

libmdbx Textbook

From the first steps to expert topics and bindings

The libmdbx project

SkyNET@K-PAX 2026-10-19 v0.15.0-150-g6ca39a94

Table of contents

1. libmdbx Textbook	4
1.1 Who this textbook is for	4
1.2 Structure and how to read it	4
1.3 Code examples	4
1.4 Versions and compatibility	5
1.5 About the textbook	5
2. libmdbx Textbook — Volume 0. Framework (table of contents)	6
2.1 The overall logic of progression	6
2.2 Volume I. Fundamentals (for beginners)	6
2.3 Volume II. Practical usage (for developers)	7
2.4 Volume III. Internal mechanisms (for the knowledgeable)	7
2.5 Volume IV. Performance and optimization (for experienced readers)	8
2.6 Volume V. Expert topics and edge cases (for advanced readers)	8
2.7 Volume VI. Bindings	9
3. Volume I. Fundamentals	10
3.1 Chapter 1. What is libmdbx	10
3.2 Chapter 2. Installation and first run	12
3.3 Chapter 3. Basic data model	15
3.4 Chapter 4. Basic CRUD operations	18
3.5 Chapter 5. Transactions — first steps	20
3.6 Volume summary	24
4. Volume II. Practical Usage	25
4.1 Chapter 6. Cursors	25
4.2 Chapter 7. Multivalues and DUPSORT	28
4.3 Chapter 8. Secondary indexes	31
4.4 Chapter 9. Environment configuration	33
4.5 Chapter 10. Durability modes	37
4.6 Chapter 11. Multithreading	39
4.7 Chapter 12. Error handling	41
4.8 Volume summary	43
5. Volume III. Internal Mechanisms	44
5.1 Chapter 13. Storage Architecture: B+tree and mmap	44
5.2 Chapter 14. MVCC and Snapshots	46
5.3 Chapter 15. Copy-on-Write and the Commit Pipeline	48
5.4 Chapter 16. GC — the Garbage Collector Inside the Database	49

5.5	Chapter 17. Growth, Shrinkage, and Defragmentation of the Database	51
5.6	Chapter 18. Nested Transactions	53
5.7	Chapter 19. The Lock File and Interprocess Synchronization	55
5.8	Chapter 20. Durability and Recovery	57
5.9	Volume Conclusion	58
6.	Volume IV. Performance and Optimization	59
6.1	Chapter 21. WAF — write amplification factor	59
6.2	Chapter 22. Choosing a configuration for the scenario	60
6.3	Chapter 23. Micro-optimizations	62
6.4	Chapter 24. Cache lookup (get-cached)	64
6.5	Chapter 25. Bulk operations	65
6.6	Chapter 26. Benchmarks and measurements	67
6.7	Volume summary	69
7.	Volume V. Expert topics and edge cases	70
7.1	Chapter 27. Rules and tips from the knowledge base	70
7.2	Chapter 28. Platform-specific notes	72
7.3	Chapter 29. Handle-Slow-Readers (HSR)	73
7.4	Chapter 30. Diagnostics and debugging	75
7.5	Chapter 31. Migrating from LMDB	77
7.6	Chapter 32. Design patterns on libmdbx	79
7.7	Chapter 33. Roadmap and the future: MithrilDB	82
7.8	Volume summary	83
8.	Volume VI. Bindings	84
8.1	Chapter 34. Overview of the bindings ecosystem	84
8.2	Chapter 35. Rust (mdbx-sys / mdbx-rs)	85
8.3	Chapter 36. Go (mdbx-go)	86
8.4	Chapter 37. Python (python-lmdbx / mdbx-py)	86
8.5	Chapter 38. Node.js (node-mdbx)	87
8.6	Chapter 39. .NET / C# (libmdbx-dotnet)	87
8.7	Chapter 40. C++ (mdbx.h++)	88
8.8	Chapter 41. Dart (mdbx-dart / Isar)	89
8.9	Chapter 42. Nim	90
8.10	Chapter 43. Java	90
8.11	Chapter 44. Haskell	90
8.12	Chapter 45. Ruby	91
8.13	Chapter 46. Scala	91
8.14	Volume summary	91
8.15	Textbook conclusion (volumes I–VI)	92

libmdbx v0.15.0-150-g6ca39a94 — built 2026-10-19 · [online version](#) · [PDF version](#)

1. libmdbx Textbook

A complete practical guide to the embedded transactional database **libmdbx** — from the first steps to expert topics and bindings. All code examples are compilable C++17/C11 programs in the [examples/](#) directory; fragments in the text are marked as "Fragment" and supplemented with links to the full code. A [PDF edition of the textbook](#) is available for offline reading.

