lsm-db Documentation

Release 0.1.0

Charles Leifer

February 07, 2017

Contents

1 Installation **3**

2 Quick-start **5**
- 2.1 Key/Value Features . . . 5
- 2.2 Slices and Iteration . . . 6
- 2.3 Cursors . . . 6
- 2.4 Transactions . . . 7

3 API Documentation **9**
- 3.1 Constants . . . 18

4 Indices and tables **21**

Python Module Index **23**

Fast Python bindings for SQLite4's LSM key/value store.

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 and 3.

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)
  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()
...
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

class lsm.**LSM**

Python wrapper for SQLite4's LSM implementation.

http://www.sqlite.org/src4/doc/trunk/www/lsmapi.wiki

Optimizing database write throughput and responsiveness is done by configuring and scheduling work and checkpoint operations, and by configuring a few other parameters.

- `autocheckpoint`, default=2048 in KB, or 2MB

  Controls how often the database is checkpointed. Increasing this value to 8MB may improve overall write throughput.

- `autoflush`, default=1024 in KB, or 1MB

  Determines how much data, in KB, is allowed to accumulate in the live in-memory tree before the tree is marked as "old". The default, 1024K, may be increased to improve overall write throughput. Decreasing this value reduces memory usage.

- `automerge`, default=4 segments

  If auto-work is enabled, then this option is set to the number of segments that the library attempts to merge simultaneously. Increasing this value may reduce the total amount of data written to the database file. Decreasing it has the opposite effect and also decreases the average number of segments in the file, which may improve reads.

  The default value is 4, but may be set to any value between 2 and 8.

- `autowork`, enabled by default

  Let the database determine when to perform checkpoints, as a part of calls to insert(), delete(), or commit(). If set to 0 (false), then the application must schedule these operations.

- `mmap`, enabled by default on 64-bit systems

  If LSM is running on 64-bit system, mmap may be set to 1 or 0. On 32-bit systems mmap is always 0.

  If enabled, the entire database file is memory mapped. If false, data is accessed using the OS file primitives. Memory mapping can significantly improve the performance of read operations, as pages do not have to be copied from OS buffers into user space.

- `multiple_processes`, enabled by default

  If set to 0 (false) the library does not use file-locking primitives to lock the database, which speeds up transactions. This option is enabled by default.

- `write_safety`, default=1

This option determines how often the library pauses to wait for data written to the file-system to be synced. Since syncing is much slower than simply copying data into OS buffers, this option has a large effect on write performance. See set_write_safety() for more info.

- `transaction_log`, enabled by default

  This option determines whether the db will write changes to a log file. If disabled, writes will be faster but there is a chance for data loss in the event of application crash or power failure. Option is enabled by default.

The speed of database read operations is largely determined by the number of segments in the database file. So optimizing read operations is also linked to the configuring and scheduling of database write operations, as these policies determine the number of segments that are present in the database file at any time.

__contains__
Return a boolean indicating whether the given key exists.

__delitem__
Dictionary API wrapper for the *delete()* and *delete_range()* methods.

Parameters **key** – Either a string or a slice. Additionally, a second parameter can be supplied indicating what seek method to use.

Note: When deleting a range of keys, the start and end keys themselves are **not** deleted, only the intervening keys.

__enter__()
Use the database as a context manager. The database will be closed when the wrapped block exits.

__getitem__
Dictionary API wrapper for the *fetch()* and *fetch_range()* methods.

Parameters **key** – Either a string or a slice. Additionally, a second parameter can be supplied indicating what seek method to use.

Examples using single keys:

- `['charlie']`, search for the key *charlie*.
- `['2014.XXX', SEEK_LE]`, return the key whose value is equal to *2014.XXX*. If no such key exists, return the lowest key that **does not exceed** *2014.XXX*. If there is no lower key, then a `KeyError` will be raised.
- `['2014.XXX', SEEK_GE]`, return the key whose value is equal to *2014.XXX*. If no such key exists, return the greatest key that **does not precede** *2014.XXX*. If there is no higher key, then a `KeyError` will be raised.

