LMDB MANUAL

Contents

[in package LMDB]

- **[system]** **"lmdb"**

```
- _Version:_ 0.1
- _Description:_ Bindings to `lmdb`, the Lightning Memory-mapped Database.
- _Licence:_ MIT, see COPYING.
- _Author:_ Fernando Borretti <eudoxiahp@gmail.com>, James Anderson
↪  <james.anderson@setf.de>, Gábor Melis <mega@retes.hu>
- _Maintainer:_ Gábor Melis <mega@retes.hu>
- _Homepage:_ <https://github.com/melisgl/lmdb>
- _Bug tracker:_ <https://github.com/melisgl/lmdb/issues>
- _Source control:_ [GIT](https://github.com/melisgl/lmdb.git)
- *Depends on:* alexandria, bordeaux-threads, cl-reexport, [mgl-pax][6fdb], osicat,
↪  trivial-features, trivial-garbage, [trivial-utf-8][d9f2]
```

1 Links

Here is the official repository and the HTML documentation for the latest version.

2 Introduction

LMDB, the Lightning Memory-mapped Database, is an ACID key-value database with MVCC. It is a small C library ("C lmdb" from now on), around which `lmdb` is a Common Lisp wrapper. `lmdb` covers most of C lmdb's functionality, has a simplified API, much needed Safety checks, and comprehensive documentation.

Compared to other key-value stores, `lmdb`'s distuingishing features are:

- Transactions span multiple keys.
- Embedded. It has no server but can be used concurrently not only by multiple threads but by multiple OS processes, too.
- Extremely high read performance: millions of transactions per second.
- Very low maintenance.

Other notable things:

- With its default - the most durable - settings, it has average write performance, which is bottlenecked by `fsync()`.
- Readers don't block readers or writers, but there is at most one writer at a time.
- Extremely simple, crash-proof design.

- The entire database (called *environment*) is backed by a single memory-mapped file, with a copy-on-write B+ tree.
- No transaction log.
- It is very much like Berkeley DB done right, without the fluff and much improved administration.

Do read the Caveats, though. On the Lisp side, this library **will not work with virtual threads** because `lmdb`'s write locking is tied to native threads.

Using `lmdb` is easy:

```
(with-temporary-env (*env*)
  (let ((db (get-db "test")))
    (with-txn (:write t)
      (put db "k1" #(2 3))
      (print (g3t db "k1")) ; => #(2 3)
      (del db "k1"))))
```

More typically, the environment and databases are opened once so that multiple threads and transactions can access them:

```
(defvar *test-db*)

(unless *env*
  (setq *env* (open-env "/tmp/lmdb-test-env/" :if-does-not-exist :create))
  (setq *test-db* (get-db "test" :value-encoding :utf-8)))

(with-txn (:write t)
  (put *test-db* 1 "hello")
  (print (g3t *test-db* 1)) ; => "hello"
  (del *test-db* 1))
```

Note how `:value-encoding` sneaked in above. This was so to make `g3t` return a string instead of an octet vector.

`lmdb` treats keys and values as opaque byte arrays to be hung on a B+ tree, and only requires a comparison function to be defined over keys. `lmdb` knows how to serialize the types `(unsigned-byte 64)` and `string` (which are often used as keys so sorting must work as expected). Serialization of the rest of the datatypes is left to the client. See Encoding and decoding data for more.

3 Design and implementation

3.1 Safety

The lmdb C API trusts client code to respect its rules. Being C, managing object lifetimes is the biggest burden. There are also rules that are documented, but not enforced. This Lisp wrapper tries to enforce these rules itself and manage object lifetimes in a safe way to avoid data corruption. How and what it does is described in the following.

Environments

- `open-env` checks that the same path is not used in multiple open environments to prevent locking issues documented in Caveats.
- `close-env` waits until all active transactions are finished before actually closing the environment. Alternatively, if `open-env` was called with `:synchronized nil`, to avoid the overhead of synchronization, the environment is closed only when garbage collected.

Transactions

- Checks are made to detect illegal operations on parent transactions (see `lmdb-illegal-access-to-parent-txn-error`).
- Access to closed transactions is reliably detected.
- C `lmdb` allows read transactions to be used in multiple threads. The synchronization cost of performing this safely (i.e. without risking access to closed and freed transaction objects) is significant so this is not supported.

Databases

- mdb_dbi_open() is wrapped by `get-db` in a transaction and is protected by a mutex to comply with C lmdb's requirements:

```
A transaction that opens a database must finish (either
commit or abort) before another transaction may open it.
Multiple concurrent transactions cannot open the same
database.
```

- mdb_dbi_close() is too dangerous to be exposed as explained in the `get-db` documentation.
- For similar reasons, `drop-db` is wrapped in `with-env`.
- mdb_env_set_mapsize(), mdb_env_set_max_readers(), and mdb_env_set_maxdbs() are only available through `open-env` because they either require that there are no write transactions or do not work on open environments.

Cursors

- As even read transactions are restricted to a single thread, so are cursors. Using a cursor from a thread other than the one in which it was created (i.e. the thread of its transaction) raises `lmdb-cursor-thread-error`. In return for this restriction, access to cursors belonging to closed transactions is reliably detected.

Signal handling The C lmdb library handles system calls being interrupted (`eintr` and `eagain`), but unwinding the stack from interrupts in the middle of `lmdb` calls can leave the in-memory data structures such as transactions inconsistent. If this happens, their further use risks data corruption. For this reason, calls to `lmdb` are performed with interrupts disabled. For SBCL, this means `sb-sys:without-interrupts`. It is an error when compiling `lmdb` if an equivalent facility is not found in the Lisp implementation. A warning is signalled if no substitute is found for

`sb-sys:with-interrupts` because this makes the body of `with-env`, `with-txn`, `with-cursor` and similar uninterruptible.

Operations that do not modify the database (`g3t`, `cursor-first`, `cursor-value`, etc) are async unwind safe, and for performance they are called without the above provisions.

Note that the library is not reentrant, so don't call `lmdb` from signal handlers.

3.2 Deviations from the C lmdb API

The following are the most prominent deviations and omissions from the C lmdb API in addition to those listed in Safety.

Environments

- mdb_reader_list() is not implemented.
- mdb_env_copy() and its close kin are not yet implemented.

Transactions

- Read-only `with-txn`s are turned into noops when "nested" (unless `ignore-parent`).

Databases

- mdb_set_compare() and mdb_set_dupsort() are not exposed. If they are needed, implement a foreign comparison function and call `liblmdb:set-compare` or `liblmdb:set-dupsort` directly or perhaps change the encoding of the data.
- Working with multiple contiguous values with `dupfixed` is not yet implemented. This functionality would belong in `put`, `cursor-put`, `cursor-next` and `cursor-value`.
- `put`, `cursor-put` do not support the `RESERVE` flag.

4 Library versions

- **[function]** **lmdb-foreign-version**

  Return the version of the C lmdb library as a string like `0.9.26`.

  Wraps mdb_version().

- **[function]** **lmdb-binding-version**

  Return a string representing the version of C lmdb based on which the CFFI bindings were created. The version string has the same format as `lmdb-foreign-version`.