1.1 Who this textbook is for

The textbook is intended for developers working with C/C++ (and, in Volume VI, with other languages through bindings). The minimum level is confident command of C and a basic understanding of what databases are. Experienced engineers may start with the second or third volume: internal mechanisms, performance, and expert topics are self-contained provided the reader is familiar with the API.

1.2 Structure and how to read it

The textbook consists of six volumes linked by a single logic of progression:

Volume	Topic	Level
Volume I. Fundamentals	What libmdbx is, installation, data model, CRUD, transactions	beginner
Volume II. Practice	Cursors, DUPSORT, indexes, configuration, durability, multithreading, errors	developer
Volume III. Mechanisms	B+tree, MVCC, CoW, GC, geometry, locking, recovery	knowledgeable
Volume IV. Performance	WAF, configurations, micro-optimizations, get-cached, bulk operations, benchmarks	experienced
Volume V. Expert topics	Rules, platforms, HSR, diagnostics, migration, patterns, roadmap	advanced
Volume VI. Bindings	Rust, Go, Python, Node.js, .NET, C++, and others	integrator

- **Beginners:** we recommend reading Volumes I–II in sequence; there you will also follow the end-to-end configurator project, whose exercises reinforce the material.
- **Experienced:** Volumes III–V as needed; each volume is self-contained.
- **Integrators:** Volume VI: binding cards with the specifics of each language.

Each chapter follows the pattern "concept → mechanism → practice → nuance", and ends with a summary, a checklist, and (in Volumes I–II) exercises.

1.3 Code examples

All examples live in [examples/](#) and are built via the project's CMake (or pkg-config):

```
cmake -S . -B build -DMDBX_ENABLE_TESTS=ON
cmake --build build
ctest --test-dir build      # also runs the examples
```

Links of the form [examples/c++/01-hello.c++](#) in the text point to a full compilable example (the project's [examples/](#) directory, also reachable from this directory via a symbolic link); short illustrations are marked "Fragment" and are not required to compile on their own.

1.4 Versions and compatibility

The material corresponds to the current `master` branch (at the time of publication — the v0.15.x series). The APIs shown are verified against `mdbx.h` / `mdbx.h++` ; some fine-grained parameters (option defaults, signatures of service functions) should be checked against the official documentation and the header files of your version.

1.5 About the textbook

The textbook was created by the AI agent **skynet** based on the libmdbx source code (the `master` branch), the official header files `mdbx.h` / `mdbx.h++` , and the project documentation. The material has been cross-checked against the code and factually verified. The goal is to provide a structured, accurate, and practical guide to libmdbx.

Feedback, remarks, and suggestions — in the project's issues or in the [Telegram group](#).

libmdbx v0.15.0-150-g6ca39a94 — built 2026-10-19 · [online version](#) · [PDF version](#)

2. libmdbx Textbook — Volume 0. Framework (table of contents)

This is the table of contents and structure of the six-volume textbook: from basic concepts to bindings. Each volume is a separate file in this directory; all volumes are linked by cross-references and a single end-to-end project.

2.1 The overall logic of progression

Basic concepts (Volume I) → Practical usage (Volume II) → Internal mechanisms (Volume III)
→ Performance and optimization (Volume IV) → Expert topics and edge cases (Volume V) → Bindings (Volume VI)

Each chapter follows the pattern "concept → mechanism → practice → nuance". A term is defined at its first use. The end-to-end project (an application configurator) runs from Volume I through Volume II; the exercises are the steps of this project.

2.2 Volume I. Fundamentals (for beginners)

Goal: a reader with no experience with embedded key-value databases understands libmdbx and can build it and run the first operations.
Length ~50–100 pages.

- **Ch. 1. What is libmdbx**

- Embedded vs client-server databases; the key-value model; why mmap; place in the landscape (LMDB/BerkeleyDB/LevelDB/RocksDB/SQLite at the level of architectural properties); license, ecosystem, known users (Erigon, Reth, Isar, Monica Pass).

- **Ch. 2. Installation and first run**

- Building (amalgamation vs full source), linking (CMake/pkg-config/static/dynamic), key build options; minimal open→put→get example; running the tests.

