lsm-db Documentation

Release 0.1.0

Charles Leifer

Jul 18, 2023

Contents

python lsm-db

fast key/value store using sqlite4

Fast Python bindings for SQLite's LSM key/value store. The LSM storage engine was initially written as part of the experimental SQLite4 rewrite (now abandoned). More recently, the LSM source code was moved into the SQLite3 source tree and has seen some improvements and fixes. This project uses the LSM code from the SQLite3 source tree.

Features:

- Embedded zero-conf database.
- Keys support in-order traversal using cursors.
- Transactional (including nested transactions).
- Single writer/multiple reader MVCC based transactional concurrency model.
- On-disk database stored in a single file.
- Data is durable in the face of application or power failure.
- Thread-safe.
- Python 2.x and 3.x.

Limitations:

- Not tested on Windoze.

The source for Python lsm-db is hosted on GitHub.

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

Contents:

CHAPTER 1

Installation

You can use `pip` to install `lsm-db`:

```
pip install lsm-db
```

The project is hosted at https://github.com/coleifer/python-lsm-db and can be installed from source:

```
git clone https://github.com/coleifer/python-lsm-db
cd lsm-db
python setup.py build
python setup.py install
```

Note: `lsm-db` depends on Cython to generate the Python extension. By default, lsm-db ships with a pre-generated C source file, so it is not strictly necessary to install Cython in order to compile `lsm-db`, but you may wish to install Cython to ensure the generated source is compatible with your setup.

After installing lsm-db, you can run the unit tests by executing the `tests` module:

```
python tests.py
```

CHAPTER 2

Quick-start

Below is a sample interactive console session designed to show some of the basic features and functionality of the `lsm-db` Python library. Also check out the *API documentation*.

To begin, instantiate a `lsm.LSM` object, specifying a path to a database file.

```
>>> from lsm import LSM
>>> db = LSM('test.ldb')
```

2.1 Key/Value Features

`lsm-db` is a key/value store, and has a dictionary-like API:

```
>>> db['foo'] = 'bar'
>>> print db['foo']
bar

>>> for i in range(4):
...     db['k%s' % i] = str(i)
...

>>> 'k3' in db
True
>>> 'k4' in db
False

>>> del db['k3']
>>> db['k3']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "lsm.pyx", line 973, in lsm.LSM.__getitem__ (lsm.c:7142)
  File "lsm.pyx", line 777, in lsm.LSM.fetch (lsm.c:5756)
  File "lsm.pyx", line 778, in lsm.LSM.fetch (lsm.c:5679)
```

(continues on next page)

(continued from previous page)

```
  File "lsm.pyx", line 1289, in lsm.Cursor.seek (lsm.c:12122)
  File "lsm.pyx", line 1311, in lsm.Cursor.seek (lsm.c:12008)
KeyError: 'k3'
```

By default when you attempt to look up a key, `lsm-db` will search for an exact match. You can also search for the closest key, if the specific key you are searching for does not exist:

```
>>> from lsm import SEEK_LE, SEEK_GE
>>> db['k1xx', SEEK_LE]  # Here we will match "k1".
'1'
>>> db['k1xx', SEEK_GE]  # Here we will match "k2".
'2'
```

`LSM` supports other common dictionary methods such as:

- `keys()`
- `values()`
- `update()`

2.2 Slices and Iteration

The database can be iterated through directly, or sliced. When you are slicing the database the start and end keys need not exist – `lsm-db` will find the closest key (details can be found in the `fetch()` documentation).

```
>>> [item for item in db]
[('foo', 'bar'), ('k0', '0'), ('k1', '1'), ('k2', '2')]

>>> db['k0':'k99']
<generator object at 0x7f2ae93072f8>

>>> list(db['k0':'k99'])
[('k0', '0'), ('k1', '1'), ('k2', '2')]
```

You can use open-ended slices. If the lower- or upper-bound is outside the range of keys an empty list is returned.

```
>>> list(db['k0':])
[('k0', '0'), ('k1', '1'), ('k2', '2')]

>>> list(db[:'k1'])
[('foo', 'bar'), ('k0', '0'), ('k1', '1')]

>>> list(db[:'aaa'])
[]
```

