Package ‘mdbx’

September 28, 2026

Title Bindings to the 'libmdbx' Embedded Key-Value Store

Version 0.1.1

Description Provides low-level bindings to 'libmdbx', a compact and fast transactional key-value store built on memory-mapped files (<https://libmdbx.dqdkfa.ru/>). Database environments, transactions, and byte-oriented read and write operations are exposed directly. The 'libmdbx' sources are bundled and compiled into the package, so no system library installation is required.

License MIT + file LICENSE

URL https://pedrobtz.github.io/mdbx/, https://github.com/pedrobtz/mdbx

BugReports https://github.com/pedrobtz/mdbx/issues

Encoding UTF-8

RoxygenNote 8.0.0

Suggests knitr, rmarkdown, testthat (>= 3.0.0)

Config/testthat/edition 3

Config/Needs/check decor, tibble, vctrs

LinkingTo cpp11

NeedsCompilation yes

Author Pedro Baltazar [aut, cre, cph],
Leonid Yuriev [ctb, cph] (Vendored 'libmdbx' library in src/vendor/libmdbx; see inst/COPYRIGHTS.),
Howard Chu [ctb, cph] (Author of LMDB, from which libmdbx derives; see inst/COPYRIGHTS.),
Symas Corporation [cph] (Copyright holder of LMDB; see inst/COPYRIGHTS.),
Martin Hedenfalk [ctb, cph] (Author of btree.c, from which LMDB derives; see inst/COPYRIGHTS.)

Maintainer Pedro Baltazar <pedrobtz@gmail.com>

Repository CRAN

Date/Publication 2026-09-28 13:20:02 UTC

Contents

`mdbx-concurrency` *Concurrency in mdbx*

Description

How 'libmdbx' shares a database between threads and processes, and what that means for R. This is the contract the rest of the package is built on; the short version is **many readers and one writer, across processes, never within one**.

One transaction at a time per environment

'libmdbx' binds a transaction to the thread that began it, and R is single-threaded. An environment therefore supports exactly one live transaction: `mdbx_txn_begin()` refuses a second rather than deadlocking, and `mdbx_env_close()` refuses while one is open.

There is no way around that by opening the environment twice. 'libmdbx' documents opening an environment more than once from a single process as an error, so `mdbx_env_open()` refuses a path this process already has open, naming the conflict instead of reporting the lock failure 'libmdbx' would. Pass the handle you have, or close it first. A process that needs independent key spaces wants named databases (`mdbx_dbi_open()`) rather than a second environment.

Using a transaction from another thread is rejected by 'libmdbx' itself, with `MDBX_THREAD_MISMATCH`, because the package is compiled with `MDBX_TXN_CHECKOWNER`. `mdbx_version()$build$options` shows that flag, and the test suite asserts it actually fires rather than trusting the build.

Concurrency comes from separate processes

Any number of processes may hold read transactions on one database at the same time, and one of them may hold a write transaction while they do. Writers are serialized against each other by a lock file: a second writer waits, or fails immediately with `MDBX_BUSY` if the transaction was begun with `flags = "TRY"`.

Readers never block writers and writers never block readers. A read transaction sees the snapshot that existed when it began and keeps it, even as other processes commit — so a long-lived reader is safe, though it does hold back the pages its snapshot needs. End read transactions promptly.

fork() does not carry an environment with it

`parallel::mclapply()`, `parallel::mcparallel()` and anything else built on `fork()` give the child a copy of the R object but not the mapping, lock or reader slot behind it. 'libmdbx' invalidates the inherited environment in its own after-fork hook.

Using one in the child is therefore an error naming the fork, rather than a crash: every entry point checks the process that opened the handle. `mdbx_env_is_open()` reports `FALSE` in the child, and closing is refused, so a worker cannot release a lock it never held.

Open the environment inside the worker instead. Each process gets its own handle, and the many-readers-one-writer rules above then apply normally:

```
parallel::mclapply(keys, function(key) {
  env <- mdbx_env_open(path)
  on.exit(mdbx_env_close(env))
  mdbx_with_read(env, function(txn) mdbx_get(txn, key))
})
```

The same applies to `future`, `callr` and any other backend that forks. A backend that starts fresh R processes instead has nothing to inherit, and needs no special care beyond opening its own environment.

Durability is per-environment, not per-process

Flags such as "SAFE_NOSYNC" are properties of the database as it is currently open, and a process that opens an environment another process already has open inherits its flags. "ACCEDE" asks for that explicitly rather than failing on the difference. See mdbx_flags().

See Also

mdbx_txn_begin(), mdbx_env_open(), mdbx_dbi_open(), mdbx_flags()

---

mdbx-errors *Errors raised by mdbx*

Description

How to handle a failure from 'libmdbx' without matching its message text.

Details

A status 'libmdbx' returns and this package could not turn into an ordinary value reaches R as a condition carrying the status itself, not only a sentence describing it. Contention, a full map and a full DBI table are all expected outcomes that a caller may want to retry, grow or report differently, and deciding which is which by parsing English is a contract nobody should have to depend on.

Class and fields

Every such condition inherits from mdbx_error, and from error and condition as usual. When the status has a symbolic name, the condition also carries that name lower-cased as its most specific class — so MDBX_BUSY arrives as:

```
c("mdbx_busy", "mdbx_error", "error", "condition")
```

Three fields beyond message:

code The 'libmdbx' status, as an integer. Negative for MDBX's own codes, positive for a system errno passed through.

name The symbolic name, such as "MDBX_BUSY", or NA for a system errno, which has no MDBX name.

call NULL. The messages name what failed and the argument responsible, so there is nothing a call would add.

What is not an mdbx_error

This package's own refusals — a read-only transaction asked to write, a handle used after its environment closed, an argument of the wrong type, a second `mdbx_env_open()` on a path already open — are ordinary errors with no code. They report a mistake in the calling code rather than a condition the database reached, so there is nothing to retry and no status to inspect. Assertion failures inside 'libmdbx' are also ordinary errors: see `mdbx_txn_state()` for what becomes of the handles.