- **Ch. 3. Basic data model**

- MDBX_val; the main table and dbi; MDBX_CREATE flags; constraints (key/value/number of tables); integer keys; zero keys/values (difference from LMDB).

- **Ch. 4. Basic CRUD operations**

- put/get/del/replace; flags (NOOVERWRITE/NODUPDATA/CURRENT/ALLDUPS); reading with get vs cursor; full C examples; handling return codes.

- **Ch. 5. Transactions — first steps**

- Why transactions; begin/commit/abort; read-only vs read-write; MVCC intuitively; the "one transaction — one thread" rule; examples.

2.3 Volume II. Practical usage (for developers)

Goal: the reader builds applications: cursors, multivalues, indexes, configuration, durability, multithreading, error handling. Length ~100–200 pages.

- **Ch. 6. Cursors** — positioning (FIRST/LAST/NEXT/prev/SET/SET_RANGE/LOWER/UPPER_BOUND), batch operations, UB after deletion, clone/bind, the safe-delete pattern in DUPSORT, examples.
- **Ch. 7. Multivalues and DUPSORT** — DUPFIXED/INTEGERDUP/REVERSEDUP, the nested value subtree, inverted "key→list of IDs" indexes, GET_BOTH/GET_BOTH_RANGE, secondary index.
- **Ch. 8. Secondary indexes** — main table + index tables; DUPSORT index vs separate table; maintaining consistency; composite keys; deletion by index; mini-ORM.
- **Ch. 9. Environment configuration** — geometry (lower/now/upper/growth_step/shrink_threshold), maxreaders/maxdbs/pagesize, flags, runtime options MDBX_opt_*, choosing a 4/8/64 KB page, env_info/env_stat.
- **Ch. 10. Durability modes** — SYNC_DURABLE/NOMETASYNC/SAFE_NOSYNC/UTTERLY_NOSYNC/WRITEMAP; when to use which; synchbytes/syncperiod; mdbx_env_sync_ex; platform nuances (F_FULLFSYNC, boot_id, LockFileEx).
- **Ch. 11. Multithreading** — sticky threads; NOSTICKYTHREADS (pools/coroutines); thread registration; TLS destructors; fork+resurrect; txn_clone; parking/eviction; multiple environments; MDBX_EXCLUSIVE.
- **Ch. 12. Error handling** — positive/negative codes; RESULT_TRUE/FALSE; strerror_r vs strerror; MAP_FULL/TXN_FULL; expected codes (KEYEXIST/NOTFOUND); BAD_TXN/BAD_DBI/EBADSIGN; WANNA_RECOVERY/MVCC_RETARDED; THREAD_MISMATCH/TXN_OVERLAPPING/BUSY; retry strategies.

2.4 Volume III. Internal mechanisms (for the knowledgeable)

Goal: the reader can explain the database's behavior through its architecture — a bridge between "how to use" and "how to configure/debug". Length ~100–200 pages.

- **Ch. 13. Storage architecture: B+tree and mmap** — branches/leaves/overflow; why B+tree; page; mmap without the buffer cache; CoW pages; page states; the reachability invariant.
- **Ch. 14. MVCC and snapshots** — page versioning; the meta-page troika (Troika), two-phase commit, the finite state machine of 216 states; RLT; wait-free reading; safe64; the writer (mutex, front txnid, DPL); comparison with PostgreSQL/Oracle.
- **Ch. 15. Copy-on-Write and the commit pipeline** — touch→copy→DPL→spill→commit; DPL deduplication; spill (LRU, overflow priority, cursor protection, denominators); loose; refund; pipeline stages (GC→refund→spill→audit→meta→sync→unlock) and their impact on latency.
- **Ch. 16. GC** — why not a free-list; record format (key=txnid, value=PNL); the detent and its computation/caching; FIFO vs LIFO; BigFoot; recursive update; rp_augment_limit/ gc_time_limit; early GC cleanup 2025; non-deferred cleanup (devel 0.14.x).
- **Ch. 17. Growth, shrinking, and defragmentation** — why it grows (readers/SAFE_NOSYNC/fragmentation); geometry; MAP_FULL and mitigation measures (steady-point/HSR/eviction); auto-compaction via refund and its limits; mdbx_env_defrag; hysteresis; capacity estimation.
- **Ch. 18. Nested transactions** — purpose; the CoW shadow of the parent, paired cursors; join vs undo; the fate of tables; the anti-pattern "a nested transaction in every function"; workaround for lack of space; compatibility with WRITEMAP.
- **Ch. 19. The lock file and interprocess synchronization** — why LCK; implementations (POSIX-2008/2001, SysV, WIN32FILES), why LockFileEx rather than named mutexes; LCK versioning; recovery after a crash; reader_check; the no-LCK mode.
- **Ch. 20. Durability and recovery** — two-phase meta (weak/steady); recovery without WAL; open for recovery (monotonic numbers, exclusive); boot_id and LXC; page cache incoherence (#269, FORCE_CHECK_MMAP_COHERENCY); mdbx_chk; what survives a power failure per mode.

