"]
```

The key may be one or more strings, to specify prefixes. The last element is always the key:

```
>>> db["prefix1", "prefix2", "key"] = "myvalue"
>>> db["prefix1", "prefix2", "key"]
'myvalue'
>>> del db["prefix1", "prefix2", "key"]
```

It is possible to retrieve a stateful instance of :py:class:~database.LevelDB that accounts for prefixes using Ellipsis as key:

```
>>> list(db)
[('a1', 'value1'), ('b1', 'value2'), ('b2', 'value3'), ('c1', 'value4')]
>>> db_b = db['b', ...]
>>> db_b["1"]
'value2'
>>> list(db_b)
[('1', 'value2'), ('2', 'value3')]
>>> list(db["c", ...])
[('1', 'value4')]
```

Parameters **key** – one or more strings to specify prefixes. It's possible to specify sub-db using the Ellipsis as key.

Returns the value associated to the key in leveldb or a sub-db.

__delitem__(*key: Union[str, Iterable[str]]*) → None

Delete the couple (key, value) from leveldb.

The key may be one or more strings, to specify prefixes. The last element is always the key:

```
>>> db["prefix", "key"] = "value"
>>> db["prefix", "key"]
'value'
>>> db["prefixkey"]
'value'
>>> del db["prefix", "key"]
>>> print(db["prefix", "key"])
None
>>> print(db["prefixkey"])
None
```

Parameters **key** – one or more strings to specify prefixes.

close() → None

Close the database

INDEX

Symbols

C

L

P