Remember too that the common "expected" outcomes are not errors at all. A missing key is `NULL` from `mdbx_get()`, a refused overwrite is `FALSE` from `mdbx_put()`, and deleting an absent key is `FALSE` from `mdbx_del()`.

See Also

`mdbx_txn_begin()` for `flags = "TRY"`, which turns waiting for another process's writer into an immediate `MDBX_BUSY`; mdbx-concurrency

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path, max_dbs = 2)

# Reserving only two named databases makes the third one fail.
condition <- tryCatch(
  mdbx_with_write(env, function(txn) {
    for (i in 1:3) mdbx_dbi_open(txn, paste0("db", i), create = TRUE)
  }),
  mdbx_error = function(e) e
)

class(condition)
condition$name
condition$code

mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

---

mdbx_dbi_drop *Empty or delete a named database*

Description

`delete = FALSE` removes every record but keeps the database. `delete = TRUE` removes the database itself, after which the handle refers to nothing and reopening it needs `create = TRUE` again.

Usage

```
mdbx_dbi_drop(txn, db, delete = FALSE)
```

Arguments

txn	An mdbx_txn object from mdbx_txn_begin(), opened for writing. Both emptying and deleting are writes, so a read transaction is refused.
db	An mdbx_dbi object from mdbx_dbi_open(), or NULL for the main database — which can never be deleted, and can be emptied only while no named database exists to be destroyed along with it.
delete	If TRUE, delete the database rather than just emptying it.

Details

The main database is the exception, twice over. It is what records the named ones, so it cannot be deleted at all: db = NULL with delete = TRUE is an error rather than the quiet emptying 'libmdbx' would perform. And emptying it destroys every named database along with it, for the same reason — so that is refused too while any named database exists. Drop those by name first if you really mean to, or delete the main database's own keys individually.

Emptying a database that holds records also resets its sequence counter to zero, because 'libmdbx' rewrites the database's record and the counter lives in it. (Emptying one that is already empty rewrites nothing and leaves the counter alone, but that is not a distinction to build on.) Do not rely on ids minted before an emptying staying unique afterwards — if they are still referenced somewhere, remove the records by deleting their keys instead.

Like every other write, this takes effect only when the transaction commits.

Value

NULL, invisibly.

See Also

mdbx_dbi_open()

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path, max_dbs = 8)

mdbx_with_write(env, function(txn) {
  scratch <- mdbx_dbi_open(txn, "scratch", create = TRUE)
  mdbx_put(txn, "k", "v", db = scratch)
  mdbx_dbi_drop(txn, scratch)
  mdbx_keys(txn, db = scratch)
})

mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

mdbx_dbi_list *List the named databases in an environment*

Description

Reports the named databases visible to this transaction. Visibility is the transaction's own: a database this transaction created is listed immediately, and one it deleted is gone immediately, both before any commit. What other transactions see is decided when this one commits or aborts — until then they see neither the creation nor the deletion.

Usage

```
mdbx_dbi_list(txn, as = c("character", "raw"))
```

Arguments

txn	An mdbx_txn object, from mdbx_txn_begin().
as	"character" (the default) to decode names as UTF-8 text, or "raw" for a list of raw vectors.

Details

That makes this the way to ask whether a database exists without handling an error, which is what mdbx_dbi_open() raises for a name that was never created.

Names are bytes, like keys, so a name that is not valid UTF-8 text needs as = "raw". The unnamed main database is not listed, having no name.

Value

A character vector of names, or a list of raw vectors if as = "raw". Empty when the environment has only the main database.

See Also

mdbx_dbi_open()

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path, max_dbs = 8)

mdbx_with_write(env, function(txn) {
  mdbx_dbi_open(txn, "files", create = TRUE)
  mdbx_dbi_open(txn, "metadata", create = TRUE)
})

mdbx_with_read(env, function(txn) mdbx_dbi_list(txn))
```

```
mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

---

mdbx_dbi_open *Open a named database*

Description

An environment holds an unnamed main database and, if max_dbs allows, any number of named ones. Named databases are independent key spaces: the same key may appear in several with different values, and mdbx_keys() on one never sees another's.

Usage

```
mdbx_dbi_open(txn, name, create = FALSE)
```

Arguments

txn	An mdbx_txn object, from mdbx_txn_begin(). Creating a database needs a write transaction; opening an existing one does not.
name	The database's name, a single string.
create	If TRUE, create the database when it does not exist — which needs a write transaction, and is refused in a read one. If FALSE, opening a database that was never created is an error naming it.

Details

The database is opened for the duration of this transaction and re-resolved by name in later ones, so the returned handle stays usable for the life of the environment — but only if the transaction that created it **commits**. If it aborts, the database was never created and the handle refers to nothing; passing it as db then reports the database as missing, naming it.

Opening a database that does not exist is an error rather than NULL: a name is something you wrote, so a mistyped one is worth reporting where it was written. To find out whether one exists without handling an error, look for it in mdbx_dbi_list().

Reserve capacity with max_dbs in mdbx_env_open() before opening any: the libmdbx default leaves no room for named databases at all, and running out reports MDBX_DBS_FULL.

Value

An mdbx_dbi object, to pass as the db argument of mdbx_get(), mdbx_put(), mdbx_del(), mdbx_keys() and mdbx_items().

See Also

mdbx_dbi_list(), mdbx_dbi_drop(), mdbx_env_open() for max_dbs

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path, max_dbs = 8)

mdbx_with_write(env, function(txn) {
  files <- mdbx_dbi_open(txn, "files", create = TRUE)
  metadata <- mdbx_dbi_open(txn, "metadata", create = TRUE)

  mdbx_put(txn, "abc", "/data/abc.parquet", db = files)
  mdbx_put(txn, "abc", '{"size":1234}', db = metadata)
})

# The same key, two databases, two values.
mdbx_with_read(env, function(txn) {
  c(files = mdbx_get(txn, "abc", db = mdbx_dbi_open(txn, "files")),
    metadata = mdbx_get(txn, "abc", db = mdbx_dbi_open(txn, "metadata")))
})

# Opening one that was never created is an error, so a reader that does not
# know which exist yet asks rather than catching.
mdbx_with_read(env, function(txn) {
  c("files" %in% mdbx_dbi_list(txn), "sizes" %in% mdbx_dbi_list(txn))
})

mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

`mdbx_dbi_sequence` *A database's sequence counter*

Description

Every database carries a 64-bit counter that 'libmdbx' stores with it. Reading it with `increment = 0` reports its current value; a positive `increment` reserves that many values and returns the first, so two callers in separate transactions can never be handed the same number.