5 Environments

An environment (class `env`) is basically a single memory-mapped file holding all the data, plus some flags determining how we interact it. An environment can have multiple databases (class `db`), each of which is a B+ tree within the same file. An environment is like a database in a relational db, and the databases in it are like tables and indices. The terminology comes from Berkeley `db`.

5.1 Environments reference

- **[class]** **env**

  An environment object through which a memory-mapped data file can be accessed. Always to be created by `open-env`.

- **[reader]** **env-path** *env (:path)*

  The location of the memory-mapped file and the environment lock file.

- **[reader]** **env-max-dbs** *env (:max-dbs)*

  The maximum number of named databases in the environment. Currently a moderate number is cheap, but a huge number gets expensive: 7-120 words per transaction, and every `get-db` does a linear search of the opened database.

- **[reader]** **env-max-readers** *env (:max-readers)*

  The maximum number of threads/reader slots. See the documentation of the reader lock table for more.

- **[reader]** **env-map-size** *env (:map-size)*

  Specifies the size of the data file in bytes.

- **[reader]** **env-mode** *env (:mode)*

- **[reader]** **env-flags** *env (:flags)*

  A plist of the options as captured by `open-env`. For example, `(:fixed-map nil :subdir t ...)`.

5.2 Opening and closing environments

- **[variable]** ***env-class*** *env*

  The default class `open-env` instaniates. Must be a subclass of `env`. This provides a way to associate application specific data with `env` objects.

- **[function]** **open-env** *path &key (class *env-class*) (if-does-not-exist :error) (synchronized t) (max-dbs 1) (max-readers 126) (map-size (* 1024 1024)) (mode 436) (subdir t) (sync t) (meta-sync t) read-only (tls t) (read-ahead t) (lock t) (mem-init t) fixed-map write-map map-async*

Create an env object through which the lmdb environment can be accessed and open it. To prevent corruption, an error is signalled if the same data file is opened multiple times. However, the checks performed do not work on remote filesystems (see env-path).

lmdb-error is signalled if opening the environment fails for any other reason.

Unless explicitly noted, none of the arguments persist (i.e. they are not saved in the data file).

path is the filesystem location of the environment files (see subdir below for more). Do not use lmdb data files on remote filesystems, even between processes on the same host. This breaks flock() on some OSes, possibly memory map sync, and certainly sync between programs on different hosts.

if-does-not-exist determines what happens if env-path does not exists:

- :error: An error is signalled.
- :create: A new memory-mapped file is created ensuring that all containing directories exist.
- nil: Return nil without doing anything.

See close-env for the description of synchronized.

- max-dbs: The maximum number of named databases in the environment. Currently a moderate number is cheap, but a huge number gets expensive: 7-120 words per transaction, and every get-db does a linear search of the opened database.
- map-size: Specifies the size of the data file in bytes. The new size takes effect immediately for the current process, but will not be persisted to any others until a write transaction has been committed by the current process. Also, only map size increases are persisted into the environment. If the map size is increased by another process, and data has grown beyond the range of the current mapsize, starting a new transaction (see with-txn) will signal lmdb-map-resized-error. If zero is specified for map-size, then the persisted size is used from the data file. Also see lmdb-map-full-error.
- mode: Unix file mode for files created. The default is #o664. Has no effect when opening an existing environment.

The rest of the arguments correspond to lmdb environment flags and are available in the plist env-flags.

- subdir: If subdir, then the path is a directory which holds the data.mdb and the lock.mdb files. If subdir is nil, the path is the filename of the data file and the lock file has the same name plus a -lock suffix.
- sync: If nil, don't fsync after commit. 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 sync-env is called. However, if the filesystem preserves write order (very few do) and the write-map (currently unsupported) flag is not used, 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.

- `meta-sync`: If `nil`, flush system buffers to disk only once per transaction, but omit the metadata flush. Defer that until the system flushes files to disk, the next commit of a non-read-only transaction or `sync-env`. This optimization maintains database integrity, but a system crash may undo the last committed transaction. I.e. it preserves the ACI (atomicity, consistency, isolation) but not D (durability) database property.
- `read-only`: Map the data file in read-only mode. It is an error to try to modify anything in it.
- `tls`: Setting it to `nil` allows each OS thread to have multiple read-only transactions (see `with-txn`'s `ignore-parent` argument). It also allows and transactions not to be tied to a single thread, but that's quite dangerous, see Safety.
- `read-ahead`: Turn off readahead as in `madvise(MADV_RANDOM)`. Most operating systems perform read-ahead 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. This option is not implemented on Windows.
- `lock`: Data corruption lurks here. If `nil`, 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.
- `mem-init`: If `nil`, don't initialize `malloc`ed 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`ed 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 `write-map`, which writes directly to the mmap instead of using malloc for pages.
- `fixed-map` (experimental): This flag must be specified when creating the environment and is stored persistently in the data file. If successful, the memory map will always reside at the same virtual address and pointers used to reference data items in the database will be constant across multiple invocations. This option may not always work, depending on how the operating system has allocated memory to shared libraries and other uses.

Unsupported flags (an error is signalled if they are changed from their default values):

- `write-map`: Use a writable memory map unless `read-only` is set. This is faster and uses fewer mallocs, but loses protection from application bugs like wild pointer writes

and other bad updates into the database. Incompatible with nested transactions. This may be slightly faster for `dbs` that fit entirely in RAM, but is slower for `dbs` larger than RAM. Do not mix processes with and without `write-map` on the same environment. This can defeat durability (`sync-env`, etc).

- `map-async`: When using `write-map`, use asynchronous flushes to disk. As with `sync nil`, a system crash can then corrupt the database or lose the last transactions. Calling #sync ensures on-disk database integrity until next commit.

Open environments have a finalizer attached to them that takes care of freeing foreign resources. Thus, the common idiom:

```
(setq *env* (open-env "some-path"))
```

is okay for development, too. No need to always do `with-env`, which does not mesh with threads anyway.

Wraps mdb_env_create() and mdb_env_open().

- **[function]** **close-env** *env &key force*

  Close `env` and free the memory. Closing an already closed `env` has no effect.

  Since accessing Transactions, Databases and Cursors after closing their environment would risk database curruption, `close-env` makes sure that they are not in use. There are two ways this can happen:

  - If `env` was opened `:synchronized` (see `open-env`), then `close-env` waits until there are no active transactions in `env` before closing it. This requires synchronization and introduces some overhead, which might be noticable for workloads involving lots of quick read transactions. It is an `lmdb-error` to attempt to close an environment in a `with-txn` to avoid deadlocks.
  - On the other hand, if `synchronized` was `nil`, then - unless `force` is true - calling `close-env` signals an `lmdb-error` to avoid the Safety issues involved in closing the environment. Environments opened with `:synchronized nil` are only closed when they are garbage collected and their finalizer is run. Still, for production it might be worth it to gain the last bit of performance.

  Wraps mdb_env_close().

- **[variable]** ***env*** *nil*

  The default `env` for macros and function that take an environment argument.

- **[macro]** **with-env** *(env path &rest open-env-args) &body body*

  Bind the variable `env` to a new enviroment returned by `open-env` called with `path` and `open-env-args`, execute `body`, and `close-env`. The following example binds the default environment:

```
(with-env (*env* "/tmp/lmdb-test" :if-does-not-exist :create)
  ...)
```

- **[function]** **open-env-p** *env*

  See if `env` is open, i.e. `open-env` has been called on it without a corresponding `close-env`.

