Package ‘thor’

June 9, 2025

Title Interface to 'LMDB'

Version 1.2.0

Description Key-value store, implemented as a wrapper around 'LMDB';
the ``lightning memory-mapped database'' <https://www.symas.com/mdb>.
'LMDB' is a transactional key value store that uses a memory map
for efficient access. This package wraps the entire 'LMDB'
interface (except duplicated keys), and provides objects for
transactions and cursors.

License MIT + file LICENSE

URL https://github.com/richfitz/thor, https://richfitz.github.io/thor/

BugReports https://github.com/richfitz/thor/issues

Imports R6, storr

Suggests ids, knitr, rmarkdown, testthat

RoxygenNote 7.1.1

VignetteBuilder knitr

ByteCompile true

Encoding UTF-8

Language en-GB

NeedsCompilation yes

Author Rich FitzJohn [aut, cre],
Howard Chu [aut, cph],
Symas Corporation [cph],
Martin Hedenfalk [aut, cph],
The OpenLDAP Foundation [cph]

Maintainer Rich FitzJohn <rich.fitzjohn@gmail.com>

Repository CRAN

Date/Publication 2025-06-09 10:10:02 UTC

Contents

`mdb_cursor` *Use mdb transactions*

Description

Cursors are required for some advanced queries on an mdb database, when the basic set of functions in `mdb_txn` is not sufficient.

Details

Cursors must be created from within a transaction (which in turn are created from an environment).

Methods

`close` Close the cursor

Usage: `close()`

Value: None, called for side effects only

Note: In lmdb.h this is `mdb_cursor_close()`

`put` Store data using the cursor

Usage: `put(key, value, overwrite = TRUE, append = FALSE)`

Arguments:

- `key`: The key (string or raw)
- `value`: The value (string or raw)
- `overwrite`: As for `mdb_txn` `$put`
- `append`: As for `mdb_txn` `$put`

Value: Logical scalar, indicating if data was previously stored at this key

Note: In lmdb.h this is `mdb_cursor_put()`

`del` Delete the current key

Usage: `del()`

Value: Logical, indicating if a value was deleted (which will be `TRUE` if the cursor was valid before this operation). Primarily called for its side effect of deleting the data. After deletion, we call `mdb_cursor_get` with `MDB_GET_CURRENT` which will re-validate the cursor.

Note: In lmdb.h this is `mdb_cursor_del()`

`replace` Replace a key's current value with a new value, returning the old value. This is like doing a `get()` followed by a `put` within a transaction.

Usage: `replace(key, value, as_raw = NULL)`

Arguments:

- `key`: The key to replace
- `value`: The new value to store
- `as_raw`: Return the value as raw. With a value of `NULL` it will return a string if possible (i.e., if there are no null bytes) and a raw vector otherwise. With `as_raw = TRUE` we always return a raw vector. With `as_raw = FALSE` we always return a string, or throw an error if this is not possible.

`pop` Delete a key's value, returning the value just before it was deleted. This is like doing a `get` followed by a `del` within a transaction.

Usage: `pop(key, as_raw = NULL)`

Arguments:

- `key`: The key to delete
- `as_raw`: Return the value as raw. With a value of `NULL` it will return a string if possible (i.e., if there are no null bytes) and a raw vector otherwise. With `as_raw = TRUE` we always return a raw vector. With `as_raw = FALSE` we always return a string, or throw an error if this is not possible.

Value: Depending on `as_raw` and if there is a value stored, `NULL`, a character string or a raw vector

`first` Move the cursor to the first item in the database

Usage: `first()`

Value: Invisibly, a logical indicating if the cursor position is valid, but primarily called for side effects

`last` Move the cursor to the last item in the database

Usage: `last()`

Value: Invisibly, a logical indicating if the cursor position is valid, but primarily called for side effects

`move_next` Move the cursor to the next item in the database. If called while at the last item in the database, this will invalidate the cursor position.

Usage: `move_next()`

Value: Invisibly, a logical indicating if the cursor position is valid, but primarily called for side effects

`move_prev` Move the cursor to the previous item in the database. If called while at the first item in the database, this will invalidate the cursor position.

Usage: `move_prev()`

Value: Invisibly, a logical indicating if the cursor position is valid, but primarily called for side effects

`move_to` Move the cursor to the item in the database with key `key`. If `key` does not exist, this will invalidate the cursor position.

Usage: `move_to(key)`

Arguments:

- key: Key to move to (string or raw)

Value: Invisibly, a logical indicating if the cursor position is valid, but primarily called for side effects

`seek` Move the cursor to the item in the database with key equal to or greater than `key`. If `key` does not exist and no key with a key greater than `key` exists, this will invalidate the cursor position.

Usage: `seek(key)`

Arguments:

- key: Key to seek (string or raw)

Value: Invisibly, a logical indicating if the cursor position is valid, but primarily called for side effects

`get` Move to a key and fetch the value

Usage: `get(key, as_proxy = FALSE, as_raw = NULL)`

Arguments:

- key: The key to find (string or raw)
- as_proxy: Return as an mdb_proxy object?
- as_raw: Return the value as raw. With a value of NULL it will return a string if possible (i.e., if there are no null bytes) and a raw vector otherwise. With as_raw = TRUE we always return a raw vector. With as_raw = FALSE we always return a string, or throw an error if this is not possible.

Value: Depending on as_raw and if there is a value stored, NULL, a character string or a raw vector

`is_valid` Test if cursor is valid (i.e., that it is pointing at data that can be retrieved). Cursors start off invalid until placed (e.g., `first`, `last`) and can be invalidated by moving off the beginning or end of the database.

Usage: `is_valid()`

`key` Return the current key

Usage: `key(as_proxy = FALSE, as_raw = NULL)`

Arguments:

- as_proxy: Return as an mdb_proxy object?
- as_raw: Return the value as raw. With a value of NULL it will return a string if possible (i.e., if there are no null bytes) and a raw vector otherwise. With as_raw = TRUE we always return a raw vector. With as_raw = FALSE we always return a string, or throw an error if this is not possible.

`value` Return the current value

Usage: `value(as_proxy = FALSE, as_raw = NULL)`

Arguments:

- as_proxy: Return as an mdb_proxy object?
- as_raw: Return the value as raw. With a value of NULL it will return a string if possible (i.e., if there are no null bytes) and a raw vector otherwise. With as_raw = TRUE we always return a raw vector. With as_raw = FALSE we always return a string, or throw an error if this is not possible.