Usage

```
mdbx_dbi_sequence(txn, db = NULL, increment = 0)
```

Arguments

`txn`	An `mdbx_txn` object from `mdbx_txn_begin()`. Incrementing needs a write transaction and is refused in a read one; reading the counter — `increment = 0` — works in either.
`db`	An `mdbx_dbi` object from `mdbx_dbi_open()`, or `NULL` for the main database.
`increment`	How many values to reserve. `0`, the default, reads the counter without changing it.

Details

It is the natural way to mint ids — a monotonically increasing insertion order, of the kind a cache uses to evict what was stored longest ago. Encode the result big-endian if it is going to be a key, so that byte order matches numeric order.

Like every other write, an increment only stands if the transaction commits. Emptying or deleting the database can reset the counter to zero — see `mdbx_dbi_drop()`.

Value

The counter's value before the increment, as a number.

See Also

`mdbx_dbi_open()`

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path, max_dbs = 8)

mdbx_with_write(env, function(txn) {
  ids <- mdbx_dbi_open(txn, "ids", create = TRUE)

  c(first = mdbx_dbi_sequence(txn, ids, 1),
    second = mdbx_dbi_sequence(txn, ids, 1),
    current = mdbx_dbi_sequence(txn, ids))
})

mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

---

mdbx_del *Delete a key*

Description

Removes `key` and its value. Deleting a key that is not present is not an error; the return value says which happened.

Usage

```
mdbx_del(txn, key, db = NULL)
```

Arguments

`txn`	An `mdbx_txn` object from `mdbx_txn_begin()`, opened with `write = TRUE`.
`key`	A raw vector, or a single string, which is stored as its UTF-8 bytes.
`db`	An `mdbx_dbi` object from `mdbx_dbi_open()` naming a database to address instead of the unnamed main one, or `NULL` for the main database.

Value

TRUE if a record existed and was removed, FALSE otherwise. Returned invisibly.

See Also

mdbx_get(), mdbx_put()

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path)

mdbx_with_write(env, function(txn) {
  mdbx_put(txn, "k", "v")
  mdbx_del(txn, "k")
})

mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

---

mdbx_env_close *Close an MDBX environment*

Description

Closes the environment and releases its native resources. Closing is idempotent: calling it on an already-closed environment does nothing, so it is safe to pair an explicit close with an on.exit() guard.

Usage

```
mdbx_env_close(env)
```

Arguments

env	An mdbx_env object, from mdbx_env_open().

Details

Closing is refused while any transaction on the environment is still open, because 'libmdbx' documents that using a transaction after its environment closes is undefined behaviour. Commit or abort them first — or use mdbx_with_read() / mdbx_with_write(), which cannot leave one open.

Using a closed environment for anything else is an error rather than a crash.

Value

NULL, invisibly.

See Also

mdbx_env_open(), mdbx_env_is_open()

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path)

mdbx_env_close(env)
mdbx_env_is_open(env)

unlink(c(path, paste0(path, "-lck")))
```

---

mdbx_env_get_flags *Flags an environment is using*

Description

Reports every flag in effect on an open environment, including the ones mdbx_env_open() set from its own readonly and subdir arguments — this describes the environment, not the call that made it.

Usage

```
mdbx_env_get_flags(env)
```

Arguments

env	An mdbx_env object, from mdbx_env_open().

Details

Flags can appear that were never asked for. 'libmdbx' normalizes the sync modes, so both "SAFE_NOSYNC" and "UTTERLY_NOSYNC" also report "NOMETASYNC" — the weaker relaxation is implied by the stronger one, and clearing the stronger one does not clear it. "UTTERLY_NOSYNC" does *not* additionally report "SAFE_NOSYNC", whose bits it contains: naming both would describe one durability mode as two. An environment another process opened first may also carry flags this one did not ask for; that is what "ACCEDE" is about.

'libmdbx' keeps internal state in the same word, which is not reported here: only the names in mdbx_flags() are ever returned.

Value

A character vector of flag names, empty if the environment is at 'libmdbx"s defaults in every respect.

See Also

mdbx_flags(), mdbx_env_set_flags()

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path, flags = "NOMETASYNC")

mdbx_env_get_flags(env)

mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

`mdbx_env_info` *Environment information*

Description

Reports the environment as a whole: the size limits it was opened with, how much of the map is in use, the most recent transaction, and reader slots.

Usage

```
mdbx_env_info(x, ...)
```

Arguments

`x`	An `mdbx_env` from `mdbx_env_open()`, or an `mdbx_txn` from `mdbx_txn_begin()`.
`...`	Unused, for extensibility.

Details

The choice between an environment and a transaction behaves as described for `mdbx_env_stat()`.

Value

A named list of numbers. The `geo_*` entries are the datafile geometry — `geo_lower` and `geo_upper` are the bounds the map may grow between (`geo_upper` is what `map_size` sets in `mdbx_env_open()`), `geo_current` its present size, and `geo_shrink` / `geo_grow` the steps it changes by. Also `mapsize`, `file_size`, `last_pgno`, `recent_txnid`, `latter_reader_txnid`, `maxreaders`, `numreaders`, `pagesize` and `sys_pagesize`.

This is a useful subset, not the whole of 'libmdbx''s `MDBX_envinfo`; the omitted fields are meta-page signatures, boot ids, page-operation counters and sync timings, which are diagnostics for 'libmdbx' itself.

See Also

`mdbx_env_stat()`

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path, map_size = 4 * 1024^2)

info <- mdbx_env_info(env)
info$geo_upper
info$numreaders

mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

---

mdbx_env_is_open *Is an MDBX environment still open?*

Description

Is an MDBX environment still open?

Usage

```
mdbx_env_is_open(env)
```

Arguments

env	An mdbx_env object, from mdbx_env_open().

Value

TRUE if the environment is open and usable, FALSE once it has been closed.

See Also

mdbx_env_open(), mdbx_env_close()

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path)

mdbx_env_is_open(env)
mdbx_env_close(env)
mdbx_env_is_open(env)

unlink(c(path, paste0(path, "-lck")))
```

mdbx_env_open *Open an MDBX environment*

Description

Opens, and by default creates, an MDBX environment at `path`. An environment is the unit that holds the memory map, the lock, and the databases within it; transactions and all data access happen against one.