5.3 Miscellaneous environment functions

- **[function]** **check-for-stale-readers** *&optional (env *env*)*

  Check for stale entries in the reader lock table. See Caveats. This function is called automatically by `open-env`. If other OS processes or threads accessing `env` abort without closing read transactions, call this function periodically to get rid off them. Alternatively, close all environments accessing the data file.

  Wraps mdb_reader_check().

- **[function]** **env-statistics** *&optional (env *env*)*

  Return statistics about `env` as a plist.

  - `:page-size`: The size of a database page in bytes.
  - `:depth`: The height of the B-tree.
  - `:branch-pages`: The number of internal (non-leaf) pages.
  - `:leaf-pages`: The number of leaf pages.
  - `:overflow-pages`: The number of overflow pages.
  - `:entries`: The number of data items.

  Wraps mdb_env_stat().

- **[function]** **env-info** *&optional (env *env*)*

  Return information about `env` as a plist.

  - `:map-address`: Address of memory map, if fixed (see `open-env`'s `fixed-map`).
  - `:map-size`: Size of the memory map in bytes.
  - `:last-page-number`: Id of the last used page.
  - `:last-txn-id`: Id of the last committed transaction.
  - `:maximum-readers`: The number of reader slots.
  - `:n-readers`: The number of reader slots current used.

  Wraps mdb_env_info().

- **[function]** **sync-env** *&optional (env *env*)*

  Flush the data buffers to disk as in calling `fsync()`. When `env` had been opened with `:sync nil` or `:meta-sync nil`, this may be handy to force flushing the OS buffers to disk, which avoids potential durability and integrity issues.

  Wraps mdb_env_sync().

- **[function]** **env-max-key-size** *&optional (env *env*)*

  Return the maximum size of keys and `dupsort` data in bytes. Depends on the compile-time constant `mdb_maxkeysize` in the C library. The default is 511. If this limit is exceeded `lmdb-bad-valsize-error` is signalled.

  Wraps mdb_env_get_maxkeysize().

- **[macro]** **with-temporary-env** *(env &rest open-env-args) &body body*

  Run `body` with an open temporary environment bound to `env`. In more detail, create an environment in a fresh temporary directory in an OS specific location. `open-env-args` is a list of keyword arguments and values for `open-env`. This macro is intended for testing and examples.

```
(with-temporary-env (*env*)
  (let ((db (get-db "test")))
    (with-txn (:write t)
      (put db "k1" #(2 3))
      (print (g3t db "k1")) ; => #(2 3)
      (del db "k1"))))
```

  Since data corruption in temporary environments is not a concern, unlike `with-env`, `with-temporary-env` closes the environment even if it was opened with `:synchronized nil` (see `open-env` and `close-env`).

6 Transactions

The `lmdb` environment supports transactional reads and writes. By default, these provide the standard ACID (atomicity, consistency, isolation, durability) guarantees. Writes from a transaction are not immediately visible to other transactions. When the transaction is committed, all its writes become visible atomically for future transactions even if Lisp crashes or there is power failure. If the transaction is aborted, its writes are discarded.

Transactions span the entire environment (see `env`). All the updates made in the course of an update transaction - writing records across all databases, creating databases, and destroying databases - are either completed atomically or rolled back.

Write transactions can be nested. Child transactions see the uncommitted writes of their parent. The child transaction can commit or abort, at which point its writes become visible to the parent transaction or are discarded. If the parent aborts, all of the writes performed in the context of the parent, including those from committed child transactions, are discarded.

- **[macro]** **with-txn** *(&key (env '*env*) write ignore-parent (sync t) (meta-sync t)) &body body*

  Start a transaction in `env`, execute `body`. Then, if the transaction is open (see `open-txn-p`) and `body` returned normally, attempt to commit the transaction. Next, if `body` performed a non-local exit or committing failed, but the transaction is still open, then abort it. It is explicitly allowed to call `commit-txn` or `abort-txn` within `with-txn`.

Transactions provide ACID guarantees (with `sync` and `meta-sync` both on). They span the entire environment, they are not specific to individual `db`.

  - If `write` is `nil`, the transaction is read-only and no writes (e.g. `put`) may be performed in the transaction. On the flipside, many read-only transactions can run concurrently (see `env-max-readers`), while write transactions are mutually exclusive. Furthermore, the single write transaction can also run concurrently with read transactions, just keep in mind that read transactions hold on to the state of the environment at the time of their creation and thus prevent pages since replaced from being reused.
  - If `ignore-parent` is true, then in an enclosing `with-txn`, instead of creating a child transaction, start an independent transaction.
  - If `sync` is `nil`, then no flushing of buffers will take place after a commit as if the environment had been opened with `:sync nil`.
  - Likewise, `meta-sync` is the per-transaction equivalent of the `open-env`'s `meta-sync`.

Also see Nesting transactions.

Wraps mdb_txn_begin().

- **[glossary-term]** **active transaction**

  The active transaction in some environment and thread is the transaction of the innermost `with-txn` being executed in the thread that belongs to the environment. In most cases, this is simply the enclosing `with-txn`, but if `with-txn`s with different `:env` arguments are nested, then it may not be:

```
(with-temporary-env (env)
  (let ((db (get-db "db" :env env)))
    (with-temporary-env (inner-env)
      (with-txn (:env env :write t)
        (with-txn (:env inner-env)
          (put db #(1) #(2)))))))
```

  In the above example, `db` is known to belong to `env` so although the immediately enclosing transaction belongs to INNER-ENV, `put` is executed in context of the outer, write transaction because that's the innermost in `env`.

  Operations that require a transaction always attempt to use the active transaction even if it is not open (see `open-txn-p`).

- **[function]** **open-txn-p** *&optional env*

  See if there is an active transaction and it is open, i.e. `commit-txn` or `abort-txn` have not been called on it. Also, `reset-txn` without a corresponding `renew-txn` closes the transaction.

- **[function]** **txn-id**

  The ID of `txn`. IDs are integers incrementing from 1. For a read-only transaction, this corresponds to the snapshot being read; concurrent readers will frequently have the same

transaction ID. Only committed write transactions increment the ID. If a transaction aborts, the ID may be re-used by the next writer.

- **[function]** **commit-txn** *&optional env*

  Commit the innermost enclosig transaction (or active transaction belonging to `env` if `env` is specified) or signal an error if it is not open. If `txn` is not nested in another transaction, committing makes updates performed visible to future transactions. If `txn` is a child transaction, then committing makes updates visible to its parent only. For read-only transactions, committing releases the reference to a historical version environment, allowing reuse of pages replaced since.

  Wraps mdb_txn_commit().

- **[function]** **abort-txn** *&optional env*

  Close `txn` by discarding all updates performed, which will then not be visible to either parent or future transactions. Aborting an already closed transaction is a noop. Always succeeds.

  Wraps mdb_txn_abort().

- **[function]** **renew-txn** *&optional env*

  Renew `txn` that was reset by `reset-txn`. This acquires a new reader lock that had been released by `reset-txn`. After renewal, it is as if `txn` had just been started.

  Wraps mdb_txn_renew().