Examples

```
# Start by creating a new environment, and within that a write
# transaction, and from that a new cursor.  But first put a bunch
# of data into the database
env <- thor::mdb_env(tempfile())
env$mput(letters, LETTERS)
txn <- env$begin(write = TRUE)
cur <- txn$cursor()

# Move the cursor to the first position
cur$first()

# The key and value:
cur$key()
cur$value()

# Move to a different key:
cur$move_to("g")
cur$value()

# Delete the current item
cur$del()
cur$key()

# We can't move to 'g' any more as it's gone:
(cur$move_to("g"))
cur$key() # NULL

# But we can *seek* 'g', which will move to 'h'
(cur$seek("g"))
cur$key() # "h"

# Get raw values out:
cur$value(as_raw = TRUE)

# Cleanup
env$destroy()
```

`mdb_dbi` *Use mdb transactions*

Description

Database handles are fairly opaque objects used to indicate which database within an `mdb_env` operations will happen to. This object has therefore got very few methods, all of which are purely informative. Most commonly, a `mdb_dbi` object will be passed into the `mdb_env`'s `$begin()` method to begin a transaction on a particular database.

Methods

`path` Return the absolute path to the LMDB store (on disk)

Usage: `path()`

Value: A string

Note: In lmdb.h this is `mdb_env_get_path()`

`flags` Return flags as used in construction of the LMDB environment

Usage: `flags()`

Value: A named logical vector. Names correspond to arguments to the constructor.

Note: In lmdb.h this is `mdb_env_get_flags()`

`info` Brief information about the LMDB environment

Usage: `info()`

Value: An integer vector with elements `mapsize`, `last_pgno`, `last_txnid`, `maxreaders` and `numreaders`.

Note: In lmdb.h this is `mdb_env_info()`

`stat` Brief statistics about the LMDB environment.

Usage: `stat()`

Value: An integer vector with elements `psize` (the size of a database page), `depth` (depth of the B-tree), `brancb_pages` (number of internal non-leaf) pages), `leaf_pages` (number of leaf pages), `overflow_pages` (number of overflow pages) and `entries` (number of data items).

Note: In lmdb.h this is `mdb_env_stat()`

`maxkeysize` The maximum size of a key (the value can be bigger than this)

Usage: `maxkeysize()`

Value: A single integer

Note: In lmdb.h this is `mdb_env_get_maxkeysize()`

`maxreaders` The maximum number of readers

Usage: `maxreaders()`

Value: A single integer

Note: In lmdb.h this is `mdb_env_get_maxreaders()`

`begin` Begin a transaction

Usage: `begin(db = NULL, write = FALSE, sync = NULL, metasync = NULL)`

Arguments:

- `db`: A database handle, as returned by `open_database`. If `NULL` (the default) then the default database will be used.
- `write`: Scalar logical, indicating if this should be a write transaction. There can be only one write transaction per database (see `mdb_txn` for more details) - it is an error to try to open more than one.
- `sync`: Scalar logical, indicating if the data should be synchronised synchronised (flushed to disk) after writes; see main parameter list.
- `metasync`: Scalar logical, indicating if the metadata should be synchronised (flushed to disk) after writes; see main parameter list.

Details: Transactions are the key objects for interacting with an LMDB database (aside from the convenience interface below). They are described in more detail in mdb_txn.

Value: A mdb_txn object

Note: In lmdb.h this is mdb_begin()

with_transaction Evaluate some code within a transaction

Usage: with_transaction(fun, db = NULL, write = FALSE)

Arguments:

- fun: A function of one argument that does the work of the transaction. with_transaction will pass the transaction to this function. This is most easily explained with an example, so see the bottom of the help
- db: A database handle, as returned by open_database. If NULL (the default) then the default database will be used.
- write: Scalar logical, indicating if this should be a write transaction. There can be only one write transaction per database (see mdb_txn for more details) - it is an error to try to open more than one.

Details: This exists to simplify a pattern where one wants to open a transaction, evaluate some code with that transaction and if anything goes wrong abort, but otherwise commit. It is most useful with read-write transactions, but can be used with both (and the default is for readonly transactions, like begin().

open_database Open a named database, or return one if already opened.

Usage: open_database(key = NULL, reversekey = FALSE, create = TRUE)

Arguments:

- key: Name of the database; if NULL this returns the default database (always open).
- reversekey: Compare strings in reverse order? See reversekey documentation above
- create: Create database if it does not exist already?

Details: LMDB environments can hold multiple databases, provided they have been opened with maxdbs greater than one. There is always a "default" database - this is unnamed and cannot be dropped. Other databases have a key (i.e., a name) and can be dropped. These database objects are passed through to other methods, notably drop_database and begin

Note: In lmdb.h this is mdb_open()

drop_database Drop a database

Usage: drop_database(db, delete = TRUE)

Arguments:

- db: A database object, as returned by open_database
- delete: Scalar logical, indicating if the database should be deleted too. If FALSE, the values are deleted from the database (i.e., it is emptied). If TRUE then the actual database is deleted too.

Value: No return value, called for side effects only

Note: In lmdb.h this is mdb_drop()

sync Flush the data buffers to disk.

Usage: sync(force = FALSE)

Arguments:

- `force`: Scalar logical; force a synchronous flush. Otherwise if the environment was constructed with `sync = FALSE` the flushes will be omitted, and with `mapasync = TRUE` they will be asynchronous.

Details: Data is always written to disk when a transaction is committed, but the operating system may keep it buffered. LMDB always flushes the OS buffers upon commit as well, unless the environment was opened with `sync = FALSE` or in part `metasync = FALSE`. This call is not valid if the environment was opened with `readonly = TRUE`.

Note: In lmdb.h this is `mdb_env_sync()`

`copy` Copy the entire environment state to a new path. This can be used to make a backup of the database.

Usage: `copy(path, compact = FALSE)`

Arguments:

- `path`: Scalar character; the new path
- `compact`: Scalar logical; perform compaction while copying? This omits free pages and sequentially renumbers all pages in output. This can take longer than the default but produce a smaller database

Value: Invisibly, the new path (allowing use of `$copy(tempfile)`)

Note: In lmdb.h this is `mdb_env_copy()` & `mdb_env_copy2()`

`close` Close the environment. This closes all cursors and transactions (active write transactions are aborted).

Usage: `close()`

Value: No return value, called for side effects only

Note: In lmdb.h this is `mdb_env_close()`

`destroy` Totally destroy an LMDB environment. This closes the database and removes the files. Use with care!

Usage: `destroy()`

Value: No return value, called for side effects only

`reader_list` List information about database readers

Usage: `reader_list()`

Value: A character matrix with columns `pid` (process ID), `thread` (a pointer address), and `txnid` (a small integer)

Note: In lmdb.h this is `mdb_reader_list()`

`reader_check` Check for, and remove, stale entries in the reader lock table.

Usage: `reader_check()`

Value: An integer, being the number of stale readers discarded. However, this function is primarily called for its side effect.