Usage

```
mdbx_env_open(
  path,
  readonly = FALSE,
  create = TRUE,
  subdir = FALSE,
  max_dbs = 16L,
  map_size = NULL,
  max_readers = NULL,
  mode = "0664",
  flags = NULL
)
```

Arguments

`path`	Path to the environment. With `subdir = FALSE` (the default) this is the data file itself, and the lock file is the same path with `-lck` appended. With `subdir = TRUE` it is a directory, which libmdbx populates with `mdbx.dat` and `mdbx.lck`.
`readonly`	If `TRUE`, open for reading only; no write transaction can be started, and the environment must already exist.
`create`	If `FALSE`, require the environment to exist already rather than creating it. This is a best-effort check rather than an atomic one: libmdbx has no "open but never create" flag, so the existence test and the open are two steps, and another process deleting the database in between would leave a fresh one created here.
`subdir`	Selects the on-disk layout described under `path`. It applies only when creating; opening an existing environment detects the layout.
`max_dbs`	How many named databases to make room for. The default of 16 is this package's, not libmdbx's: libmdbx reserves none, which makes `mdbx_dbi_open()` fail with `MDBX_DBS_FULL` on an environment opened with default arguments. Unused slots cost nothing. Each name that Each name that `mdbx_dbi_open()` resolves occupies one slot. Raise this if you need more than 16, or pass `NULL` to take libmdbx's default of none — which leaves only the unnamed main database usable.
`map_size`	Upper bound, in bytes, on the size the memory map may grow to, or `NULL` for the libmdbx default. Only the upper bound is set; the initial size and growth behaviour stay at their defaults.

`max_readers` Number of reader slots to make room for, or `NULL` for the libmdbx default. One slot is used per process holding a read transaction, so this is the ceiling on concurrent readers across all processes — see mdbx-concurrency. The default is derived from the lock file's page size (a few hundred, platform-dependent); raise it only if you expect more concurrent reader processes than that. It sizes the lock file, so it takes effect only for the first process to open the environment.

`mode` File permissions for a newly created database, as a string of octal digits or an octmode object. The default `"0664"` is libmdbx's own, and is masked by the process `umask` as usual — so it typically lands as `0644`, readable by everyone. Pass `"0600"` for a database only its owner can read. Ignored when the database already exists.

`flags` A character vector of 'libmdbx' flag names, or `NULL` for none. These are the remaining `MDBX_*` environment flags, with the prefix dropped — `mdbx_flags()` lists them and explains what each does.

The durability flags live here: `"NOMETASYNC"`, `"SAFE_NOSYNC"` and `"UTTERLY_NOSYNC"` each make commits cheaper by giving up some of what a crash cannot take away, and the last of the three can leave the database corrupt. The default — passing nothing — is fully durable. Read the Durability section of `mdbx_flags()` before using any of them.

Details

The environment is closed when `mdbx_env_close()` is called on it, or when the object is garbage collected, whichever happens first. Relying on garbage collection is safe but not timely; close explicitly when the moment matters.

One handle per environment per process. 'libmdbx' documents opening an environment more than once from a single process as an error, so a second call on a path this process already has open is refused whatever its other arguments say — keep the handle you were given and share it, or close it first. The refusal looks past the spelling: a relative path and an absolute one, a symlinked directory, and the `mdbx.dat` inside a `subdir = TRUE` environment all name the environment they resolve to, and the message says which spelling the open handle was created under. Other processes are unaffected: opening the same environment concurrently from several of them is the normal case, and the one mdbx-concurrency is about.

Value

An `mdbx_env` object.

See Also

`mdbx_env_close()`, `mdbx_env_is_open()`, `mdbx_flags()`, mdbx-concurrency

Examples

```
path <- tempfile(fileext = ".mdbx")

env <- mdbx_env_open(path)
env
```

```
mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

`mdbx_env_reader_check` *Reclaim reader slots from processes that died*

Description

A process holding a read transaction occupies a slot in the environment's reader table. If it exits without ending the transaction — killed, crashed, or `SIGKILL`ed — the slot stays occupied. Enough of those and new readers fail with `MDBX_READERS_FULL` even though nothing is actually reading.

Usage

```
mdbx_env_reader_check(env)
```

Arguments

`env`	An `mdbx_env` object, from `mdbx_env_open()`.

Details

This asks 'libmdbx' to check every occupied slot and release the ones whose owning process is gone. It is safe to call at any time and costs nothing when there is nothing to reclaim, so it is a reasonable thing to run when opening a long-lived environment that other processes also use.

Raising `mdbx_env_open()`'s `max_readers` makes the table bigger; this makes room in the table you have. See mdbx-concurrency.

Value

The number of stale slots that were released, invisibly. Zero when every occupied slot belongs to a live process.

See Also

`mdbx_env_info()`, which reports `numreaders` and `maxreaders`.

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path)

# Nothing has died, so nothing is reclaimed.
mdbx_env_reader_check(env)

mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

mdbx_env_set_flags *Change an environment's flags*

Description

Sets or clears flags on an open environment. Only the flags 'libmdbx' documents as changeable at any time can be set this way — `mdbx_flags()$runtime` marks them; the rest are fixed when the environment is opened.

Usage