- **[function]** **reset-txn** *&optional env*

  Abort the open, read-only `txn`, release the reference to the historical version of the environment, but make it faster to start another read-only transaction with `renew-txn`. This is accomplished by not deallocating some data structures, and keeping the slot in the reader table. Cursors opened within the transaction must not be used again, except if renewed (see RENEW-CURSOR). If `txn` is an open, read-only transaction, this function always succeeds.

  Wraps mdb_txn_reset().

6.1 Nesting transactions

When `with-txn`s are nested (i.e. one is executed in the dynamic extent of another), we speak of nested transactions. Transaction can be nested to arbitrary levels. Child transactions may be committed or aborted independently from their parent transaction (the immediately enclosing `with-txn`). Committing a child transaction only makes the updates made by it visible to the parent. If the parent then aborts, the child's updates are aborted too. If the parent commits, all child transactions that were not aborted are committed, too.

Actually, the C lmdb library only supports nesting write transactions. To simplify usage, the Lisp side turns read-only `with-txn`s nested in another `with-txn`s into noops.

```
(with-temporary-env (*env*)
  (let ((db (get-db "test" :value-encoding :uint64)))
```

```
;; Create a top-level write transaction.
(with-txn (:write t)
  (put db "p" 0)
  ;; First child transaction
  (with-txn (:write t)
    ;; Writes of the parent are visible in children.
    (assert (= (g3t db "p") 0))
    (put db "c1" 1))
  ;; Parent sees what the child committed (but it's not visible to
  ;; unrelated transactions).
  (assert (= (g3t db "c1") 1))
  ;; Second child transaction
  (with-txn (:write t)
    ;; Sees writes from the parent that came from the first child.
    (assert (= (g3t db "c1") 1))
    (put db "c1" 2)
    (put db "c2" 2)
    (abort-txn)))
;; Create a top-level read transaction to check what was committed.
(with-txn ()
  ;; Since the second child aborted, its writes are discarded.
  (assert (= (g3t db "p") 0))
  (assert (= (g3t db "c1") 1))
  (assert (null (g3t db "c2"))))))
```

`commit-txn`, `abort-txn`, and `reset-txn` all close the active transaction (see `open-txn-p`). When the active transaction is not open, database operations such as `g3t`, `put`, `del` signal `lmdb-bad-txn-error`. Furthermore, any Cursors created in the context of the transaction will no longer be valid (but see `cursor-renew`).

An `lmdb` parent transaction and its cursors must not issue operations other than `commit-txn` and `abort-txn` while there are active child transactions. As the Lisp side does not expose transaction objects directly, performing Basic operations in the parent transaction is not possible, but it is possible with Cursors as they are tied to the transaction in which they were created.

`ignore-parent` true overrides the default nesting semantics of `with-txn` and creates a new top-level transaction, which is not a child of the enclosing `with-txn`.

- Since `lmdb` is single-writer, on nesting an `ignore-parent` write transaction in another write transaction, `lmdb-bad-txn-error` is signalled to avoid the deadlock.
- Nesting a read-only `with-txn` with `ignore-parent` in another read-only `with-txn` is `lmdb-bad-rslot-error` error with the `tls` option because it would create two read-only transactions in the same thread.

Nesting a read transaction in another transaction would be an `lmdb-bad-rslot-error` according to the C lmdb library, but a read-only `with-txn` with `ignore-parent nil` nested in another `with-txn` is turned into a noop so this edge case is papered over.

7 Databases

7.1 The unnamed database

`lmdb` has a default, unnamed database backed by a B+ tree. This db can hold normal key-value pairs and named databases. The unnamed database can be accessed by passing `nil` as the database name to `get-db`. There are some restrictions on the flags of the unnamed database, see `lmdb-incompatible-error`.

7.2 `dupsort`

A prominent feature of `lmdb` is the ability to associate multiple sorted values with keys, which is enabled by the `dupsort` argument of `get-db`. Just as a named database is a B+ tree associated with a key (its name) in the B+ tree of the unnamed database, so do these sorted duplicates form a B+ tree under a key in a named or the unnamed database. Among the Basic operations, `put` and `del` are equipped to deal with duplicate values, but `g3t` is too limited, and Cursors are needed to make full use of `dupsort`.

When using this feature the limit on the maximum key size applies to duplicate data, as well. See `env-max-key-size`.

7.3 Database API

- **[variable]** ***db-class*** *db*

  The default class that `get-db` instantiates. Must a subclass of `db`. This provides a way to associate application specific data with `db` objects.

- **[function]** **get-db** *name &key (class *db-class*) (env *env*) (if-does-not-exist :create) key-encoding value-encoding integer-key reverse-key dupsort integer-dup reverse-dup dupfixed*

  Open the database with `name` in the open environment `env`, and return a `db` object. If `name` is `nil`, then the The unnamed database is opened.

  If `get-db` is called with the same name multiple times, the returned `db` objects will be associated with the same database (although they may not be `eq`). The first time `get-db` is called with any given name and environment, it must not be from an open transaction. This is because `get-db` starts a transaction itself to comply with C lmdb's requirements on mdb_dbi_open() (see Safety). Since dbi handles are cached within `env`, subsequent calls do not involve `mdb_dbi_open()` and are thus permissible within transactions.

  `class` designates the class which will instantiated. See `*db-class*`.

  If `if-does-not-exist` is `:create`, then a new named database is created. If `if-does-not-exist` is `:error`, then an error is signalled if the database does not exists.

  `key-encoding` and `value-encoding` are both one of `nil`, `:uint64`, `:octets` or `:utf-8`. `key-encoding` is set to `:uint64` when `integer-key` is true. `value-encoding` is set to `:uint64` when `integer-dup` is true. Note that changing the encoding does *not* reencode already existing data. See Encoding and decoding data for the full semantics.

`get-db` may be called more than once with the same `name` and `env`, and the returned `db` objects will have the same underlying C lmdb database, but they may have different `key-encoding` and `value-encoding`.

The following flags are for database creation, they do not have any effect in subsequent calls (except for the The unnamed database).

- `integer-key`: Keys in the database are C `unsigned` or `size_t` integers encoded in native byte order. Keys must all be either `unsigned` or `size_t`, they cannot be mixed in a single database.
- `reverse-key`: Keys are strings to be compared in reverse order, from the end of the strings to the beginning. By default, keys are treated as strings and compared from beginning to end.
- `dupsort`: Duplicate keys may be used in the database (or, from another perspective, keys may have multiple values, stored in sorted order). By default, keys must be unique and may have only a single value. Also, see `dupsort`.
- `integer-dup`: This option specifies that duplicate data items are binary integers, similarly to `integer-key`. Only matters if `dupsort`.
- `reverse-dup`: This option specifies that duplicate data items should be compared as strings in reverse order. Only matters if `dupsort`.
- `dupfixed`: This flag may only be used in combination `dupsort`. When true, data items for this database must all be the same size, which allows further optimizations in storage and retrieval. Currently, the wrapper functions that could take advantage of this (e.g. `put`, `cursor-put`, `cursor-next` and `cursor-value`), do not.

No function to close a database (an equivalent to mdb_dbi_close()) is provided due to subtle races and corruption it could cause when an `MDB_dbi` (unsigned integer, similar to an fd) is assigned by a subsequent open to another named database.