Note: In lmdb.h this is `mdb_reader_check()`

`get` Retrieve a value from the database

Usage: `get(key, missing_is_error = TRUE, as_raw = NULL, db = NULL)`

Arguments:

- `key`: A string (or raw vector) - the key to get

- missing_is_error: Logical, indicating if a missing value is an error (by default it is). Alternatively, with missing_is_error = FALSE, a missing value will return NULL. Because no value can be NULL (all values must have nonzero length) a NULL is unambiguously missing.
- as_raw: Either NULL, or a logical, to indicate the result type required. With as_raw = NULL, the default, the value will be returned as a string if possible. If not possible it will return a raw vector. With as_raw = TRUE, get() will *always* return a raw vector, even when it is possibly to represent the value as a string. If as_raw = FALSE, get will return a string, but throw an error if this is not possible. This is discussed in more detail in the thor vignette (vignette("thor"))
- db: A database handle that would be passed through to create the transaction (see the $begin method).

Details: This is a helper method that establishes a temporary read-only transaction, calls the corresponding method in mdb_txn and then aborts the transaction.

Note: In lmdb.h this is mdb_get()

put Put values into the database. In other systems, this might be called "set".

Usage: put(key, value, overwrite = TRUE, append = FALSE, db = NULL)

Arguments:

- key: The name of the key (string or raw vector)
- value: The value to save (string or raw vector)
- overwrite: Logical - when TRUE it will overwrite existing data; when FALSE throw an error
- append: Logical - when TRUE, append the given key/value to the end of the database. This option allows fast bulk loading when keys are already known to be in the correct order. But if you load unsorted keys with append = TRUE an error will be thrown
- db: A database handle that would be passed through to create the transaction (see the $begin method).

Details: This is a helper method that establishes a temporary read-write transaction, calls the corresponding method in mdb_txn and then commits the transaction. This will only be possible to use if there is not an existing write transaction in effect for this environment.

Note: In lmdb.h this is mdb_put()

del Remove a key/value pair from the database

Usage: del(key, db = NULL)

Arguments:

- key: The name of the key (string or raw vector)
- db: A database handle that would be passed through to create the transaction (see the $begin method).

Details: This is a helper method that establishes a temporary read-write transaction, calls the corresponding method in mdb_txn and then commits the transaction. This will only be possible to use if there is not an existing write transaction in effect for this environment.

Value: A scalar logical, indicating if the value was deleted

Note: In lmdb.h this is mdb_del()

exists Test if a key exists in the database.

Usage: exists(key, db = NULL)

Arguments:

- key: The name of the key to test (string or raw vector). Unlike get, put and del (but like mget, mput and mdel), exists is *vectorised*. So the input here can be; a character vector of any length (returning the same length logical vector), a raw vector (representing one key, returning a scalar logical) or a list with each element being either a scalar character or a raw vector, returning a logical the same length as the list.
- db: A database handle that would be passed through to create the transaction (see the $begin method).

Details: This is an extension of the raw LMDB API and works by using mdb_get for each key (which for lmdb need not copy data) and then testing whether the return value is MDB_SUCCESS or MDB_NOTFOUND.

This is a helper method that establishes a temporary read-only transaction, calls the corresponding method in mdb_txn and then aborts the transaction.

Value: A logical vector

`list` List keys in the database

Usage: `list(starts_with = NULL, as_raw = FALSE, size = NULL, db = NULL)`

Arguments:

- starts_with: Optionally, a prefix for all strings. Note that is not a regular expression or a filename glob. Using foo will match foo, foo:bar and foobar but not fo or FOO. Because LMDB stores keys in a sorted tree, using a prefix can greatly reduce the number of keys that need to be tested.
- as_raw: Same interpretation as as_raw in $get() but with a different default. It is expected that most of the time keys will be strings, so by default we'll try and return a character vector as_raw = FALSE. Change the default if your database contains raw keys.
- size: For use with starts_with, optionally a guess at the number of keys that would be returned. with starts_with = NULL we can look the number of keys up directly so this is ignored.
- db: A database handle that would be passed through to create the transaction (see the $begin method).

Details: This is a helper method that establishes a temporary read-only transaction, calls the corresponding method in mdb_txn and then aborts the transaction.

`mget` Get values for multiple keys at once (like $get but vectorised over key)

Usage: `mget(key, as_raw = NULL, db = NULL)`

Arguments:

- key: The keys to get values for. Zero, one or more keys are allowed.
- as_raw: As for $get(), logical (or NULL) indicating if raw or string output is expected or desired.
- db: A database handle that would be passed through to create the transaction (see the $begin method).

Details: This is a helper method that establishes a temporary read-only transaction, calls the corresponding method in mdb_txn and then aborts the transaction.

`mput` Put multiple values into the database (like $put but vectorised over key/value).

Usage: `mput(key, value, overwrite = TRUE, append = FALSE, db = NULL)`

Arguments:

- key: The keys to set

- value: The values to set against these keys. Must be the same length as key.
- overwrite: As for $put
- append: As for $put
- db: A database handle that would be passed through to create the transaction (see the $begin method).

Details: The implementation simply calls mdb_put repeatedly (but with a single round of error checking) so duplicate key entries will result in the last key winning.

This is a helper method that establishes a temporary read-write transaction, calls the corresponding method in mdb_txn and then commits the transaction. This will only be possible to use if there is not an existing write transaction in effect for this environment.

mdel Delete multiple values from the database (like $del but vectorised over key).

Usage: mdel(key, db = NULL)

Arguments:

- key: The keys to delete
- db: A database handle that would be passed through to create the transaction (see the $begin method).

Details: This is a helper method that establishes a temporary read-write transaction, calls the corresponding method in mdb_txn and then commits the transaction. This will only be possible to use if there is not an existing write transaction in effect for this environment.

Value: A logical vector, the same length as key, indicating if each key was deleted.

Examples

```
# As always, start with the environment.  Because we're going to
# use more than one database, we must set `maxdbs` to more than 1:
env <- thor::mdb_env(tempfile(), maxdbs = 10)

# The default environment - every database
db <- env$open_database()
# The default database will always have id 1 and no name
db$id()
db$name()

# A different database
foo <- env$open_database("foo")
foo$id()
foo$name()

# Opening a database multiple times has no effect - it returns the
# same data base every call.
identical(env$open_database("foo"), foo) # TRUE

# Then we can put some data int the new database:
txn <- env$begin(foo, write = TRUE)
txn$put("hello", "world")
txn$commit()

# Now we have values in the "foo" database, but not the default one:
```

```
env$get("hello", db = NULL, missing_is_error = FALSE) # NULL
env$get("hello", db = foo,  missing_is_error = FALSE) # "world"

# Cleanup
env$destroy()
```

---

mdb_env *Create an mdb_env environment*

Description

Create a mdb_env "environment" object. This is the way that interacts with a lmdb database and once created, includes methods for querying the environment, creating databases, starting transactions and (through those) adding, getting and removing data. This page includes *reference* documentation for the object and readers are first directed to the vignette (`vignette("thor")`).