```
mdbx_env_set_flags(env, flags, on = TRUE)
```

Arguments

env	An mdbx_env object, from mdbx_env_open().
flags	A character vector of flag names — see mdbx_flags().
on	TRUE to set the flags, FALSE to clear them.

Details

This is refused while a transaction is open, because 'libmdbx' serializes flag changes against the writer lock and would return MDBX_BUSY. Changing durability between transactions is the intended use: relax it for a bulk load, restore it and call mdbx_env_sync() afterwards.

Value

NULL, invisibly.

See Also

mdbx_flags(), mdbx_env_get_flags(), mdbx_env_sync()

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path)

# Bulk load without paying for a flush per commit.
mdbx_env_set_flags(env, "SAFE_NOSYNC")
for (i in 1:10) {
  mdbx_with_write(env, function(txn) mdbx_put(txn, sprintf("k%d", i), "v"))
}

# Then make it durable again, and flush what is outstanding.
mdbx_env_set_flags(env, "SAFE_NOSYNC", on = FALSE)
mdbx_env_sync(env)
```

```
mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

`mdbx_env_stat` *Database statistics*

Description

Reports the shape of the main database: its page size, B-tree depth, page counts, and how many records it holds.

Usage

```
mdbx_env_stat(x, ...)

## S3 method for class 'mdbx_txn'
mdbx_env_stat(x, db = NULL, ...)
```

Arguments

`x`	An `mdbx_env` from `mdbx_env_open()`, or an `mdbx_txn` from `mdbx_txn_begin()`.
`...`	Unused, for extensibility.
`db`	An `mdbx_dbi` object from `mdbx_dbi_open()` to report on instead of the main database, or `NULL` for the main one. Only available when `x` is a transaction, since a database is resolved within one.

Details

Passing a transaction reports what that transaction sees, including changes it has made but not committed. Passing the environment reports the same thing whenever this thread holds a transaction, because the environment form reuses it — so uncommitted changes are included, and an abort takes the counts back down.

The two therefore agree in a single-threaded R session. They differ only when a transaction holds a snapshot the environment has moved past, which requires another process to have committed in the meantime.

Without `db`, the counts cover the **whole environment** — every named database as well as the main one. Measured, since 'libmdbx' does not say so: a main database of 4 keys plus a named database of 7 reports 11 entries. Pass `db` for one database's own B-tree.

Value

A named list of numbers: `pagesize`, `depth`, `branch_pages`, `leaf_pages`, `overflow_pages`, `entries` (the number of records), and `mod_txnid` (the transaction that last modified the database). Counts are `double` because 'libmdbx' reports them as 64-bit integers, which R has no type for; every realistic value is exact.

See Also

mdbx_env_info()

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path)

mdbx_with_write(env, function(txn) {
  mdbx_put(txn, "a", "1")

  # Counted before the commit.
  mdbx_env_stat(txn)$entries
})

mdbx_env_stat(env)$entries

mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

---

mdbx_env_sync *Flush an environment to disk*

Description

Writes and flushes any data a relaxed durability mode has left outstanding. With the default durability there is never anything to flush, because mdbx_txn_commit() has already done it.

Usage

```
mdbx_env_sync(env, force = TRUE, nonblock = FALSE)
```

Arguments

env	An mdbx_env object, from mdbx_env_open().
force	If TRUE, always flush. If FALSE, flush only if a threshold set on the environment has been reached, which is 'libmdbx"s polling mode.
nonblock	If TRUE, give up rather than wait when another process is in a write transaction. The wait is signalled as an MDBX_BUSY error.

Details

Under "SAFE_NOSYNC" this also establishes a new steady commit point, which is what lets 'libmdbx' start reusing freed pages again — so it bounds file growth as well as making data durable.

Value

Invisibly, TRUE if there was unsynced data and it was written, or FALSE if nothing was pending.

See Also

mdbx_flags(), mdbx_env_set_flags()

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path, flags = "SAFE_NOSYNC")

mdbx_with_write(env, function(txn) mdbx_put(txn, "k", "v"))

# The commit above flushed nothing; this does.
mdbx_env_sync(env)

mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

---

mdbx_flags *Flags accepted by 'libmdbx'*

Description

Lists the flag names mdbx_env_open(), mdbx_txn_begin() and mdbx_env_set_flags() accept. They are 'libmdbx''s own names with the MDBX_ prefix dropped, so anything written about MDBX_SAFE_NOSYNC upstream applies to "SAFE_NOSYNC" here.

Usage

```
mdbx_flags()
```

Value

A data frame with one row per flag and the columns flag, scope ("env" or "txn"), settable (accepted by mdbx_env_open() or mdbx_txn_begin()) and runtime (accepted by mdbx_env_set_flags()).

Durability

By default every commit is fully durable: data and metadata are flushed before mdbx_txn_commit() returns, and a crash at any moment leaves the database intact. Three flags trade that away, in increasing order of risk:

"NOMETASYNC" Skips the metadata flush. A system crash may undo the last committed transaction. Integrity is never at risk.

"SAFE_NOSYNC" Flushes nothing on commit. A crash rolls the database back to the last steady commit — recent transactions are lost, but the database **cannot be corrupted**. The cost is file growth, because pages freed since that steady point cannot be reused; mdbx_env_sync() establishes a new one.

`"UTTERLY_NOSYNC"` As above, but previous steady commits are wiped too. A crash shortly after a commit **can corrupt the database beyond recovery**. 'libmdbx"s own documentation cites a messenger that lost user data this way. Suitable only for data you are prepared to regenerate from scratch.

Measure before reaching for these. The cost they remove is per-*commit*, not per-write: on a benchmark of the vendored library, 2000 single-write transactions ran 89x faster under `"SAFE_NOSYNC"`, while one transaction of 200000 writes ran 1.1x faster. Batching writes into fewer transactions is usually the same win at no risk.

Other flags

`"WRITEMAP"` maps the database writable and updates it in place, which is faster but exposes the map to stray writes from the process; it also changes what `"SAFE_NOSYNC"` does, to asynchronous mmap flushes. `"LIFORECLAIM"` reuses the most recently freed pages first, which suits filesystems with copy-on-write or trim. `"NORDAHEAD"` suppresses readahead for databases larger than RAM, `"NOMEMINIT"` skips zero-filling new pages, `"EXCLUSIVE"` takes the environment for this process alone, `"ACCEDE"` accepts the flags an existing environment was opened with rather than conflicting with them, and `"VALIDATION"` turns on expensive internal checking for debugging.

What is not here

`"RDONLY"` and `"NOSUBDIR"` are reported but not settable, because they are the `readonly` and `subdir` arguments of `mdbx_env_open()`. `NOSTICKYTHREADS` is never set: it lifts 'libmdbx"s one-transaction-per-thread rule, which this package's transaction registry and finalizer ordering rely on.

See Also

`mdbx_env_open()`, `mdbx_env_set_flags()`, `mdbx_env_sync()`

Examples