Wraps mdb_dbi_open().

- **[class]** **db**

  A database in an environment (class `env`). Always to be created by `get-db`.

- **[reader]** **db-name** *db (:name)*

  The name of the database.

- **[reader]** **db-key-encoding** *db (:key-encoding)*

  The `encoding` that was passed as `key-encoding` to `get-db`.

- **[reader]** **db-value-encoding** *db (:value-encoding)*

  The `encoding` that was passed as `value-encoding` to `get-db`.

- **[function]** **drop-db** *name path &key open-env-args (delete t)*

Empty the database with `name` in the environment denoted by `path`. If `delete`, then delete the database. Since closing a database is dangerous (see `get-db`), `drop-db` opens and closes the environment itself.

Wraps mdb_drop().

- **[function]** **db-statistics** *db*

  Return statistics about the database.

  Wraps mdb_stat().

8 Encoding and decoding data

In the C lmdb library, keys and values are opaque byte vectors only ever inspected internally to maintain the sort order (of keys and also duplicate values if `dupsort`). The client is given the freedom and the responsibility to choose how to perform conversion to and from byte vectors.

`lmdb` exposes this full flexibility while at the same time providing reasonable defaults for the common cases. In particular, with the `key-encoding` and `value-encoding` arguments of `get-db`, the data (meaning the key or value here) encoding can be declared explicitly.

Even if the encoding is undeclared, it is recommended to use a single type for keys (and duplicate values) to avoid unexpected conflicts that could arise, for example, when the UTF-8 encoding of a string and the `:uint64` encoding of an integer coincide. The same consideration doubly applies to named databases, which share the key space with normal key-value pairs in the default database (see The unnamed database).

Together, `:uint64` and `:utf-8` cover the common cases for keys. They trade off dynamic typing for easy sortability (using the default C lmdb behaviour). On the other hand, when sorting is not concern (either for keys and values), serialization may be done more freely. For this purpose, using an encoding of `:octets` or `nil` with cl-conspack is recommended because it works with complex objects, it encodes object types, it is fast and space-efficient, has a stable specification and an alternative implementation in C. For example:

```
(with-temporary-env (*env*)
  (let ((db (get-db "test")))
    (with-txn (:write t)
      (put db "key1" (cpk:encode (list :some "stuff" 42)))
      (cpk:decode (g3t db "key1")))))
=> (:SOME "stuff" 42)
```

Note that multiple `db` objects with different encodings can be associated with the same C lmdb database, which declutters the code:

```
(defvar *cpk-encoding*
  (cons #'cpk:encode (alexandria:compose #'cpk:decode #'mdb-val-to-octets)))

(with-temporary-env (*env*)
  (let ((next-id-db (get-db "test" :key-encoding *cpk-encoding*
                                   :value-encoding :uint64))
        (db (get-db "test" :key-encoding *cpk-encoding*
```

```
                            :value-encoding *cpk-encoding*)))
    (with-txn (:write t)
      (let ((id (or (g3t next-id-db :next-id) 0)))
        (put next-id-db :next-id (1+ id))
        (put db id (list :some "stuff" 42))
        (g3t db id)))))
=> (:SOME "stuff" 42)
=> T
```

- **[type]** **encoding**

  The following values are supported:

  - `:uint64`: Data to be encoded must be of type `(unsigned-byte 64)`, which is then encoded as an 8 byte array in *native* byte order with `uint64-to-octets`. The reverse transformation takes place when returning values. This is the encoding used for `integer-key` and `integer-dup` `db`s.
  - `:octets`: Note the plural. Data to be encoded (e.g. `key` argument of `g3t`) must be a 1D byte array. If its element type is `(unsigned-byte 8)`, then the data can be passed to the foreign code more efficiently, but declaring the element type is not required. For example, `vector`s can be used as long as the actual elements are of type `(unsigned-byte 8)`. Foreign byte arrays to be decoded (e.g. the value returned by `g3t`) are returned as `octets`.
  - `:utf-8`: Data to be encoded must be a string, which is converted to octets by TRIVIAL-UTF-8. Null-terminated. Foreign byte arrays are decoded the same way.
  - `nil`: Data is encoded using the default encoding according to its Lisp type: strings as `:utf-8`, vectors as `:octets`, `(unsigned-byte 64)` as `:uint64`. Decoding is always performed as `:octets`.
  - A `cons`: Data is encoded by the function in the `car` of the cons and decoded by the function in the `cdr`. For example, `:uint64` is equivalent to `(cons #'uint64-to-octets #'mdb-val-to-uint64)`.

- **[macro]** **with-mdb-val-slots** *(%bytes size mdb-val) &body body*

  Bind `%bytes` and `size` locally to the corresponding slots of `mdb-val`. `mdb-val` is an opaque handle for a foreign `MDB_val` struct, that holds the pointer to a byte array and the number of bytes in the array. This macro is needed to access the foreign data in a function used as `*key-decoder*` or `*value-decoder*`. `mdb-val` is dynamic extent, so don't hold on to it. Also, the pointer to which `%bytes` is bound is valid only within the context of current top-level transaction.

- **[type]** **octets** *&optional (size '*)*

  A 1D `simple-array` of `(unsigned-byte 8)`.

- **[function]** **mdb-val-to-octets** *mdb-val*

  A utility function provided for writing `*key-decoder*` and `*value-decoder*` functions. It returns a Lisp octet vector that holds the same bytes as `mdb-val`.

- **[function]** **uint64-to-octets** *n*

  Convert an `(unsigned-byte 64)` to `octets` of length 8 taking the native byte order representation of `n`. Suitable as a `*key-encoder*` or `*value-encoder*`.

- **[function]** **octets-to-uint64** *octets*

  The inverse of `uint64-to-octets`. Use `mdb-val-to-uint64` as a `*key-decoder*` or `*value-decoder*`.

- **[function]** **mdb-val-to-uint64** *mdb-val*

  Like `octets-to-uint64`, but suitable for `*key-decoder*` or `*value-decoder*` that decodes unsigned 64 bit integers in native byte order. This function is called automatically when the encoding is known to require it (see `get-db`'s `integer-key`, `:value-encoding`, etc).

- **[function]** **string-to-octets** *string*

  Convert `string` to `octets` by encoding it as UTF-8 with null termination. Suitable as a `*key-encoder*` or `*value-encoder*`.

- **[function]** **octets-to-string** *octets*

  The inverse of `string-to-octets`. Use `mdb-val-to-string` as a `*key-decoder*` or `*value-decoder*`.

- **[function]** **mdb-val-to-string** *mdb-val*

  Like `octets-to-string`, but suitable as a `*key-decoder*` or `*value-decoder*`.

8.1 Overriding encodings

Using multiple `db` objects with different encodings is the recommended practice (see the example in Encoding and decoding data), but when that is inconvenient, one can override the encodings with the following variables.

- **[variable]** ***key-encoder*** *nil*

  A function designator, `nil` or an `encoding`. If non-`nil`, it overrides the encoding method determined by `key-encoding` (see `get-db`). It is called with a single argument, the key, when it is to be converted to an octet vector.

- **[variable]** ***key-decoder*** *nil*

  A function designator, `nil` or an `encoding`. If non-`nil`, it is called with a single `mdb-val` argument (see `with-mdb-val-slots`), that holds a pointer to data to be decoded and its size. This function is called whenever a key is to be decoded and overrides the `key-encoding` argument of `get-db`.