Examples using slices (SEEK_LE and SEEK_GE cannot be used with slices):

- `['a':'z']`, return all keys from *a* to *z* in ascending order.
- `['z':'a']`, return all keys from *z* to *a* in reverse order.
- `['a':]`, return all key/value pairs from `a` on up.
- `[:'z']`, return all key/value pairs up to and including `z`.
- `['a'::True]`, return all key/value pairs from `a` on up in reverse order.

Note: When fetching slices, a `KeyError` will not be raised under any circumstances.

__init__

Parameters

- **filename** (*str*) – Path to database file.
- **open_database** (*bool*) – Whether to open the database automatically when the class is instantiated.
- **options** – Values for the various tunable options.

__iter__

Efficiently iterate through the items in the database. This method yields successive key/value pairs.

Note: The return value is a generator.

__reversed__()

Efficiently iterate through the items in the database in reverse order. This method yields successive key/value pairs.

__setitem__

Dictionary API wrapper for the *insert()* method.

begin()

Begin a transaction. Transactions can be nested.

Note: In most cases it is preferable to use the *transaction()* context manager/decorator.

checkpoint()

Write to the database file header. If the current snapshot has already been checkpointed, calling this function is a no-op. In this case if pnKB is not NULL, *nkb is set to 0. Or, if the current snapshot is successfully checkpointed by this function and pbKB is not NULL, *nkb is set to the number of bytes written to the database file since the previous checkpoint (the same measure as returned by the LSM_INFO_CHECKPOINT_SIZE query).

checkpoint_size()

The number of KB written to the database file since the most recent checkpoint.

close()

Close the database. If the database was already closed, this will return False, otherwise returns True on success.

Warning: You must close all cursors before attempting to close the db, otherwise an `IOError` will be raised.

commit()

Commit the inner-most transaction.

Returns Boolean indicating whether the changes were commited.

cursor()

Create a cursor and return it as a context manager. After the wrapped block, the cursor is closed.

Parameters **reverse** (*bool*) – Whether the cursor will iterate over keys in descending order.

Example:

```
with lsm_db.cursor() as cursor:
    for key, value in cursor.fetch_range('a', 'z'):
        # do something with data...

with lsm_db.cursor(reverse=True) as cursor:
    for key, value in cursor.fetch_range('z', 'a'):
        # data is now ordered descending order.
```

Note: In general the *cursor()* context manager should be used as it ensures cursors are properly cleaned up when you are done using them.

LSM databases cannot be closed as long as there are any open cursors, so it is very important to close them when finished.

delete()

Remove the specified key and value from the database. If the key does not exist, no exception is raised.

Note: You can delete keys using Python's dictionary API:

```
# These are equivalent:
lsm_db.delete('some-key')
del lsm_db['some-key']
```

delete_range()

Delete a range of keys, though the start and end keys themselves are not deleted.

Parameters

- **start** (*str*) – Beginning of range. This key is **not** removed.
- **end** (*str*) – End of range. This key is **not** removed.

Rather than using *delete_range()*, you can use Python's `del` keyword, specifying a slice of keys.

Example:

```
>>> for key in 'abcdef':
...     db[key] = key.upper()

>>> del db['a':'c']  # This will only delete 'b'.
>>> 'a' in db, 'b' in db, 'c' in db
(True, False, True)

>>> del db['0':'d']
>>> print list(db)
[('d', 'D'), ('e', 'E'), ('f', 'F')]
```

fetch()

Retrieve a value from the database.

Parameters

- **key** (*str*) – The key to retrieve.
- **seek_method** (*int*) – Instruct the database how to match the key.

Raises `KeyError` if a matching key cannot be found. See below for more details.

The following seek methods can be specified.

- **SEEK_EQ (default): match key based on equality. If no match is** found, then a `KeyError` is raised.
- **SEEK_LE: if the key does not exist, return the largest key in the** database that is *smaller* than the given key. If no smaller key exists, then a `KeyError` will be raised.
- **SEEK_GE: if the key does not exist, return the smallest key in** the database that is *larger* than the given key. If no larger key exists, then a `KeyError` will be raised.

Note: Instead of calling *fetch()*, you can simply treat your database like a dictionary.