Usage

```
mdb_env(
  path,
  mode = as.octmode("644"),
  subdir = TRUE,
  readonly = FALSE,
  metasync = TRUE,
  sync = TRUE,
  writemap = FALSE,
  lock = TRUE,
  mapasync = FALSE,
  rdahead = TRUE,
  meminit = TRUE,
  maxdbs = NULL,
  maxreaders = NULL,
  mapsize = NULL,
  reversekey = FALSE,
  create = TRUE
)
```

Arguments

path	The directory in which the database files will reside. If `create` is `TRUE` this path will be created for you if it does not exist (in contrast with the `lmdb` C API). If `subdir` is `FALSE` this is the path to the database file and an additional lock file will be created by appending "-lock" to `path`.
mode	The file mode (UNIX file permissions) to set on created files. this must be an `octmode` object, with the default (`as.octmode("644")`) being user-writeable and world-readable.

subdir — By default, lmdb creates its files within a directory (at path). If subdir = FALSE then the path is interpreted as the path to the main database file and a lock file will be created with "-lock" appended to the filename. Passing subdir = FALSE is equivalent to lmdb's MDB_NOSUBDIR flag.

readonly — Open the environment in read-only mode. No write operations are allowed. LMDB will still modify the lock file. Passing readonly = TRUE is equivalent to lmdb's MDB_RDONLY flag. If you want a to modify nothing no disk, pass lock = FALSE also (but beware that concurrent access may not go to plan).

metasync — If FALSE, flush system buffers to disk only once per transaction, omit the metadata flush. Defer that until the system flushes files to disk, or next commit or the next call to the $sync() method. This optimization maintains database integrity, but a system crash may undo the last committed transaction. I.e. it preserves the A, C and I (atomicity, consistency, isolation) properties but not D (durability) database property. Passing metasync = FALSE is equivalent to lmdb's MDB_NOMETASYNC flag.

sync — If FALSE, don't flush system buffers to disk when committing a transaction. This optimization means a system crash can corrupt the database or lose the last transactions if buffers are not yet flushed to disk. The risk is governed by how often the system flushes dirty buffers to disk and how often the $sync() method is called. However, if the filesystem preserves write order and writemap = FALSE, transactions exhibit ACI (atomicity, consistency, isolation) properties and only lose D (durability). I.e. database integrity is maintained, but a system crash may undo the final transactions. Note that sync = FALSE, writemap = TRUE leaves the system with no hint for when to write transactions to disk, unless $sync() is called. map_async = TRUE, writemap = TRUE may be preferable. Passing sync = FALSE is equivalent to lmdb's MDB_NOSYNC flag.

writemap — If TRUE, use a writeable memory map unless readonly = TRUE is set. This uses fewer mallocs but loses protection from application bugs like wild pointer writes and other bad updates into the database. This may be slightly faster for databases that fit entirely in RAM, but is slower for databases larger than RAM. Incompatible with nested transactions. Do not mix processes with writemap = TRUE and writemap = FALSE on the same environment. This can defeat durability ($sync() etc). Passing writemap = TRUE is equivalent to lmdb's MDB_WRITEMAP flag.

lock — If FALSE, don't do any locking. If concurrent access is anticipated, the caller must manage all concurrency itself. For proper operation the caller must enforce single-writer semantics, and must ensure that no readers are using old transactions while a writer is active. The simplest approach is to use an exclusive lock so that no readers may be active at all when a writer begins. Passing lock = FALSE is equivalent to lmdb's MDB_NOLOCK flag.

mapasync — If TRUE, When using writemap = TRUE, use asynchronous flushes to disk. As with sync = FALSE, a system crash can then corrupt the database or lose the last transactions. Calling $sync() ensures on-disk database integrity until next commit. Passing mapasync = FALSE is equivalent to lmdb's MDB_MAPASYNC flag.

rdahead — If FALSE, turn off readahead. Most operating systems perform readahead on read requests by default. This option turns it off if the OS supports it. Turning it off may help random read performance when the DB is larger than RAM and

system RAM is full. rdahead = FALSE is not implemented on Windows. Passing rdahead = FALSE is equivalent to lmdb's MDB_NORDAHEAD flag.

meminit — If FALSE, don't initialize malloc'd memory before writing to unused spaces in the data file. By default, memory for pages written to the data file is obtained using malloc. While these pages may be reused in subsequent transactions, freshly malloc'd pages will be initialized to zeroes before use. This avoids persisting leftover data from other code (that used the heap and subsequently freed the memory) into the data file. Note that many other system libraries may allocate and free memory from the heap for arbitrary uses. E.g., stdio may use the heap for file I/O buffers. This initialization step has a modest performance cost so some applications may want to disable it using this flag. This option can be a problem for applications which handle sensitive data like passwords, and it makes memory checkers like Valgrind noisy. This flag is not needed with writemap = TRUE, which writes directly to the mmap instead of using malloc for pages. Passing meminit = FALSE is equivalent to lmdb's MDB_NOMEMINIT.

maxdbs — The number of databases available within the environment. If 0 (the default), then the environment holds just one database (the main db). To use named databases this must be set greater than one.

maxreaders — Maximum number of simultaneous read transactions. Can only be set in the first process to open an environment.

mapsize — Maximum size database may grow to; used to size the memory mapping. This is measured in bytes, and the default (as set in lmdb) is only 1MB (2^20 bytes). If database grows larger than map_size, an error will be thrown and the user must close and reopen the mdb_env. On 64-bit there is no penalty for making this huge (say 1TB). Must be <2GB on 32-bit. Increasing this may cause your operating system to report the disk as being used while your database is open, though this is just the amount reserved.

reversekey — Passed through to open_database for the main database. If TRUE, keys are strings to be compared in reverse order, from the end of the strings to the beginning (e.g., DNS names). By default, keys are treated as strings and compared from beginning to end. Passing reversekey = TRUE is equivalent to lmdb's MDB_REVERSEKEY.

create — If FALSE, do not create the directory path if it is missing.

Details

The thor package is a wrapper around lmdb and so below I have provided pointers to relevant options in lmdb - the wrapper is fairly thin and so picks up limitations and restrictions from the underlying library. Some portions of the documentation here derives from the lmdb source documentation - the file lmdb.h in particular.

Methods

path Return the absolute path to the LMDB store (on disk)

Usage: path()

Value: A string

Note: In lmdb.h this is mdb_env_get_path()

`flags` Return flags as used in construction of the LMDB environment

Usage: `flags()`

Value: A named logical vector. Names correspond to arguments to the constructor.