```
mdbx_flags()

# The durability flags, and where each may be set.
flags <- mdbx_flags()
flags[grepl("SYNC", flags$flag), ]
```

---

mdbx_get *Read a value*

Description

Looks up `key` in the transaction's snapshot.

Usage

```
mdbx_get(txn, key, default = NULL, as = c("character", "raw"), db = NULL)
```

Arguments

txn	An mdbx_txn object, from mdbx_txn_begin().
key	A raw vector, or a single string, which is stored as its UTF-8 bytes. The two are interchangeable: "k" and charToRaw("k") name the same key. Only raw can express a key containing a NUL byte.
default	Value returned when key is absent. It is returned exactly as given, and is never decoded, whatever as is set to.
as	"character" (the default) to decode the stored bytes as UTF-8 text, or "raw" to return them untouched. Decoding fails, rather than returning something corrupt, if the value contains a NUL byte or is not valid UTF-8 — which is what any non-text value will do.
db	An mdbx_dbi object from mdbx_dbi_open() naming a database to address instead of the unnamed main one, or NULL for the main database.

Details

A key that is not present returns default (NULL unless you say otherwise). That is unambiguous: a *stored* zero-length value comes back as "" under the default decoding, or as raw(0) with as = "raw". Neither is NULL, so absence and emptiness stay distinguishable either way.

By default the stored bytes are decoded as UTF-8 text, so a value written as a string comes back as one. MDBX records no type, so this is an assumption rather than something the database knows: **pass as = "raw" for any value that is not text**, including anything written with serialize(). Decoding raises an error rather than returning something corrupt when the bytes are not valid UTF-8 text, so a wrong assumption is never silent.

The returned vector is a copy. 'libmdbx' hands out memory owned by the database, valid only until the transaction ends, so nothing here points into the memory map.

Value

A length-1 character vector — or a raw vector if as = "raw" — or default if the key is not present.

See Also

mdbx_put(), mdbx_del()

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path)

mdbx_with_write(env, function(txn) {
  mdbx_put(txn, "answer", "42")
})

# Decoded as text by default.
mdbx_with_read(env, function(txn) {
  mdbx_get(txn, "answer")
})
```

```
# Anything that is not text needs as = "raw".
mdbx_with_read(env, function(txn) {
  mdbx_get(txn, "answer", as = "raw")
})

mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

`mdbx_items` *List the keys and values in a database*

Description

Walks the whole database in key order and returns both keys and values, in one pass. Use `mdbx_keys()` instead when you do not need the values — fetching them is the expensive part.

Usage

```
mdbx_items(
  txn,
  limit = NULL,
  as = c("character", "raw"),
  db = NULL,
  start = NULL,
  reverse = FALSE,
  keys_as = NULL
)
```

Arguments

`txn`	An `mdbx_txn` object, from `mdbx_txn_begin()`.
`limit`	Maximum number of records to return, in key order. `NULL`, the default, returns all of them, subject to the mdbx_scan_max guard; `Inf` returns all of them unconditionally.
`as`	`"character"` (the default) to decode values as UTF-8 text, or `"raw"` for a list of raw vectors. Decoding fails rather than corrupting anything, so a database holding `serialize()` output needs `as = "raw"`. Governs keys too unless `keys_as` says otherwise.
`db`	An `mdbx_dbi` object from `mdbx_dbi_open()` naming a database to address instead of the unnamed main one, or `NULL` for the main database.
`start`	Begin at this key rather than at an end: the first key at or after it going forwards, or the last key at or before it going backwards. A raw vector or a single string, as for `mdbx_get()`. `start` is *inclusive*, so a key that exists is returned rather than skipped. To walk a database in chunks, pass the last key of one call as the `start` of the next and

drop the first record of the result — otherwise it repeats. `start` positions the cursor and nothing more: it does not bound how much comes back, so it does not exempt a scan from mdbx_scan_max. Pass `limit` for that.

`reverse` — If `TRUE`, walk from the last key towards the first. `limit = 1` with `reverse = TRUE` is the largest key, which is otherwise only reachable by reading every key.

`keys_as` — How to decode keys, when that differs from the values. An index typically has binary keys — an encoded timestamp or counter — and text values, which is `as = "character", keys_as = "raw"`.

Details

The two components are parallel: `keys[[i]]` names `values[[i]]`. When keys are text, `setNames(items$values, items$keys)` turns the result into a lookup list.

The caution in `mdbx_keys()` about materializing everything applies here with more force, since values are usually larger than keys.

Value

A list with two parallel components, `keys` and `values`, each a character vector or a list of raw vectors according to `as`.

See Also

`mdbx_keys()`, `mdbx_get()`

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path)

mdbx_with_write(env, function(txn) {
  mdbx_put(txn, "a", "1")
  mdbx_put(txn, "b", "2")
})

items <- mdbx_with_read(env, function(txn) mdbx_items(txn))
items

# A lookup list, when the keys are text.
stats::setNames(items$values, items$keys)

mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

mdbx_keys *List the keys in a database*

Description

Walks the whole database in key order and returns its keys.

Usage