Example:

```
# These are equivalent:
val = lsm_db.fetch('key')
val = lsm_db['key']

# You can specify the `seek_method` by passing a tuple:
val = lsm_db.fetch('other-key', SEEK_LE)
val = lsm_db['other-key', SEEK_LE]
```

---

fetch_range()

Fetch a range of keys, inclusive of both the start and end keys. If the start key is not specified, then the first key in the database will be used. If the end key is not specified, then all succeeding keys will be fetched.

If the start key is less than the end key, then the keys will be returned in ascending order. The logic for selecting the first and last key in the event either key is missing is such that:

- The start key will be the smallest key in the database that is larger than the given key (same as `SEEK_GE`).
- The end key will be the largest key in the database that is smaller than the given key (same as `SEEK_LE`).

If the start key is greater than the end key, then the keys will be returned in descending order. The logic for selecting the first and last key in the event either key is missing is such that:

- The start key will be the largest key in the database that is smaller than the given key (same as `SEEK_LE`).
- The end key will be the smallest key in the database that is larger than the given key (same as `SEEK_GE`).

Note: If one or both keys is `None` and you wish to fetch in reverse, you need to specify a third parameter, `reverse=True`.

Note: Rather than using *fetch_range()*, you can use the *__setitem__()* API and pass in a slice. The examples below will use the slice API.

Say we have the following data:

```
db.update({
  'a': 'A',
  'c': 'C',
  'd': 'D',
```

```
    'f': 'F',
})
```

Here are some example calls using ascending order:

```
>>> db['a':'d']
[('a', 'A'), ('c', 'C'), ('d', 'D')]

>>> db['a':'e']
[('a', 'A'), ('c', 'C'), ('d', 'D')]

>>> db['b':'e']
[('c', 'C'), ('d', 'D')]
```

If one of the boundaries is not specified (`None`), then it will start at the lowest or highest key, respectively.

```
>>> db[:'ccc']
[('a', 'A'), ('c', 'C')]

>>> db['ccc':]
[('d', 'D'), ('f', 'F')]

>>> db[:'x']
[('a', 'A'), ('c', 'C'), ('d', 'D'), ('f', 'F')]
```

If the start key is higher than the highest key, no results are returned.

```
>>> db['x':]
[]
```

If the end key is lower than the lowest key, no results are returned.

```
>>> db[:'0']
[]
```

Note: If the start key is greater than the end key, lsm-python will assume you want the range in reverse order.

Examples in descending (reverse) order:

```
>>> db['d':'a']
[('d', 'D'), ('c', 'C'), ('a', 'A')]

>>> db['e':'a']
[('d', 'D'), ('c', 'C'), ('a', 'A')]

>>> db['e':'b']
[('d', 'D'), ('c', 'C')]
```

If one of the boundaries is not specified (`None`), then it will start at the highest and lowest keys, respectively.

```
>>> db['ccc'::True]
[('c', 'C'), ('a', 'A')]

>>> db[:'ccc':True]
[('f', 'F'), ('d', 'D')]
```

```
>>> db['x'::True]
[('f', 'F'), ('d', 'D'), ('c', 'C'), ('a', 'A')]
```

If the end key is higher than the highest key, no results are returned.

```
>>> db[:'x':True]
[]
```

If the start key is lower than the lowest key, no results are returned.

```
>>> db['0'::True]
[]
```

flush()
Flush the in-memory tree to disk, creating a new segment.

The contents of an old in-memory tree may be written into the database file at any point. Once its contents have been written (or "flushed") to the database file, the in-memory tree may be discarded. Flushing an in-memory tree to the database file creates a new database "segment". A database segment is an immutable b-tree structure stored within the database file. A single database file may contain up to 64 segments.

At any point, two or more existing segments within the database file may be merged together into a single segment. Once their contents has been merged into the new segment, the original segments may be discarded.

After the set of segments in a database file has been modified (either by flushing an in-memory tree to disk or by merging existing segments together), the changes may be made persistent by "checkpointing" the database. Checkpointing involves updating the database file header and (usually) syncing the contents of the database file to disk.

insert()
Insert a key/value pair to the database. If the key exists, the previous value will be overwritten.

Note: Rather than calling *insert()*, you can simply treat your database as a dictionary and use the *__setitem__()* API:

```
# These are equivalent:
lsm_db.insert('key', 'value')
lsm_db['key'] = 'value'
```

---

keys()
Return a generator that successively yields the keys in the database.