2.5 Volume IV. Performance and optimization (for experienced readers)

Goal: the reader tunes libmdbx to their scenario, finds and eliminates bottlenecks. Length ~80–150 pages.

- **Ch. 21. WAF** — what and why; sources of amplification; batching as the main lever; tree height; the spill trade-off; `prefer_waf_insteadof_balance/merge_threshold`; calculating WAF with numbers.
- **Ch. 22. Choosing a configuration for your scenario** — 6 scenarios (write-heavy, read-heavy, mobile, analytics, multi-process reading, low latency): `modes/flags/geometry/page/batching/numbers`.
- **Ch. 23. Micro-optimizations** — SIMD search, branchless bsearch, radix-sort, atomics for weak memory models, minimizing syscalls, `prefault+mincore`, auto-appending split, merge with a dirty neighbor; when they matter.
- **Ch. 24. Cache lookup (get-cached)** — the concept, statuses HIT/CONFIRMED/REFRESHED/DIRTY/BEHIND/UNABLE/RACE; when it speeds things up; ABA; the multithreaded variant + NOSTICKYTHREADS; SingleThreaded vs multithreaded; history (Gabriel RABHI, Ghost Body Object).
- **Ch. 25. Bulk operations** — `bunch_delete`, `get_batch`, `GET/PUT_MULTIPLE`, `estimate_range/distance/move`, `txn_clone`, batching 100–10000 operations; when the "maximum transaction" is an anti-pattern.
- **Ch. 26. Benchmarks and measurements** — `ioarena`; what to measure; `commit_latency` by stages; PROFGC; typical benchmarking mistakes; "6–7× slower than LMDB" — what it means; reference points (200 TPS, 1–3M get/s, 20K–10M inserts/s); read scaling.

2.6 Volume V. Expert topics and edge cases (for advanced readers)

Goal: the reader knows the subtleties and pitfalls, can debug and make architectural decisions. Length ~100–200 pages.

- **Ch. 27. Rules and advice** — proven practices and pitfalls, criticality context, counterexamples.
- **Ch. 28. Platform nuances** — Linux (`boot_id`, `/proc`, `/dev/shm`, `tmpfs/ENOSPC`, page cache, `mincore`); Windows (`LockFileEx`, `LARGEADDRESSAWARE`, 2G/4G, WSL1/2); macOS/iOS (`F_FULLFSYNC`, `APPLE_SPEED_INSTEADOF_DURABILITY`, `F_REALLOCATE`); Android/bionic; containers (LXC/Docker); Wine; Linux <4.x; `tmpfs`.
- **Ch. 29. Handle-Slow-Readers (HSR)** — why; `mdbx_env_set_hsr`; parameters and returns (`wait/kill/growth/MAP_FULL`); typical implementations; HSR + `SAFE_NOSYNC`; a C example.
- **Ch. 30. Diagnostics and debugging** — tools (`mdbx_chk` options and exit codes, PROFGC and the semantics of `gc_prof`, `commit_latency` by stages, `txn_info`, `reader_check/list`, logging, `ASAN/UBSAN/valgrind`); step-by-step scenarios (database growth, commit slowdown, `MAP_FULL`, deadlock/BUSY, a long-running reader, corruption, slot leak, performance, fragmentation, fork, Windows slowdowns, containers); checklists before production.
- **Ch. 31. Migration from LMDB** — advantages; format compatibility (0.11.x ↔ 0.12.x); breaking changes (`NOLOCK` removed, `NOTLS` → `NOSTICKYTHREADS`); the `mdb_` → `mdbx_` mapping table; behavioral differences (meta troika, two-phase commit, checksums); migration checklist.
- **Ch. 32. Design patterns** — 8 patterns (auto-ID via sequence, DUPSORT secondary index, composite key+comparator, task queue table-as-queue, ring buffer, full-scan iterator, replication via per-table `txnid`, read-your-writes via `clone/embark_read`): problem → solution with code → trade-offs → alternatives.
- **Ch. 33. Roadmap and the future: MithrilDB** — the current state; a common API for several formats; amalgamation; replication (nonlinear GC); change subscription; encryption/compression; a streaming mode for long values ("Streaming BLOBs"); SWIG.