Note: In lmdb.h this is `mdb_env_get_flags()`

`info` Brief information about the LMDB environment

Usage: `info()`

Value: An integer vector with elements `mapsize`, `last_pgno`, `last_txnid`, `maxreaders` and `numreaders`.

Note: In lmdb.h this is `mdb_env_info()`

`stat` Brief statistics about the LMDB environment.

Usage: `stat()`

Value: An integer vector with elements `psize` (the size of a database page), `depth` (depth of the B-tree), `brancb_pages` (number of internal non-leaf) pages), `leaf_pages` (number of leaf pages), `overflow_pages` (number of overflow pages) and `entries` (number of data items).

Note: In lmdb.h this is `mdb_env_stat()`

`maxkeysize` The maximum size of a key (the value can be bigger than this)

Usage: `maxkeysize()`

Value: A single integer

Note: In lmdb.h this is `mdb_env_get_maxkeysize()`

`maxreaders` The maximum number of readers

Usage: `maxreaders()`

Value: A single integer

Note: In lmdb.h this is `mdb_env_get_maxreaders()`

`begin` Begin a transaction

Usage: `begin(db = NULL, write = FALSE, sync = NULL, metasync = NULL)`

Arguments:

- `db`: A database handle, as returned by `open_database`. If `NULL` (the default) then the default database will be used.
- `write`: Scalar logical, indicating if this should be a write transaction. There can be only one write transaction per database (see `mdb_txn` for more details) - it is an error to try to open more than one.
- `sync`: Scalar logical, indicating if the data should be synchronised synchronised (flushed to disk) after writes; see main parameter list.
- `metasync`: Scalar logical, indicating if the metadata should be synchronised (flushed to disk) after writes; see main parameter list.

Details: Transactions are the key objects for interacting with an LMDB database (aside from the convenience interface below). They are described in more detail in `mdb_txn`.

Value: A `mdb_txn` object

Note: In lmdb.h this is `mdb_begin()`

`with_transaction` Evaluate some code within a transaction

Usage: `with_transaction(fun, db = NULL, write = FALSE)`

Arguments:

- fun: A function of one argument that does the work of the transaction. with_transaction will pass the transaction to this function. This is most easily explained with an example, so see the bottom of the help
- db: A database handle, as returned by open_database. If NULL (the default) then the default database will be used.
- write: Scalar logical, indicating if this should be a write transaction. There can be only one write transaction per database (see mdb_txn for more details) - it is an error to try to open more than one.

Details: This exists to simplify a pattern where one wants to open a transaction, evaluate some code with that transaction and if anything goes wrong abort, but otherwise commit. It is most useful with read-write transactions, but can be used with both (and the default is for readonly transactions, like begin().

open_database Open a named database, or return one if already opened.

Usage: open_database(key = NULL, reversekey = FALSE, create = TRUE)

Arguments:

- key: Name of the database; if NULL this returns the default database (always open).
- reversekey: Compare strings in reverse order? See reversekey documentation above
- create: Create database if it does not exist already?

Details: LMDB environments can hold multiple databases, provided they have been opened with maxdbs greater than one. There is always a "default" database - this is unnamed and cannot be dropped. Other databases have a key (i.e., a name) and can be dropped. These database objects are passed through to other methods, notably drop_database and begin

Note: In lmdb.h this is mdb_open()

drop_database Drop a database

Usage: drop_database(db, delete = TRUE)

Arguments:

- db: A database object, as returned by open_database
- delete: Scalar logical, indicating if the database should be deleted too. If FALSE, the values are deleted from the database (i.e., it is emptied). If TRUE then the actual database is deleted too.

Value: No return value, called for side effects only

Note: In lmdb.h this is mdb_drop()

sync Flush the data buffers to disk.

Usage: sync(force = FALSE)

Arguments:

- force: Scalar logical; force a synchronous flush. Otherwise if the environment was constructed with sync = FALSE the flushes will be omitted, and with mapasync = TRUE they will be asynchronous.

Details: Data is always written to disk when a transaction is committed, but the operating system may keep it buffered. LMDB always flushes the OS buffers upon commit as well, unless the environment was opened with sync = FALSE or in part metasync = FALSE. This call is not valid if the environment was opened with readonly = TRUE.

Note: In lmdb.h this is mdb_env_sync()

`copy` Copy the entire environment state to a new path. This can be used to make a backup of the database.

Usage: `copy(path, compact = FALSE)`

Arguments:

- `path`: Scalar character; the new path
- `compact`: Scalar logical; perform compaction while copying? This omits free pages and sequentially renumbers all pages in output. This can take longer than the default but produce a smaller database

Value: Invisibly, the new path (allowing use of `$copy(tempfile)`)

Note: In lmdb.h this is `mdb_env_copy()` & `mdb_env_copy2()`

`close` Close the environment. This closes all cursors and transactions (active write transactions are aborted).

Usage: `close()`

Value: No return value, called for side effects only

Note: In lmdb.h this is `mdb_env_close()`

`destroy` Totally destroy an LMDB environment. This closes the database and removes the files. Use with care!

Usage: `destroy()`

Value: No return value, called for side effects only

`reader_list` List information about database readers

Usage: `reader_list()`

Value: A character matrix with columns `pid` (process ID), `thread` (a pointer address), and `txnid` (a small integer)

Note: In lmdb.h this is `mdb_reader_list()`

`reader_check` Check for, and remove, stale entries in the reader lock table.

Usage: `reader_check()`

Value: An integer, being the number of stale readers discarded. However, this function is primarily called for its side effect.

Note: In lmdb.h this is `mdb_reader_check()`

`get` Retrieve a value from the database

Usage: `get(key, missing_is_error = TRUE, as_raw = NULL, db = NULL)`

Arguments:

- `key`: A string (or raw vector) - the key to get
- `missing_is_error`: Logical, indicating if a missing value is an error (by default it is). Alternatively, with `missing_is_error = FALSE`, a missing value will return `NULL`. Because no value can be `NULL` (all values must have nonzero length) a `NULL` is unambiguously missing.
- `as_raw`: Either `NULL`, or a logical, to indicate the result type required. With `as_raw = NULL`, the default, the value will be returned as a string if possible. If not possible it will return a raw vector. With `as_raw = TRUE`, `get()` will *always* return a raw vector, even when it is possibly to represent the value as a string. If `as_raw = FALSE`, `get` will return a string, but throw an error if this is not possible. This is discussed in more detail in the thor vignette (`vignette("thor")`)

- db: A database handle that would be passed through to create the transaction (see the $begin method).

Details: This is a helper method that establishes a temporary read-only transaction, calls the corresponding method in mdb_txn and then aborts the transaction.