Parameters **reverse** (*bool*) – Return the keys in reverse order.

Return type generator

open()
Open the database. If the database was already open, this will return False, otherwise returns True on success.

pages_read()
The number of 4KB pages read from the database file during the lifetime of this connection.

pages_written()
The number of 4KB pages written to the database file during the lifetime of this connection.

rollback()
Rollback the inner-most transaction. If *keep_transaction* is *True*, then the transaction will remain open after the changes were rolled back.

Parameters **keep_transaction** (*bool*) – Whether the transaction will remain open after the changes are rolled back (default=True).

Returns Boolean indicating whether the changes were rolled back.

transaction()

Create a context manager that runs the wrapped block in a transaction.

Example:

```
with lsm_db.transaction() as txn:
    lsm_db['k1'] = 'v1'

with lsm_db.transaction() as txn:
    lsm_db['k1'] = 'v1-1'
    txn.rollback()

assert lsm_db['k1'] == 'v1'
```

You can also use the *transaction()* method as a decorator. If the wrapped function returns normally, the transaction is committed, otherwise it is rolled back.

```
@lsm_db.transaction()
def transfer_funds(from_account, to_account, amount):
    # transfer money...
    return
```

tree_size()

At any time, there are either one or two tree structures held in shared memory that new database clients will access (there may also be additional tree structures being used by older clients - this API does not provide information on them). One tree structure - the current tree - is used to accumulate new data written to the database. The other tree structure - the old tree - is a read-only tree holding older data and may be flushed to disk at any time.

Assuming no error occurs, the location pointed to by the first of the two (int *) arguments is set to the size of the old in-memory tree in KB. The second is set to the size of the current, or live in-memory tree.

update()

Add an arbitrary number of key/value pairs. Unlike the Python `dict.update` method, *update()* does not accept arbitrary keyword arguments and only takes a single dictionary as the parameter.

Parameters **values** (*dict*) – A dictionary of key/value pairs.

values()

Return a generator that successively yields the values in the database. The values are **ordered based on their key**.

Parameters **reverse** (*bool*) – Return the values in reverse key-order.

Return type generator

work()

Explicitly perform work on the database structure.

If the database has an old in-memory tree when *work()* is called, it is flushed to disk. If this means that more than `nkb` of data is written to the database file, no further work is performed. Otherwise, the number of KB written is subtracted from nKB before proceeding.

Typically you will use `1` for the parameter in order to *optimize* the database.

Parameters **nkb** (*int*) – Limit on the number of KB of data that should be written to the database file before the call returns. It is a hint and is not honored strictly.

Returns The number of KB written to the database file.

Note: A background thread or process is ideal for running this method.

class `lsm.`**`Cursor`**

Wrapper around the *lsm_cursor* object.

Functions *`seek()`*, *`first()`*, and *`last()`* are *seek* functions. Whether or not *`next()`* and *`previous()`* may be called successfully depends on the most recent seek function called on the cursor. Specifically,

- At least one seek function must have been called on the cursor.
- To call `next()`, the most recent call to a seek function must have been either `first()` or a call to `seek()` specifying `SEEK_GE`.
- To call `previous()`, the most recent call to a seek function must have been either `last()` or a call to `seek()` specifying `SEEK_LE`.

Otherwise, if the above conditions are not met when `next()` or `previous()` is called, `LSM_MISUSE` is returned and the cursor position remains unchanged.

For more information, see:

http://www.sqlite.org/src4/doc/trunk/www/lsmusr.wiki#reading_from_a_database

`__enter__``()`
Expose the cursor as a context manager. After the wrapped block, the cursor will be closed, which is very important.

`__iter__`
Iterate from the cursor's current position. The iterator returns successive key/value pairs.

`compare``()`
Compare the given key with key at the cursor's current position.

`fetch_range``()`
Fetch a range of keys, inclusive of both the start and end keys. If the start key is not specified, then the first key will be used. If the end key is not specified, then all succeeding keys will be fetched.

For complete details, see the docstring for *`LSM.fetch_range()`*.

`fetch_until``()`
This method returns a generator that yields key/value pairs obtained by iterating from the cursor's current position until it reaches the given `key`.

`first``()`
Jump to the first key in the database.