2.7 Volume VI. Bindings

Goal: the reader uses libmdbx from their language with an understanding of the binding's specifics and limitations. Length ~50–150 pages.

- **Ch. 34. Ecosystem overview** — the full list of officially tracked bindings; statuses (mature/experimental/deprecated); the "language → binding" map.
- **Ch. 35–46. Binding cards** (following a single template: overview/installation/basic example/ threading/memory/nuances/typical errors/performance/gaps/references): 35 Rust (mdbx-sys/mdbx-rs), 36 Go (mdbx-go), 37 Python (python-lmdbx/mdbx-py), 38 Node.js (node-mdbx), 39 .NET/C# (libmdbx-dotnet), 40 C++ (mdbx.h++), 41 Dart (mdbx-dart), 42 Nim, 43 Java, 44 Haskell, 45 Ruby, 46 Scala. Zig (officially tracked) — a short reference in Ch. 34 (§34.2), without a separate card.
- **Mandatory nuances:** Go+NOSTICKYTHREADS (deadlock scenarios, CGO); Rust+async (Send/Sync, use-after-free after commit, blocking calls); .NET+GC (SafeHandle, pinning, lifetime); Python+GIL (long reads, fork after import).
- **Honesty:** when data is insufficient — "requires analysis of the binding's repository".

libmdbx v0.15.0-150-g6ca39a94 — built 2026-10-19 · [online version](#) · [PDF version](#)

3. Volume I. Fundamentals

Level: for beginners — those who have never worked with embedded key-value databases. **Volume goal:** you understand what libmdbx is, can build it, open a database, and perform your first read and write operations in transactions. **End-to-end project:** across Volumes I–II we build a single application — an **application configurator** (a simple key-value parameter store that gradually gains indexes, multithreading, and optimizations).

The volume's progression: concept → practice → mechanism → nuance. The example code is complete and compilable; the helper functions (`env_open` in C++ / `ex_env_open` in C, plus `check_rc` and `die`) are shared across the whole textbook (see [examples/common/](#)).

3.1 Chapter 1. What is libmdbx

3.1.1 1.1. Embedded database

Databases come in two fundamentally different kinds: **client-server** and **embedded**.

A client-server database (PostgreSQL, MySQL) is a separate server process. Your application talks to it over the network or a socket: it sends SQL and receives answers. The server owns the data files and manages the cache, concurrent access, and the journal. You pay for this with deployment complexity, a separate process, and network overhead.

An embedded database (SQLite, libmdbx) is a **library** that you link directly into your process. There is no server and no network: your application calls functions that read and write the file directly. The advantages are simplicity (one library), speed (no network layer), and easy distribution (the application carries its own database). The price — you are responsible for concurrent access and backups yourself.

libmdbx is an embedded transactional **key-value** database, a deeply reworked descendant of the well-known LMDB library. "Transactional" means that a group of operations is executed atomically: either all changes are applied, or none are.

Analogy. SQLite is an "embedded PostgreSQL". libmdbx is an "embedded high-performance cache server": it does not know SQL or tables in the relational sense, but it can deliver data at memory-access speed.

3.1.2 1.2. Key-value data model

A key-value database stores simple pairs: **key** → **value**. Both parts are just byte sequences whose meaning only your application knows.

```
"user:1001" → {"name": "Anna", "role": "admin"}
"user:1002" → {"name": "Boris", "role": "user"}
"theme"    → "dark"
```

How libmdbx differs from an "ordinary" dictionary (`std::map`, `HashMap`):

- **Keys are always ordered.** Data is stored in a balanced tree (B+tree), so keys can be traversed in order, ranges searched, and the "nearest greater key" found. This is like an array with sorted keys rather than a hash table.
- **Values can be large** — up to ~2 GB (they go onto separate "overflow" pages).
- **Data lives in a memory-mapped file.** Reading a key is effectively dereferencing a pointer, without copying and without serialization.