Note: In lmdb.h this is mdb_get()

put Put values into the database. In other systems, this might be called "set".

Usage: put(key, value, overwrite = TRUE, append = FALSE, db = NULL)

Arguments:

- key: The name of the key (string or raw vector)
- value: The value to save (string or raw vector)
- overwrite: Logical - when TRUE it will overwrite existing data; when FALSE throw an error
- append: Logical - when TRUE, append the given key/value to the end of the database. This option allows fast bulk loading when keys are already known to be in the correct order. But if you load unsorted keys with append = TRUE an error will be thrown
- db: A database handle that would be passed through to create the transaction (see the $begin method).

Details: This is a helper method that establishes a temporary read-write transaction, calls the corresponding method in mdb_txn and then commits the transaction. This will only be possible to use if there is not an existing write transaction in effect for this environment.

Note: In lmdb.h this is mdb_put()

del Remove a key/value pair from the database

Usage: del(key, db = NULL)

Arguments:

- key: The name of the key (string or raw vector)
- db: A database handle that would be passed through to create the transaction (see the $begin method).

Details: This is a helper method that establishes a temporary read-write transaction, calls the corresponding method in mdb_txn and then commits the transaction. This will only be possible to use if there is not an existing write transaction in effect for this environment.

Value: A scalar logical, indicating if the value was deleted

Note: In lmdb.h this is mdb_del()

exists Test if a key exists in the database.

Usage: exists(key, db = NULL)

Arguments:

- key: The name of the key to test (string or raw vector). Unlike get, put and del (but like mget, mput and mdel), exists is *vectorised*. So the input here can be; a character vector of any length (returning the same length logical vector), a raw vector (representing one key, returning a scalar logical) or a list with each element being either a scalar character or a raw vector, returning a logical the same length as the list.
- db: A database handle that would be passed through to create the transaction (see the $begin method).

Details: This is an extension of the raw LMDB API and works by using `mdb_get` for each key (which for lmdb need not copy data) and then testing whether the return value is `MDB_SUCCESS` or `MDB_NOTFOUND`.

This is a helper method that establishes a temporary read-only transaction, calls the corresponding method in `mdb_txn` and then aborts the transaction.

Value: A logical vector

`list` List keys in the database

Usage: `list(starts_with = NULL, as_raw = FALSE, size = NULL, db = NULL)`

Arguments:

- `starts_with`: Optionally, a prefix for all strings. Note that is not a regular expression or a filename glob. Using `foo` will match `foo`, `foo:bar` and `foobar` but not `fo` or `FOO`. Because LMDB stores keys in a sorted tree, using a prefix can greatly reduce the number of keys that need to be tested.
- `as_raw`: Same interpretation as `as_raw` in `$get()` but with a different default. It is expected that most of the time keys will be strings, so by default we'll try and return a character vector `as_raw = FALSE`. Change the default if your database contains raw keys.
- `size`: For use with `starts_with`, optionally a guess at the number of keys that would be returned. with `starts_with = NULL` we can look the number of keys up directly so this is ignored.
- `db`: A database handle that would be passed through to create the transaction (see the `$begin` method).

Details: This is a helper method that establishes a temporary read-only transaction, calls the corresponding method in `mdb_txn` and then aborts the transaction.

`mget` Get values for multiple keys at once (like `$get` but vectorised over `key`)

Usage: `mget(key, as_raw = NULL, db = NULL)`

Arguments:

- `key`: The keys to get values for. Zero, one or more keys are allowed.
- `as_raw`: As for `$get()`, logical (or `NULL`) indicating if raw or string output is expected or desired.
- `db`: A database handle that would be passed through to create the transaction (see the `$begin` method).

Details: This is a helper method that establishes a temporary read-only transaction, calls the corresponding method in `mdb_txn` and then aborts the transaction.

`mput` Put multiple values into the database (like `$put` but vectorised over `key`/`value`).

Usage: `mput(key, value, overwrite = TRUE, append = FALSE, db = NULL)`

Arguments:

- `key`: The keys to set
- `value`: The values to set against these keys. Must be the same length as `key`.
- `overwrite`: As for `$put`
- `append`: As for `$put`
- `db`: A database handle that would be passed through to create the transaction (see the `$begin` method).

Details: The implementation simply calls `mdb_put` repeatedly (but with a single round of error checking) so duplicate `key` entries will result in the last key winning.

This is a helper method that establishes a temporary read-write transaction, calls the corresponding method in `mdb_txn` and then commits the transaction. This will only be possible to use if there is not an existing write transaction in effect for this environment.

`mdel` Delete multiple values from the database (like `$del` but vectorised over `key`).

Usage: `mdel(key, db = NULL)`

Arguments:

- `key`: The keys to delete
- `db`: A database handle that would be passed through to create the transaction (see the `$begin` method).

Details: This is a helper method that establishes a temporary read-write transaction, calls the corresponding method in `mdb_txn` and then commits the transaction. This will only be possible to use if there is not an existing write transaction in effect for this environment.

Value: A logical vector, the same length as `key`, indicating if each key was deleted.

Examples

```
# Create a new environment (just using defaults)
env <- thor::mdb_env(tempfile())

# At its most simple (using temporary transactions)
env$put("a", "hello world")
env$get("a")

# Or create transactions
txn <- env$begin(write = TRUE)
txn$put("b", "another")
txn$put("c", "value")

# Transaction not committed so value not visible outside our transaction
env$get("b", missing_is_error = FALSE)

# After committing, the values are visible for new transactions
txn$commit()
env$get("b", missing_is_error = FALSE)

# A convenience method, 'with_transaction' exists to allow
# transactional workflows with less code repetition.

# This will get the old value of a key 'a', set 'a' to a new value
# and return the old value:
env$with_transaction(function(txn) {
  val <- txn$get("a")
  txn$put("a", "new_value")
  val
}, write = TRUE)

# If an error occurred, the transaction would be aborted.  So far,
# not very interesting!
```

```
# More interesting: implementing redis's RPOPLPUSH that takes the
# last value off of the end of one list and pushes it into the
# start of another.
rpoplpush <- function(env, src, dest) {
  f <- function(txn) {
    # Take the value out of the source list and update
    val <- unserialize(txn$get(src, as_raw = TRUE))
    take <- val[[length(val)]]
    txn$put(src, serialize(val[-length(val)], NULL))

    # Put the value onto the destination list
    val <- unserialize(txn$get(dest, as_raw = TRUE))
    txn$put(dest, serialize(c(val, take), NULL))

    # And we'll return the value that was modified
    take
  }
  env$with_transaction(f, write = TRUE)
}

# Set things up - a source list with numbers 1:5 and an empty
# destination list
env$put("src", serialize(1:5, NULL))
env$put("dest", serialize(integer(0), NULL))

# then try it out:
rpoplpush(env, "src", "dest") # 5
rpoplpush(env, "src", "dest") # 4
rpoplpush(env, "src", "dest") # 3

# Here is the state of the two lists
unserialize(env$get("src"))
unserialize(env$get("dest"))

# The above code will fail if one of the lists is available
env$del("dest")
try(rpoplpush(env, "src", "dest"))

# but because it's in a transaction, this failed attempt leaves src
# unchanged
unserialize(env$get("src"))
```

---

mdb_proxy *Proxy values*

Description

Proxy object. These exist to try and exploit LMDB's copy-free design. LMDB can pass back a read-only pointer to memory without copying it. So rather than immediately trying to read the

whole thing in, this class provides a "proxy" to the data. At the moment they're not terribly useful - all you can do is get the length, and peek at the first bytes! They are used internally in the package to support cursors.