  For example, if we are only interested in the length of the value and want to avoid creating a lisp vector on the heap, we can do this:

```
(with-temporary-env (*env*)
  (let ((db (get-db "test")))
    (with-txn (:write t)
      (put db "key1" "abc")
      (let ((*value-decoder* (lambda (mdb-val)
                               (with-mdb-val-slots (%bytes size mdb-val)
                                 (declare (ignore %bytes))
                                 ;; Take null termination into account.
                                 (1- size)))))
        (g3t db "key1")))))
=> 3
=> T
```

- **[variable]** ***value-encoder*** *nil*

  Like `*key-encoder*`, but for values.

- **[variable]** ***value-decoder*** *nil*

  Like `*key-decoder*`, but for values.

  Apart from performing actual decoding, the main purpose of `*value-decoder*`, one can also pass the foreign data on to other foreign functions such as `write()` directly from the decoder function and returning a constant such as `t` to avoid consing.

9 Basic operations

- **[function]** **g3t** *db key*

  Return the value from `db` associated with `key` and `t` as the second value. If `key` is not found in `db`, then `nil` is returned. If `db` supports `dupsort`, then the first value for `key` will be returned. Retrieval of other values requires the use of Cursors.

  This function is called `g3t` instead of `get` to avoid having to shadow `cl:get` when importing the `lmdb` package. On the other hand, importing the LMDB+ package, which has `lmdb::get` exported, requires some shadowing.

  The LMDB+ package is like the `lmdb` package, but it has `#'lmdb:g3t` fbound to `lmdb+:g3t` so it probably needs shadowing to avoid conflict with `cl:get`:

  ```
  (defpackage lmdb/test
    (:shadow #:get)
    (:use #:cl #:lmdb+))
  ```

  Wraps mdb_get().

- **[function]** **put** *db key value &key (overwrite t) (dupdata t) append append-dup (key-exists-error-p t)*

  Add a `key`, `value` pair to `db` within `txn` (which must support writes). Returns `t` on success.

  - `overwrite`: If `nil`, signal `lmdb-key-exists-error` if `key` already appears in `db`.

- `dupdata`: If `nil`, signal `lmdb-key-exists-error` if the `key`, `value` pair already appears in `db`. Has no effect if `db` doesn't have `dupsort`.
- `append`: Append the `key`, `value` pair to the end of `db` instead of finding `key`'s location in the B+ tree by performing comparisons. The client effectively promises that keys are inserted in sort order, which allows for fast bulk loading. If the promise is broken, a `lmdb-key-exists-error` is signalled.
- `append-dup`: The client promises that duplicate values are inserted in sort order. If the promise is broken, a `lmdb-key-exists-error` is signalled.
- If `key-exists-error-p` is `nil`, then instead of signalling `lmdb-key-exists-error` return `nil`.

May signal `lmdb-map-full-error`, `lmdb-txn-full-error`, `lmdb-txn-read-only-error`.

Wraps mdb_put().

- **[function]** **del** *db key &key value*

Delete `key` from `db`. Returns `t` if data was deleted, `nil` otherwise. If `db` supports sorted duplicates (`dupsort`), then `value` is taken into account: if it's `nil`, then all duplicate values for `key` are deleted, if it's not `nil`, then only the matching value. May signal `lmdb-txn-read-only-error`.

Wraps mdb_del().

10 Cursors

- **[macro]** **with-cursor** *(var db) &body body*

Bind `var` to a fresh `cursor` on `db`. Execute `body`, then close the cursor. Within the dynamic extent of `body`, this will be the default cursor. The cursor is tied to the active transaction.

`lmdb-cursor-thread-error` is signalled if the cursor is accessed from threads other than the one in which it was created.

Wraps mdb_cursor_open() and mdb_cursor_close().

- **[macro]** **with-implicit-cursor** *(db) &body body*

Like `with-cursor` but the cursor object is not accessible directly, only through the default cursor mechanism. The cursor is stack-allocated, which eliminates the consing of `with-cursor`. Note that stack allocation of cursors in `with-cursor` would risk data corruption if the cursor were accessed beyond its dynamic extent.

Use `with-implicit-cursor` instead of `with-cursor` if a single cursor at a time will suffice. Conversely, use `with-cursor` if a second cursor is needed. That is, use

```
(with-implicit-cursor (db)
  (cursor-set-key 1))
```

but when two cursors iterate in an interleaved manner, use `with-cursor`:

```
(with-cursor (c1 db)
  (with-cursor (c2 db)
    (cursor-first c1)
    (cursor-last c2)
    (if (some-pred (cursor-value c1) (cursor-value c2))
        (cursor-next c1)
        (cursor-prev c2))
    ...))
```

Wraps mdb_cursor_open() and mdb_cursor_close().

- **[structure]** **cursor**
- **[structure-accessor]** **cursor-db** *cursor*
- **[glossary-term]** **default cursor**

  All operations, described below, that take cursor arguments accept `nil` instead of a `cursor` object, in which case the cursor from the immediately enclosing `with-cursor` or `with-implicit-cursor` is used. This cursor is referred to as the *default cursor*.

  To reduce syntactic clutter, some operations thus make cursor arguments `&optional`. When this is undesirable - because there are keyword arguments as well - the cursor may be a required argument as in `cursor-put`. Still `nil` can be passed explicitly.

10.1 Positioning cursors

The following functions *position* or *initialize* a cursor while returning the value (*a* value with `dupsort`) associated with a key, or both the key and the value. Initialization is successful if there is the cursor points to a key-value pair, which is indicated by the last return value being `t`.

- **[function]** **cursor-first** *&optional cursor*

  Move `cursor` to the first key of its database. Return the key, the value and `t`, or `nil` if the database is empty. If `dupsort`, position `cursor` on the first value of the first key.

  Wraps mdb_cursor_get() with MDB_FIRST.

- **[function]** **cursor-first-dup** *&optional cursor*

  Move `cursor` to the first duplicate value of the current key. Return the value and `t`. Return `nil` if `cursor` is not positioned.

  Wraps mdb_cursor_get() with MDB_FIRST_DUP.

- **[function]** **cursor-last** *&optional cursor*

  Move `cursor` to the last key of its database. Return the key, the value and `t`, or `nil` if the database is empty. If `dupsort`, position `cursor` on the last value of the last key.

  Wraps mdb_cursor_get() with MDB_LAST.

- **[function]** **cursor-last-dup** *&optional cursor*

  Move `cursor` to the last duplicate value of the current key. Return the value and `t`. Return `nil` if `cursor` is not positioned.

  Wraps mdb_cursor_get() with MDB_LAST_DUP.

- **[function]** **cursor-next** *&optional cursor*

  Move `cursor` to the next key-value pair of its database and return the key, the value, and `t`. Return `nil` if there is no next item. If `dupsort`, position `cursor` on the next value of the current key if exists, else the first value of next key. If `cursor` is uninitialized, then `cursor-first` is called on it first.

  Wraps mdb_cursor_get() with MDB_NEXT.