```
mdbx_keys(
  txn,
  limit = NULL,
  as = c("character", "raw"),
  db = NULL,
  start = NULL,
  reverse = FALSE
)
```

Arguments

txn	An mdbx_txn object, from mdbx_txn_begin().
limit	Maximum number of records to return, in key order. NULL, the default, returns all of them, subject to the mdbx_scan_max guard; Inf returns all of them unconditionally.
as	"character" (the default) to decode keys as UTF-8 text, or "raw" to get a list of raw vectors. As with mdbx_get(), decoding fails rather than corrupting anything if a key is not valid UTF-8 text.
db	An mdbx_dbi object from mdbx_dbi_open() naming a database to address instead of the unnamed main one, or NULL for the main database.
start	Begin at this key rather than at an end: the first key at or after it going forwards, or the last key at or before it going backwards. A raw vector or a single string, as for mdbx_get(). start is *inclusive*, so a key that exists is returned rather than skipped. To walk a database in chunks, pass the last key of one call as the start of the next and drop the first record of the result — otherwise it repeats. start positions the cursor and nothing more: it does not bound how much comes back, so it does not exempt a scan from mdbx_scan_max. Pass limit for that.
reverse	If TRUE, walk from the last key towards the first. limit = 1 with reverse = TRUE is the largest key, which is otherwise only reachable by reading every key.

Details

Everything is materialized in memory at once, so on a large database this can be expensive. A scan with no limit therefore refuses to run when the database holds more than mdbx_scan_max records; pass an explicit limit — or limit = Inf — to go ahead anyway. mdbx_env_stat()$entries tells you how many there are before you ask.

Value

A character vector, or a list of raw vectors if `as = "raw"`. Empty (`character(0)` or `list()`) if the database has no records.

See Also

`mdbx_items()`, `mdbx_get()`

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path)

mdbx_with_write(env, function(txn) {
  mdbx_put(txn, "b", "2")
  mdbx_put(txn, "a", "1")
  mdbx_put(txn, "c", "3")
})

# Always in key order, whatever order they were written in.
mdbx_with_read(env, function(txn) mdbx_keys(txn))

mdbx_with_read(env, function(txn) mdbx_keys(txn, limit = 2))

mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

---

mdbx_limits *Size limits imposed by 'libmdbx'*

Description

Reports the largest key, value and database 'libmdbx' will accept. Most of these follow from the page size, which is why they are worth asking about rather than assuming: the maximum key is a little under half a page, so it is 8166 bytes where pages are 16 KB and 2022 bytes where they are 4 KB. Code that hardcodes a size works on one machine and fails on another.

Usage

```
mdbx_limits(x = NULL)
```

Arguments

`x`	What to report limits for. An `mdbx_env` or `mdbx_txn` uses that database's actual page size — usually what you want. A single number is treated as a page size, for asking about a platform you are not on. `NULL`, the default, uses this system's default page size.

Value

A named list of numbers: `pagesize`, `keysize_min`, `keysize_max`, `valsize_min`, `valsize_max`, `dbsize_min`, `dbsize_max` and `txnsize_max`, all in bytes. `keysize_min` and `valsize_min` are zero, which is how an empty key and an empty value are both legal.

See Also

`mdbx_env_stat()`, which reports the page size in use.

Examples

```
# This system's defaults.
mdbx_limits()$keysize_max

# The same question for a machine with 4 KB pages.
mdbx_limits(4096)$keysize_max

path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path)

# The limits that actually apply to this database.
limits <- mdbx_limits(env)
limits$keysize_max

mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

---

mdbx_put *Write a value*

Description

Stores `value` under `key`, replacing any existing value unless `overwrite` is `FALSE`.

Usage

```
mdbx_put(txn, key, value, overwrite = TRUE, db = NULL)
```

Arguments

`txn`	An `mdbx_txn` object from `mdbx_txn_begin()`, opened with `write = TRUE`.
`key`	A raw vector, or a single string, which is stored as its UTF-8 bytes.
`value`	A raw vector, or a single string, which is stored as its UTF-8 bytes. Use `serialize(x, NULL)` for an arbitrary R object.
`overwrite`	If `FALSE`, leave an existing value alone and return `FALSE` instead of replacing it.
`db`	An `mdbx_dbi` object from `mdbx_dbi_open()` naming a database to address instead of the unnamed main one, or `NULL` for the main database.

Value

TRUE if the value was stored, FALSE if overwrite = FALSE and the key already existed. Returned invisibly.

See Also

mdbx_get(), mdbx_del()

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path)

mdbx_with_write(env, function(txn) {
  mdbx_put(txn, "k", "first")

  # Refuses to replace, and says so.
  mdbx_put(txn, "k", "second", overwrite = FALSE)
})

mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

---

mdbx_scan_max *Largest scan* mdbx_keys() *and* mdbx_items() *will do unasked*

Description

A scan with no limit refuses to run when the database holds more records than this, rather than materializing the lot. One million keys is roughly 70 MB as an R character vector, and more again with their values: enough headroom for ordinary work, small enough to catch a database that was never meant to be read in one go.

Usage

```
mdbx_scan_max
```

Format

A single number.

Details

The guard applies only when limit is NULL. Any explicit limit is honoured, including limit = Inf to say "all of them, really".

Value

A length-one numeric vector, `1e6`. It is a constant rather than a function: its value is the number of records above which `mdbx_keys()` and `mdbx_items()` refuse a scan that was given no `limit`.

See Also

`mdbx_keys()`, `mdbx_items()`

Examples

```
mdbx_scan_max
```

`mdbx_txn_abort` *Abort a transaction*

Description

Discards the transaction's writes and ends it. Aborting is idempotent: doing it to an already-finished transaction does nothing, so it is safe to register with `on.exit()` alongside an explicit `mdbx_txn_commit()`.

Usage

```
mdbx_txn_abort(txn)
```

Arguments

`txn`	An `mdbx_txn` object, from `mdbx_txn_begin()`.

Value

`NULL`, invisibly.

See Also

`mdbx_txn_begin()`, `mdbx_txn_commit()`

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path)

txn <- mdbx_txn_begin(env, write = TRUE)
mdbx_txn_abort(txn)
mdbx_txn_abort(txn) # already aborted; does nothing

mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

mdbx_txn_begin *Begin a transaction*

Description

Starts a transaction against an open environment. Everything read or written happens inside one: reads see a consistent snapshot taken when the transaction began, and writes become visible to others only on commit.