3.1.3 1.3. Why mmap: memory as the data interface

Most databases maintain their own buffer cache: when reading they copy a page from disk into the library's memory; when writing they accumulate changes in this cache and periodically flush them to disk.

libmdbx takes a different approach: a **memory-mapped file** (mmap). The operating system maps the database file directly into the process's address space. The data looks like an ordinary byte array. Reading is an access by address; the kernel pulls in the page as needed (page cache). The library has no separate buffer cache — the OS page cache plays that role.

Three important properties follow from this:

1. **Reading does not copy data.** You get a pointer straight into the mmap region — no deserialization. This is the source of the record-breaking read speed (typically millions of operations per second).
2. **Writes go through Copy-on-Write (CoW).** When changing a page, the library does not touch the old version (other transactions may be reading it) but creates a new copy. Details are in Volume III.
3. **The operating system itself decides which pages to keep in memory.** The library does not choose an eviction strategy — the kernel does.

3.1.4 1.4. libmdbx's place in the landscape

Database	Model	Architectural properties
LMDB	KV, mmap, MVCC	Progenitor of the approach; simpler, fewer features
libmdbx	KV, mmap, MVCC	Reworked LMDB: more durability modes, GC instead of a free-list, stronger guarantees
BerkeleyDB	KV, classic	Older generation; page-level locking, no mmap
LevelDB/RocksDB	KV, LSM	Optimized for writes, background compaction, poor read predictability
SQLite	Relational, embedded	SQL, transactions; slower on KV workloads

Here **MVCC** (multi-version concurrency control) is the technique that lets readers and writers proceed simultaneously without blocking readers: every transaction sees an immutable snapshot of the data (details in chapter 5 and Volume III). **LSM** (log-structured merge-tree) is the alternative write-optimized architecture that trades away read predictability.

Comparison at the level of properties (not benchmarks): if you need an ordered key-value store with transactions, lightning-fast reads, and predictable writes — choose libmdbx. If you need SQL — SQLite. If writes dominate reads and you can tolerate background compaction — RocksDB.

3.1.5 1.5. License, ecosystem, who uses it

libmdbx is distributed under a permissive license: historically — the OpenLDAP Public License (OLPL); on master — **entirely Apache-2.0** (re-licensed in 0.13.x). It permits commercial use without opening your own code.

The library is used in high-load projects:

- **The Ethereum ecosystem:** Erigon, Akula, Silkworm (all fast Ethereum implementations run on libmdbx);
- **Isar** — the local database for Flutter applications;
- messengers, analytics, cache layers.

The binding ecosystem covers Rust, Go, Python, Node.js, .NET, C++, Zig, Dart, Nim, Java, Haskell, Ruby, Scala (in detail — Volume VI).

3.1.6 1.6. Summary of chapter 1

- libmdbx is an embedded transactional key-value database (a library, not a server).
- Keys are ordered (B+tree); values are arbitrary bytes up to ~2 GB.
- Data is memory-mapped (mmap): copy-free reads, Copy-on-Write writes.
- Properties: wait-free readers, a single writer, transactions, no WAL (recovery via meta pages).
- Apache-2.0 (master)/OLPL (historically); used in Ethereum clients, Isar, and other projects.

3.1.7 1.7. Exercises

1. How does a client-server database differ from an embedded one? Give an example where the embedded one is preferable.
2. What does key ordering provide beyond an ordinary dictionary?
3. Why can reading through mmap be faster than through a database's own cache?

3.1.8 1.8. Chapter 1 checklist

- [] I can explain the difference between client-server and embedded databases and say when each is preferable;
- [] I understand that libmdbx is an ordered key-value engine (B+tree), not a hash table and not SQL;
- [] I can describe the role of mmap: copy-free reads, Copy-on-Write writes, page eviction handled by the kernel;
- [] I know the key properties: MVCC, wait-free readers, a single writer, no WAL;
- [] I remember the license (Apache-2.0 on master) and examples of libmdbx users (Erigon, Isar).

3.1.9 1.9. What's next

It is time to turn this understanding of "what libmdbx is" into working code. In chapter 2 we will build the library two ways (amalgamation and CMake), link it into a project, and run the first full "hello, libmdbx" example. For the end-to-end configurator project this step is mandatory: without a built library and a verified open → put → get cycle there is no moving forward.