- **[function]** **cursor-next-nodup** *&optional cursor*

  Move `cursor` to the first value of next key pair of its database (skipping over duplicate values of the current key). Return the key, the value and `t`. Return `nil` if there is no next item. If `cursor` is uninitialized, then `cursor-first` is called on it first.

  Wraps mdb_cursor_get() with MDB_NEXT_NODUP.

- **[function]** **cursor-next-dup** *&optional cursor*

  Move `cursor` to the next value of current key pair of its database. Return the value and `t`. Return `nil` if there is no next value. If `cursor` is uninitialized, then `cursor-first` is called on it first.

  Wraps mdb_cursor_get() with MDB_NEXT_DUP.

- **[function]** **cursor-prev** *&optional cursor*

  Move `cursor` to the previous key-value pair of its database. Return the key, the value and `t`. Return `nil` if there is no previous item. If `dupsort`, position `cursor` on the previous value of the current key if exists, else the last value of previous key. If `cursor` is uninitialized, then `cursor-last` is called on it first.

  Wraps mdb_cursor_get() with MDB_PREV.

- **[function]** **cursor-prev-nodup** *&optional cursor*

  Move `cursor` to the last value of previous key pair of its database (skipping over duplicate values of the current and the previous key). Return the key, the value, and `t`. Return `nil` if there is no prev item. If `cursor` is uninitialized, then `cursor-last` is called on it first.

  Wraps mdb_cursor_get() with MDB_PREV_NODUP.

- **[function]** **cursor-prev-dup** *&optional cursor*

  Move `cursor` to the previous duplicate value of current key pair of its database. Return the value and `t`. Return `nil` if there is no prev value. If `cursor` is uninitialized, then `cursor-last` is called on it first.

Wraps mdb_cursor_get() with MDB_PREV_DUP.

- **[function]** **cursor-set-key** *key &optional cursor*

  Move `cursor` to `key` of its database. Return the corresponding value and `t`. Return `nil` if `key` was not found. If `dupsort`, position `cursor` on the first value of `key`.

  Wraps mdb_cursor_get() with MDB_SET_KEY.

- **[function]** **cursor-set-key-dup** *key value &optional cursor*

  Move `cursor` to the `key`, `value` pair of its database and return `t` on success. Return `nil` if the pair was not found.

  Wraps mdb_cursor_get() with MDB_GET_BOTH.

- **[function]** **cursor-set-range** *key &optional cursor*

  Position `cursor` on the first key equal to or greater than `key`. Return the found key, the value and `t`. Return `nil` if `key` was not found. If `dupsort`, position `cursor` on the first value of the found key.

  Wraps mdb_cursor_get() with MDB_SET_RANGE.

- **[function]** **cursor-set-range-dup** *key value &optional cursor*

  Position `cursor` exactly at `key` on the first value greater than or equal to `value`. Return the value at the position and `t` on success, or `nil` if there is no such value associated with `key`.

  Wraps mdb_cursor_get() with MDB_GET_BOTH_RANGE.

10.2 Basic cursor operations

The following operations are similar to `g3t`, `put`, `del` (the Basic operations), but `g3t` has three variants (`cursor-key-value`, `cursor-key`, and `cursor-value`). All of them require the cursor to be positioned (see Positioning cursors).

- **[function]** **cursor-key-value** *&optional cursor*

  Return the key and value `cursor` is positioned at and `t`. Return `nil` if `cursor` is uninitialized.

  Wraps mdb_cursor_get() with MDB_GET_CURRENT.

- **[function]** **cursor-key** *&optional cursor*

  Return the key `cursor` is positioned at and `t`. Return `nil` if `cursor` is uninitialized.

  Wraps mdb_cursor_get() with MDB_GET_CURRENT.

- **[function]** **cursor-value** *&optional cursor*

  Return the value `cursor` is positioned at and `t`. Return `nil` if `cursor` is uninitialized.

  Wraps mdb_cursor_get() with MDB_GET_CURRENT.

- **[function]** **cursor-put** *key value cursor &key current (overwrite t) (dupdata t) append append-dup*

  Like `put`, store key-value pairs into `cursor`'s database. `cursor` is positioned at the new item, or on failure usually near it. Return `value`.

  - `current`: Replace the item at the current cursor position. `key` must still be provided, and must match it. If using sorted duplicates (`dupsort`), `value` must still sort into the same place. This is intended to be used when the new data is the same size as the old. Otherwise it will simply perform a delete of the old record followed by an insert.
  - `overwrite`: If `nil`, signal `lmdb-key-exists-error` if `key` already appears in `cursor-db`.
  - `dupdata`: If `nil`, signal `lmdb-key-exists-error` if the `key`, `value` pair already appears in `db`. Has no effect if `cursor-db` doesn't have `dupsort`.
  - `append`: Append the `key`, `value` pair to the end of `cursor-db` instead of finding `key`'s location in the B+ tree by performing comparisons. The client effectively promises that keys are inserted in sort order, which allows for fast bulk loading. If the promise is broken, `lmdb-key-exists-error` is signalled.
  - `append-dup`: The client promises that duplicate values are inserted in sort order. If the promise is broken, `lmdb-key-exists-error` is signalled.

  May signal `lmdb-map-full-error`, `lmdb-txn-full-error`, `lmdb-txn-read-only-error`.

  Wraps mdb_cursor_put().

- **[function]** **cursor-del** *cursor &key delete-dups*

  Delete the key-value pair `cursor` is positioned at. This does not make the cursor uninitialized, so operations such as `cursor-next` can still be used on it. Both `cursor-next` and `cursor-key-value` will return the same record after this operation. If `cursor` is not initialized, `lmdb-cursor-uninitialized-error` is signalled. Returns no values.

  If `delete-dups`, delete all duplicate values that belong to the current key. With `delete-dups`, `cursor-db` must have `dupsort`, else `lmdb-incompatible-error` is signalled.

  May signal `lmdb-cursor-uninitialized-error`, `lmdb-txn-read-only-error`.

  Wraps mdb_cursor_del().

10.3 Miscellaneous cursor operations

- **[function]** **cursor-renew** *&optional cursor*

  Associate `cursor` with the active transaction (which must be read-only) as if it had been created with that transaction to begin with to avoid allocation overhead. `cursor-db` stays the same. This may be done whether the previous transaction is open or closed (see `open-txn-p`). No values are returned.

  Wraps mdb_cursor_renew().

- **[function]** **cursor-count** *&optional cursor*

  Return the number of duplicate values for the current key of `cursor`. If `cursor-db` doesn't have `dupsort`, `lmdb-incompatible-error` is signalled. If `cursor` is not initialized, `lmdb-cursor-uninitialized-error` is signalled.

  Wraps mdb_cursor_count().

- **[macro]** **do-cursor** *(key-var value-var cursor &key from-end nodup) &body body*

  Iterate over key-value pairs starting from the position of `cursor`. If `cursor` is not positioned then no key-value pairs will be seen. If `from-end`, then iterate with `cursor-prev` instead of `cursor-next`. If `nodup`, then make that `cursor-prev-nodup` and `cursor-next-nodup`.

  If `cursor` is `nil`, the default cursor is used.

  If `nodup` and not `from-end`, then the first duplicate of each key will be seen. If `nodup` and `from-end`, then the last duplicate of each key will be seen.