Usage

```
mdbx_txn_begin(env, write = FALSE, flags = NULL)
```

Arguments

env	An mdbx_env object, from mdbx_env_open().
write	If TRUE, start a read-write transaction; the environment must not have been opened with readonly = TRUE.
flags	A character vector of 'libmdbx' transaction flag names, or NULL for none. "NOMETASYNC" and "NOSYNC" relax durability for this one transaction, exactly as the environment flags of the same names do for every transaction — see the Durability section of mdbx_flags(). "TRY" fails with MDBX_BUSY rather than waiting for another process's writer. All three apply to write transactions only.

Details

'libmdbx' binds a transaction to the thread that started it, so an environment supports **one transaction at a time per thread** — and R is single-threaded, so that means one at a time. Beginning a second while one is open is an error, never a deadlock or a hang. Distinct environments are independent, and concurrency comes from separate processes: many readers and one writer may hold transactions on the same environment simultaneously.

A transaction must be ended with mdbx_txn_commit() or mdbx_txn_abort(). One abandoned to the garbage collector is aborted, but that is a backstop rather than a plan: a live write transaction holds the writer lock until it ends. mdbx_with_read() and mdbx_with_write() handle this for you.

Value

An mdbx_txn object.

See Also

mdbx_txn_commit(), mdbx_txn_abort(), mdbx_with_write(), mdbx_flags(), mdbx-concurrency

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path)

txn <- mdbx_txn_begin(env)
txn
mdbx_txn_abort(txn)

mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

`mdbx_txn_commit` *Commit a transaction*

Description

Makes the transaction's writes durable and visible to other readers, and ends the transaction.

Usage

```
mdbx_txn_commit(txn)
```

Arguments

`txn`	An `mdbx_txn` object, from `mdbx_txn_begin()`.

Details

Unlike `mdbx_txn_abort()`, this is not idempotent: committing an already-finished transaction is an error, because the second call cannot do what it appears to. Note also that a commit which cannot complete is turned into an abort by 'libmdbx' — if this raises an error, the transaction has still ended, and its writes are gone.

Some failures inside a transaction end it there and then. A rejected key or value size (`mdbx_put()` returning `MDBX_BAD_VALSIZE`) is just that one operation failing, and the transaction carries on. A failure that exhausts the map (`MDBX_MAP_FULL`) instead marks the transaction unusable: every later operation fails with `MDBX_BAD_TXN`, and the commit reports that the whole transaction was rolled back rather than committed. `mdbx_txn_state()` reads `"failed"` from the moment that happens, and `"aborted"` once the rollback has ended it.

Value

`NULL`, invisibly.

See Also

`mdbx_txn_begin()`, `mdbx_txn_abort()`

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path)

txn <- mdbx_txn_begin(env, write = TRUE)
mdbx_txn_commit(txn)
mdbx_txn_state(txn)

mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

`mdbx_txn_state` *State of a transaction*

Description

What this reports is whether the transaction can still be used, not whether its native handle happens to be allocated. Three of the five answers exist because those differ: a transaction 'libmdbx' has marked erroneous, one poisoned by an assertion failure, and one inherited across a `fork()` are each refused by every operation, and reporting them as `"active"` described only this package's own bookkeeping.

Usage

```
mdbx_txn_state(txn)
```

Arguments

`txn`	An `mdbx_txn` object, from `mdbx_txn_begin()`.

Value

One of:

`"active"` Open and usable.

`"committed"` Ended by `mdbx_txn_commit()`; its writes are durable.

`"aborted"` Ended by `mdbx_txn_abort()`, by the garbage collector, or by its environment closing; its writes are gone.

`"failed"` Still open, but 'libmdbx' has marked it erroneous — `MDBX_MAP_FULL` is the usual cause. Every operation now fails with `MDBX_BAD_TXN` and committing reports a rollback, so the only thing left to do with it is end it.

`"poisoned"` Abandoned after a 'libmdbx' assertion failure, in this transaction or in the environment that owns it. Aborting it is still safe, and does not re-enter 'libmdbx'; afterwards it reads as `"aborted"` like any other ended transaction.

`"invalid"` The handle has been reclaimed, or was inherited across a `fork()` and belongs to another process — see mdbx-concurrency.

See Also

mdbx_txn_begin()

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path)

txn <- mdbx_txn_begin(env)
mdbx_txn_state(txn)
mdbx_txn_abort(txn)
mdbx_txn_state(txn)

mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

---

mdbx_version *Version of the bundled libmdbx*

Description

Reports the version of the vendored 'libmdbx' amalgamation that was compiled into this package. The sources are bundled, so this describes the library actually in use, not one found on the system.

Usage

```
mdbx_version()
```

Value

A list with integer components `major`, `minor`, `patch` and `tweak`; character components `describe` (the upstream git-describe string) and `commit`; and `build`, a list describing how the amalgamation was compiled into this package — `datetime`, `target` (the cpu/arch/system triplet), `compiler`, `options` (the 'libmdbx' build options in effect) and `flags`.

Examples

```
mdbx_version()$describe

# How the bundled sources were compiled here.
mdbx_version()$build$target
```

`mdbx_with_write` *Run code inside a transaction*

Description

Begins a transaction, calls `fun` with it, and ends it — whatever happens. `mdbx_with_write()` commits if `fun` returns normally and aborts if it throws; `mdbx_with_read()` always aborts, which for a read transaction simply releases the snapshot. Both use `on.exit()`, so the transaction is also ended if `fun` is interrupted.

Usage

```
mdbx_with_write(env, fun)

mdbx_with_read(env, fun)
```

Arguments

`env`	An `mdbx_env` object, from `mdbx_env_open()`.
`fun`	A function of one argument, called with the `mdbx_txn`.

Details

This is the recommended way to use transactions: it makes the "abandoned a write transaction and kept the writer lock" mistake unreachable.

Value

The value of `fun`.

See Also

`mdbx_txn_begin()`

Examples

```
path <- tempfile(fileext = ".mdbx")
env <- mdbx_env_open(path)

# Committed on normal return.
mdbx_with_write(env, function(txn) mdbx_txn_state(txn))

mdbx_with_read(env, function(txn) mdbx_txn_state(txn))

mdbx_env_close(env)
unlink(c(path, paste0(path, "-lck")))
```

Index