Methods

`data` Return the value from a proxy object

Usage: `data(as_raw = NULL)`

Arguments:

- `as_raw`: Return the value as a raw vector? This has the same semantics as `mdb_env$get` - if `NULL` then the value will be returned as a string as possible, otherwise as a raw vector. If `TRUE` then the value is always returned as a raw vector, and if `FALSE` then the value is always returned as a string (or an error is thrown if that is not possible).

Value: A string or raw vector

`head` Read the first `n` bytes from a proxy

Usage: `head(n = 6L, as_raw = NULL)`

Arguments:

- `n`: The number of bytes to read. If `n` is greater than the length of the object the whole object is returned (same behaviour as head
- `as_raw`: As for `$data()`

`is_raw` Return whether we know a value to be raw or not. This is affected by whether we have successfully turned the value into a string (in which case we can return `FALSE`) or if any `NULL` bytes have been detected. The latter condition may be satisfied by reading the first bit of the proxy with `$head()`

Usage: `is_raw()`

Value: A logical if we can, otherwise `NULL` (for symmetry with `as_raw`)

`is_valid` Test if a proxy object is still valid. Once the proxy is invalid, it cannot be read from any more. Proxies are invalidated if their parent transaction is closed, or if any write operations (e.g., `put`, `del`) have occurred.

Usage: `is_valid()`

Value: Scalar logical

`size` The size of the data - the number of characters in the string, or number of bytes in the raw vector.

Usage: `size()`

Value: Scalar integer

Examples

```
# Start with a write transaction that has written a little data:
env <- thor::mdb_env(tempfile())
txn <- env$begin(write = TRUE)
txn$put("a", "apple")
txn$put("b", "banana")

# We can get a proxy object back by passing as_proxy = TRUE
p <- txn$get("a", as_proxy = TRUE)
```

```
p

# Without copying anything we can get the length of the data
p$size() # == nchar("apple")

# And of course we can get the data
p$data()
p$data(as_raw = TRUE)

# Referencing an invalid proxy is an error, but you can use
# "is_valid()" check to see if it is valid
p$is_valid()

txn$put("c", "cabbage")
p$is_valid()
try(p$data())

# It is possible to read the first few bytes; this might be useful
# to determine if (say) a value is a serialised R object:
txn$put("d", serialize(mtcars, NULL))

# The first 6 bytes of a binary serialised rds object is always
#
#   0x58 0x0a 0x00 0x00 0x00 0x02
#
# for XDR serialisation, or
#
#   0x42 0x0a 0x02 0x00 0x00 0x00
#
# for native little-endian serialisation.
#
# So with a little helper function
is_rds <- function(x) {
  h_xdr <- as.raw(c(0x58, 0x0a, 0x00, 0x00, 0x00, 0x02))
  h_bin <- as.raw(c(0x42, 0x0a, 0x02, 0x00, 0x00, 0x00))
  x6 <- head(x, 6L)
  identical(x6, h_xdr) || identical(x6, h_bin)
}

# We can see that the value stored at 'a' is not rds
p1 <- txn$get("a", as_proxy = TRUE)
is_rds(p1$head(6, as_raw = TRUE))

# But the value stored at 'd' is:
p2 <- txn$get("d", as_proxy = TRUE)
is_rds(p2$head(6, as_raw = TRUE))

# Retrieve and unserialise the value:
head(unserialize(p2$data()))
```

---

mdb_txn *Use mdb transactions*

Description

Transactions are required for every mdb operation. Even when using the convenience functions in mdb_env (get, etc), a transaction is created and committed each time. Within a transaction, either everything happens or nothing happens, and everything gets a single consistent view of the database.

Details

There can be many read transactions per environment, but only one write transactions. Because R is single-threaded, that means that you can only simultaneously write from an mdb environment from a single object - any further attempts to open write transactions it would block forever while waiting for a lock that can't be released because there is only one thread!

Methods

`id` Return the mdb internal id of the transaction

Usage: `id()`

Value: An integer

Note: In lmdb.h this is `mdb_txn_id()`

`stat` Brief statistics about the database. This is the same as mdb_env's `stat()` but applying to the transaction

Usage: `stat()`

Value: An integer vector with elements `psize` (the size of a database page), `depth` (depth of the B-tree), `brancb_pages` (number of internal non-leaf) pages), `leaf_pages` (number of leaf pages), `overflow_pages` (number of overflow pages) and `entries` (number of data items).

Note: In lmdb.h this is `mdb_stat()`

`commit` Commit all changes made in this transaction to the database, and invalidate the transaction, and any cursors belonging to it (i.e., once committed the transaction cannot be used again)

Usage: `commit()`

Value: Nothing, called for its side effects only

Note: In lmdb.h this is `mdb_txn_commit()`

`abort` Abandon all changes made in this transaction to the database, and invalidate the transaction, and any cursors belonging to it (i.e., once aborted the transaction cannot be used again). For read-only transactions there is no practical difference between abort and commit, except that using `abort` allows the transaction to be recycled more efficiently.

Usage: `abort(cache = TRUE)`

Arguments:

- `cache`: Logical, indicating if a read-only transaction should be cached for recycling

Value: Nothing, called for its side effects only

Note: In lmdb.h this is `mdb_txn_abort()`

`cursor` Create a `mdb_cursor` object in this transaction. This can be used for more powerful database interactions.

Usage: `cursor()`

Value: A `mdb_cursor` object.