  To iterate over all key-value pairs with keys >= 7:

```
(with-cursor (cursor db)
  (cursor-set-key 7 cursor)
  (do-cursor (key value cursor)
    (print (cons key value))))
```

- **[macro]** **do-cursor-dup** *(value-var cursor &key from-end) &body body*

  Iterate over duplicate values with starting from the position of `cursor`. If `cursor` is not positioned then no values will be seen. If `from-end`, then iterate with `cursor-prev-dup` instead of `cursor-next-dup`.

  If `cursor` is `nil`, the default cursor is used.

  To iterate over all values that not smaller than #(3 4 5), associated with the key 7:

```
(with-cursor (cursor db)
  (cursor-set-key-dup cursor 7 #(3 4 5))
  (do-cursor-dup (value cursor)
    (print value)))
```

- **[macro]** **do-db** *(key-var value-var db &key from-end nodup) &body body*

  Iterate over all keys and values in `db`. If `nodup`, then all but the first (or last if `from-end`) value for each key are skipped. If `from-end`, then iterate in reverse order.

  To iterate over all values in `db`:

```
(do-db (key value db)
  (print (cons key value)))
```

  This macro establishes a default cursor.

- **[macro]** **do-db-dup** *(value-var db key &key from-end) &body body*

Iterate over all values associated with `key` in `db`. If `from-end`, then iteration starts at the largest value.

To iterate over all values associated with the key 7:

```
(do-db-dup (value db 7)
  (print value))
```

This macro establishes a default cursor.

- **[function]** **list-dups** *db key &key from-end*

  A thin wrapper around `do-db-dup`, this function returns all values associated with `key` in `db` as a list. If `from-end`, then the first element of the list is the largest value.

11 Conditions

- **[condition]** **lmdb-serious-condition** *serious-condition*

  The base class of all `lmdb` conditions. Conditions that are `lmdb-serious-conditions`, but not `lmdb-errors` are corruption and internal errors, which are hard to recover from.

- **[condition]** **lmdb-error** *lmdb-serious-condition error*

  Base class for normal, recoverable `lmdb` errors.

11.1 Conditions for C lmdb error codes

The following conditions correspond to C lmdb error codes.

- **[condition]** **lmdb-key-exists-error** *lmdb-error*

  Key-value pair already exists. Signalled by `put` and `cursor-put`.

- **[condition]** **lmdb-not-found-error** *lmdb-error*

  Key-value pair does not exist. All functions (`g3t`, `cursor-next`, ...) should return `nil` instead of signalling this error. If it is signalled, that's a bug.

- **[condition]** **lmdb-page-not-found-error** *lmdb-serious-condition*

  Requested page not found - this usually indicates corruption.

- **[condition]** **lmdb-corrupted-error** *lmdb-serious-condition*

  Located page was wrong type.

- **[condition]** **lmdb-panic-error** *lmdb-serious-condition*

  Update of meta page failed or environment had fatal error.

- **[condition]** **lmdb-version-mismatch-error** *lmdb-error*

  Environment version mismatch.

- **[condition]** **lmdb-invalid-error** *lmdb-serious-condition*

  File is not a valid `lmdb` file.

- **[condition]** **lmdb-map-full-error** *lmdb-error*

  `env-map-size` reached. Reopen the environment with a larger `:map-size`.

- **[condition]** **lmdb-dbs-full-error** *lmdb-error*

  `env-max-dbs` reached. Reopen the environment with a higher `:max-dbs`.

- **[condition]** **lmdb-readers-full-error** *lmdb-error*

  `env-max-readers` reached. Reopen the environment with a higher `:max-readers`.

- **[condition]** **lmdb-txn-full-error** *lmdb-error*

  `txn` has too many dirty pages. This condition is expected to occur only when using nested read-write transactions or operations multiple items (currently not supported by this wrapper).

- **[condition]** **lmdb-cursor-full-error** *lmdb-serious-condition*

  Cursor stack too deep - internal error.

- **[condition]** **lmdb-page-full-error** *lmdb-serious-condition*

  Page has not enough space - internal error.

- **[condition]** **lmdb-map-resized-error** *lmdb-error*

  Data file contents grew beyond `env-map-size`. This can happen if another OS process using the same environment path set a larger map size than this process did.

- **[condition]** **lmdb-incompatible-error** *lmdb-error*

  Operation and `db` incompatible, or `db` type changed. This can mean:

  - The operation expects a `dupsort` or `dupfixed` database.
  - Opening a named `db` when the unnamed `db` has `dupsort` or `integer-key`.
  - Accessing a data record as a database, or vice versa.
  - The database was dropped and recreated with different flags.

- **[condition]** **lmdb-bad-rslot-error** *lmdb-error*

  Invalid reuse of reader locktable slot. May be signalled by `with-txn`.

- **[condition]** **lmdb-bad-txn-error** *lmdb-error*

  Transaction must abort, has a child, or is invalid. Signalled, for example, when a read-only transaction is nested in a read-write transaction, or when a cursor is used whose transaction has been closed (committed, aborted, or reset).

- **[condition]** **lmdb-bad-valsize-error** *lmdb-error*

  Unsupported size of key/`db` name/data, or wrong `dupfixed`, `integer-key` or `integer-dup`. See `env-max-key-size`.

- **[condition]** **lmdb-bad-dbi-error** *lmdb-error*

  The specified `dbi` was changed unexpectedly.

11.2 Additional conditions

The following conditions do not have a dedicated C lmdb error code.

- **[condition]** **lmdb-cursor-uninitialized-error** *lmdb-error*

  Cursor was not initialized. Position the cursor at a key-value pair with a function like `cursor-first` or `cursor-set-key`. Signalled when some functions return the C error code `einval`.

- **[condition]** **lmdb-cursor-thread-error** *lmdb-error*

  Cursor was accessed from a thread other than the one in which it was created. Since the foreign cursor object's lifetime is tied to the dynamic extent of its `with-cursor`, this might mean accessing garbage in foreign memory with unpredictable consequences.

- **[condition]** **lmdb-txn-read-only-error** *lmdb-error*

  Attempt was made to write in a read-only transaction. Signalled when some functions return the C error code `eaccess`.

- **[condition]** **lmdb-illegal-access-to-parent-txn-error** *lmdb-error*

  A parent transaction and its cursors may not issue any other operations than `commit-txn` and `abort-txn` while it has active child transactions. In `lmdb`, Basic operations are always executed in the active transaction, but Cursors can refer to the parent transaction:

```
(with-temporary-env (*env*)
  (let ((db (get-db "db")))
    (with-txn (:write t)
      (put db #(1) #(1))
      (with-cursor (cursor db)
        (with-txn (:write t)
          (assert-error lmdb-illegal-access-to-parent-txn-error
            (cursor-set-key #(1) cursor)))))))
```

12 Indices

Referrer definition type abbreviations:

- *f*: for definitions in the function namespace (macros, compiler macros and also methods)
- *t*: `DEFTYPE`s, classes, conditions, structs
- *d*: documentation sections and glossary terms

- *l*: definitions of definition types
- *s*: ASDF systems
- *p*: packages
- *n*: named readtables
- *v*: special variables and constants
- *r*: restarts
- *?*: other

12.1 Function and Macro Index

12.2 Variable and Constant Index

12.3 Type Index

12.4 Misc Index

12.5 Concept Index