`is_valid``()`
Return a boolean indicating whether the cursor is pointing at a valid record.

`keys``()`
Return a generator that successively yields keys.

`last``()`
Jump to the last key in the database.

`next`

`previous``()`
Move the cursor to the previous record. If no previous record exists, then a `StopIteration` will be raised.

If you encounter an Exception indicating *Misuse (21)* when calling this method, then you need to be sure that you are either calling *last()* or *seek()* with a seek method of SEEK_LE.

seek()
Seek to the given key using the specified matching method. If the operation did not find a valid key, then a KeyError will be raised.

- **SEEK_EQ (default): match key based on equality. If no match is** found then a KeyError is raised.
- **SEEK_LE: if the key does not exist, return the largest key in the** database that is *smaller* than the given key. If no smaller key exists, then a KeyError will be raised.
- **SEEK_GE: if the key does not exist, return the smallest key in** the database that is *larger* than the given key. If no larger key exists, then a KeyError will be raised.

For more details, read:

http://www.sqlite.org/src4/doc/trunk/www/lsmapi.wiki#lsm_csr_seek

values()
Return a generator that successively yields values.

class lsm.**Transaction**
Context manager and decorator to run the wrapped block in a transaction. LSM supports nested transactions, so the context manager/decorator can be mixed and matched and nested arbitrarily.

Rather than instantiating this class directly, use *LSM.transaction()*.

Example:

```
with lsm_db.transaction() as txn:
    lsm_db['k1'] = 'v1'

with lsm_db.transaction() as txn:
    lsm_db['k1'] = 'v1-1'
    txn.rollback()

assert lsm_db['k1'] == 'v1'
```

commit()
Commit the transaction and optionally open a new transaction. This is especially useful for context managers, where you may commit midway through a wrapped block of code, but want to retain transactional behavior for the rest of the block.

rollback()
Rollback the transaction and optionally retain the open transaction. This is especially useful for context managers, where you may rollback midway through a wrapped block of code, but want to retain the transactional behavior for the rest of the block.

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

l

lsm, 9

Symbols

__contains__ (lsm.LSM attribute), 10
__delitem__ (lsm.LSM attribute), 10
__enter__() (lsm.Cursor method), 17
__enter__() (lsm.LSM method), 10
__getitem__ (lsm.LSM attribute), 10
__init__ (lsm.LSM attribute), 10
__iter__ (lsm.Cursor attribute), 17
__iter__ (lsm.LSM attribute), 11
__reversed__() (lsm.LSM method), 11
__setitem__ (lsm.LSM attribute), 11

B

begin() (lsm.LSM method), 11

C

checkpoint() (lsm.LSM method), 11
checkpoint_size() (lsm.LSM method), 11
close() (lsm.LSM method), 11
commit() (lsm.LSM method), 11
commit() (lsm.Transaction method), 18
compare() (lsm.Cursor method), 17
Cursor (class in lsm), 17
cursor() (lsm.LSM method), 11

D

delete() (lsm.LSM method), 12
delete_range() (lsm.LSM method), 12

F

fetch() (lsm.LSM method), 12
fetch_range() (lsm.Cursor method), 17
fetch_range() (lsm.LSM method), 13
fetch_until() (lsm.Cursor method), 17
first() (lsm.Cursor method), 17
flush() (lsm.LSM method), 15

I

insert() (lsm.LSM method), 15
is_valid() (lsm.Cursor method), 17

K

keys() (lsm.Cursor method), 17
keys() (lsm.LSM method), 15

L

last() (lsm.Cursor method), 17
LSM (class in lsm), 9
lsm (module), 9

N

next (lsm.Cursor attribute), 17

O

open() (lsm.LSM method), 15

P

pages_read() (lsm.LSM method), 15
pages_written() (lsm.LSM method), 15
previous() (lsm.Cursor method), 17

R

rollback() (lsm.LSM method), 15
rollback() (lsm.Transaction method), 18

S

seek() (lsm.Cursor method), 18

T

Transaction (class in lsm), 18
transaction() (lsm.LSM method), 16
tree_size() (lsm.LSM method), 16

U

update() (lsm.LSM method), 16

V

values() (lsm.Cursor method), 18
values() (lsm.LSM method), 16

W

work() (lsm.LSM method), 16