Note: In lmdb.h this is `mdb_cursor_open()`

`get` Retrieve a value from the database

Usage: `get(key, missing_is_error = TRUE, as_proxy = FALSE, as_raw = NULL)`

Arguments:

- `key`: A string (or raw vector) - the key to get
- `missing_is_error`: Logical, indicating if a missing value is an error (by default it is). Alternatively, with `missing_is_error = FALSE`, a missing value will return `NULL`. Because no value can be `NULL` (all values must have nonzero length) a `NULL` is unambiguously missing.
- `as_proxy`: Return a "proxy" object, which defers doing a copy into R. See `mdb_proxy` for more information.
- `as_raw`: Either `NULL`, or a logical, to indicate the result type required. With `as_raw = NULL`, the default, the value will be returned as a string if possible. If not possible it will return a raw vector. With `as_raw = TRUE`, `get()` will *always* return a raw vector, even when it is possibly to represent the value as a string. If `as_raw = FALSE`, `get` will return a string, but throw an error if this is not possible. This is discussed in more detail in the thor vignette (`vignette("thor")`)

Note: In lmdb.h this is `mdb_get()`

`put` Put values into the database. In other systems, this might be called "`set`".

Usage: `put(key, value, overwrite = TRUE, append = FALSE)`

Arguments:

- `key`: The name of the key (string or raw vector)
- `value`: The value to save (string or raw vector)
- `overwrite`: Logical - when `TRUE` it will overwrite existing data; when `FALSE` throw an error
- `append`: Logical - when `TRUE`, append the given key/value to the end of the database. This option allows fast bulk loading when keys are already known to be in the correct order. But if you load unsorted keys with `append = TRUE` an error will be thrown

Note: In lmdb.h this is `mdb_put()`

`del` Remove a key/value pair from the database

Usage: `del(key)`

Arguments:

- `key`: The name of the key (string or raw vector)

Value: A scalar logical, indicating if the value was deleted

Note: In lmdb.h this is `mdb_del()`

`exists` Test if a key exists in the database.

Usage: `exists(key)`

Arguments:

- `key`: The name of the key to test (string or raw vector). Unlike `get`, `put` and `del` (but like `mget`, `mput` and `mdel`), `exists` is *vectorised*. So the input here can be; a character vector of any length (returning the same length logical vector), a raw vector (representing one key, returning a scalar logical) or a `list` with each element being either a scalar character or a raw vector, returning a logical the same length as the list.

Details: This is an extension of the raw LMDB API and works by using `mdb_get` for each key (which for lmdb need not copy data) and then testing whether the return value is `MDB_SUCCESS` or `MDB_NOTFOUND`.

Value: A logical vector

`list` List keys in the database

Usage: `list(starts_with = NULL, as_raw = FALSE, size = NULL)`

Arguments:

- `starts_with`: Optionally, a prefix for all strings. Note that is not a regular expression or a filename glob. Using `foo` will match `foo`, `foo:bar` and `foobar` but not `fo` or `FOO`. Because LMDB stores keys in a sorted tree, using a prefix can greatly reduce the number of keys that need to be tested.
- `as_raw`: Same interpretation as `as_raw` in `$get()` but with a different default. It is expected that most of the time keys will be strings, so by default we'll try and return a character vector `as_raw = FALSE`. Change the default if your database contains raw keys.
- `size`: For use with `starts_with`, optionally a guess at the number of keys that would be returned. with `starts_with = NULL` we can look the number of keys up directly so this is ignored.

`mget` Get values for multiple keys at once (like `$get` but vectorised over `key`)

Usage: `mget(key, as_proxy = FALSE, as_raw = NULL)`

Arguments:

- `key`: The keys to get values for. Zero, one or more keys are allowed.
- `as_proxy`: Logical, indicating if a list of `mdb_proxy` objects should be returned.
- `as_raw`: As for `$get()`, logical (or `NULL`) indicating if raw or string output is expected or desired.

`mput` Put multiple values into the database (like `$put` but vectorised over `key`/`value`).

Usage: `mput(key, value, overwrite = TRUE, append = FALSE)`

Arguments:

- `key`: The keys to set
- `value`: The values to set against these keys. Must be the same length as `key`.
- `overwrite`: As for `$put`
- `append`: As for `$put`

Details: The implementation simply calls `mdb_put` repeatedly (but with a single round of error checking) so duplicate `key` entries will result in the last key winning.

`mdel` Delete multiple values from the database (like `$del` but vectorised over `key`).

Usage: `mdel(key)`

Arguments:

- `key`: The keys to delete

Value: A logical vector, the same length as `key`, indicating if each key was deleted.

`replace` Use a temporary cursor to replace an item; this function will replace the data held at `key` and return the previous value (or `NULL` if it doesn't exist). See `mdb_cursor` for fuller documentation.

Usage: `replace(key, value, as_raw = NULL)`

Arguments:

- key: The key to replace
- value: The new value value to st key to
- as_raw: For the returned value, how should the data be returned?

Value: As for $get(), a single data item as either a string or raw vector.

pop Use a temporary cursor to "pop" an item; this function will delete an item but return the value that it had as it deletes it.

Usage: pop(key, as_raw = NULL)

Arguments:

- key: The key to pop
- as_raw: For the returned value, how should the data be returned?

Value: As for $get(), a single data item as either a string or raw vector.

cmp Compare two keys for ordering

Usage: cmp(a, b)

Arguments:

- a: A key (string or raw); it need not be in the database
- b: A key to compare with b (string or raw)

Value: A scalar integer, being -1 (if a < b), 0 (if a == b) or 1 (if a > b).

Note: In lmdb.h this is mdb_cmp()

Examples

```
# Start by creating a new environment, and within that a write
# transaction
env <- thor::mdb_env(tempfile())
txn <- env$begin(write = TRUE)

# With this transaction we can write values and see them as set
txn$put("a", "hello")
txn$get("a")

# But because the transaction is not committed, any new
# transaction will not see the values:
env$get("a", missing_is_error = FALSE) # NULL
txn2 <- env$begin()
txn2$get("a", missing_is_error = FALSE) # NULL

# Once we commit a transaction, *new* transactions will see the
# value
txn$commit()
env$get("a") # "hello"
env$begin()$get("a") # "hello"

# But old transactions retain their consistent view of the database
txn2$get("a", missing_is_error = FALSE)

# Cleanup
env$destroy()
```

storr_thor *Thor driver for storr*

Description

Storr driver for thor. This allows thor to be used as a storage backend with the storr package and presents a higher level content addressable key/value store suitable for storing R objects.

Usage

```
storr_thor(
  env,
  prefix = "",
  hash_algorithm = NULL,
  default_namespace = "objects"
)

driver_thor(env, prefix = "", hash_algorithm = NULL)
```

Arguments

env	A thor environment
prefix	An optional prefix. If given, use a : as the last character for nice looking keys (e.g., storr: will generate keys like storr:keys:namespace:name. If not given then we assume that storr is the only user of this database and if destroy is called it will delete the entire database.
hash_algorithm	Optional hash algorithm to use. Defaults to md5, or whatever the existing algorithm is if the database has been opened. You cannot mix algorithms.
default_namespace	The default namespace to store objects in. Defaults to objects, as does other storr drivers.

Index