To retrieve keys in reverse order, simply use a higher key as the first parameter of your slice. If you are retrieving an open-ended slice, you can specify `True` as the `step` parameter of the slice.

```
>>> list(db['k1':'aaa'])  # Since 'k1' > 'aaa', keys are retrieved in reverse:
[('k1', '1'), ('k0', '0'), ('foo', 'bar')]

>>> list(db['k1'::True])  # Open-ended slices specify True for step:
[('k1', '1'), ('k0', '0'), ('foo', 'bar')]
```

You can also **delete** slices of keys, but note that the delete **will not** include the keys themselves:

```
>>> del db['k0':'k99']

>>> list(db)  # Note that 'k0' still exists.
[('foo', 'bar'), ('k0', '0')]
```

2.3 Cursors

While slicing may cover most use-cases, for finer-grained control you can use cursors for traversing records.

```
>>> with db.cursor() as cursor:
...     for key, value in cursor:
...         print key, '=>', value
...
foo => bar
k0 => 0

>>> db.update({'k1': '1', 'k2': '2', 'k3': '3'})

>>> with db.cursor() as cursor:
...     cursor.first()
...     print cursor.key()
...     cursor.last()
...     print cursor.key()
...     cursor.previous()
...     print cursor.key()
...
foo
k3
k2

>>> with db.cursor() as cursor:
...     cursor.seek('k0', SEEK_GE)
...     print list(cursor.fetch_until('k99'))
...
[('k0', '0'), ('k1', '1'), ('k2', '2'), ('k3', '3')]
```

Note: It is very important to close a cursor when you are through using it. For this reason, it is recommended you use the `cursor()` context-manager, which ensures the cursor is closed properly.

2.4 Transactions

`lsm-db` supports nested transactions. The simplest way to use transactions is with the `transaction()` method, which doubles as a context-manager or decorator.

```
>>> with db.transaction() as txn:
...     db['k1'] = '1-mod'
...     with db.transaction() as txn2:
...         db['k2'] = '2-mod'
...         txn2.rollback()
...
```

(continues on next page)

(continued from previous page)

```
True
>>> print db['k1'], db['k2']
1-mod 2
```

You can commit or roll-back transactions part-way through a wrapped block:

```
>>> with db.transaction() as txn:
...     db['k1'] = 'outer txn'
...     txn.commit()  # The write is preserved.
...
...     db['k1'] = 'outer txn-2'
...     with db.transaction() as txn2:
...         db['k1'] = 'inner-txn'  # This is commited after the block ends.
...     print db['k1']  # Prints "inner-txn".
...     txn.rollback()  # Rolls back both the changes from txn2 and the preceding␣
↪write.
...     print db['k1']
...
1              <- Return value from call to commit().
inner-txn      <- Printed after end of txn2.
True           <- Return value of call to rollback().
outer txn      <- Printed after rollback.
```

If you like, you can also explicitly call `begin()`, `commit()`, and `rollback()`:

```
>>> db.begin()
>>> db['foo'] = 'baze'
>>> print db['foo']
baze
>>> db.rollback()
True
>>> print db['foo']
bar
```

CHAPTER 3

API Documentation

3.1 Constants

Seek methods, can be used when fetching records or slices.

`SEEK_EQ` The cursor is left at EOF (invalidated). A call to lsm_csr_valid() returns non-zero.

`SEEK_LE` The cursor is left pointing to the largest key in the database that is smaller than (pKey/nKey). If the database contains no keys smaller than (pKey/nKey), the cursor is left at EOF.

`SEEK_GE` The cursor is left pointing to the smallest key in the database that is larger than (pKey/nKey). If the database contains no keys larger than (pKey/nKey), the cursor is left at EOF.

If the fourth parameter is `SEEK_LEFAST`, this function searches the database in a similar manner to `SEEK_LE`, with two differences:

Even if a key can be found (the cursor is not left at EOF), the lsm_csr_value() function may not be used (attempts to do so return LSM_MISUSE).

The key that the cursor is left pointing to may be one that has been recently deleted from the database. In this case it is guaranteed that the returned key is larger than any key currently in the database that is less than or equal to (pKey/nKey).

`SEEK_LEFAST` requests are intended to be used to allocate database keys.

Used in calls to `LSM.set_safety()`.

- `SAFETY_OFF`
- `SAFETY_NORMAL`
- `SAFETY_FULL`

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index