3.2 Chapter 2. Installation and first run

3.2.1 2.1. Building from source

libmdbx can be built in two ways:

- **Amalgamated** (amalgamation): a single `mdbx.c` file plus the `mdbx.h` header, which you can simply drop into your project. Minimal dependencies, ideal for embedding.
- **Full source**: a directory with `CMakeLists.txt`, tests, examples, and utilities (`mdbx_stat`, `mdbx_chk`, `mdbx_copy`, `mdbx_dump`, `mdbx_load`, `mdbx_defrag`).

Amalgamation: download `mdbx.h` and `mdbx.c` and add `mdbx.c` to your project's build. Nothing more is required — this is a true "single-file" engine.

3.2.2 2.2. Adding to a project

CMake (full source):

```
add_subdirectory(Libmdbx)
target_link_libraries(yourapp PRIVATE mdbx)
```

or **pkg-config**:

```
pkg-config --cflags --libs libmdbx
```

Building it yourself:

```
make -j
make install # installs libmdbx.a / libmdbx.so, mdbx.h, .pc
```

Static vs dynamic. Static linking is easier to distribute (no .so versioning problems) and gives more optimization opportunities. Dynamic linking lets you update the engine without rebuilding the application.

3.2.3 2.3. Key build options

Option	Meaning
<code>MDBX_LOCKING</code>	Locking implementation: <code>POSIX2008</code> (Linux, default), <code>POSIX2001</code> , <code>SYSV</code> (macOS default), <code>WIN32FILES</code> (Windows)
<code>MDBX_CHECKING</code>	Level of internal checks/assertions (−1...3). 2 / 3 are very slow — for debugging only
<code>MDBX_FORCE_ASSERTIONS</code>	Deprecated option (equivalent to <code>MDBX_CHECKING=2</code>); prefer <code>MDBX_CHECKING</code>
<code>MDBX_ENABLE_BIGFOOT</code>	Chains of GC records for very large frees; enabled by default on 64-bit builds
<code>MDBX_DEBUG</code>	Logging/assertions/audit (0 by default)
<code>MDBX_VALIDATION</code>	Additional structural checks on every operation
<code>MDBX_ENABLE_PGOP_STAT</code>	Page operation counters (for diagnostics)

Tip: for production, build with `MDBX_CHECKING=0` (or omit it entirely) and `NDEBUG`. A debug build with assertions can be several times slower and mask real timings.

3.2.4 2.4. Minimal example

The first full listing of the textbook. It opens a database, writes a key, and reads it back.

```
#include <stdio.h>
#include <string.h>
#include <mdbx.h>

static void die(const char *what, int rc) {
    fprintf(stderr, "%s: %s (%d)\n", what, mdbx_strerror(rc), rc);
    exit(1);
}

int main(void) {
    MDBX_env *env = NULL;
    MDBX_txn *txn = NULL;
    MDBX_dbi dbi;
    MDBX_val key, data;
    int rc;

    /* 1. Create the environment */
    rc = mdbx_env_create(&env);
    if (rc != MDBX_SUCCESS) die("env_create", rc);

    /* 2. Open (or create) the database file; default geometry */
    rc = mdbx_env_open(env, "./demo.mdbx",
                       MDBX_NOSUBDIR, 0664);
    if (rc != MDBX_SUCCESS) die("env_open", rc);

    /* 3. Read-write transaction */
    rc = mdbx_txn_begin(env, NULL, MDBX_TXN_READWRITE, &txn);
    if (rc != MDBX_SUCCESS) die("txn_begin", rc);

    /* 4. Open the main table (NULL name = main) */
    rc = mdbx_dbi_open(txn, NULL, 0, &dbi);
    if (rc != MDBX_SUCCESS) die("dbi_open", rc);

    /* 5. Write "greeting" -> "hello, libmdbx" */
    const char *k = "greeting";
    const char *v = "hello, libmdbx";
    key.iov_len = strlen(k);
    key.iov_base = (void *)k;
    data.iov_len = strlen(v);
    data.iov_base = (void *)v;
    rc = mdbx_put(txn, dbi, &key, &data, 0);
    if (rc != MDBX_SUCCESS) die("put", rc);

    /* 6. Commit */
    rc = mdbx_txn_commit(txn);
    if (rc != MDBX_SUCCESS) die("txn_commit", rc);

    /* 7. Read in a new read-only transaction */
    rc = mdbx_txn_begin(env, NULL, MDBX_TXN_RDONLY, &txn);
    if (rc != MDBX_SUCCESS) die("txn_begin-ro", rc);
    rc = mdbx_get(txn, dbi, &key, &data);
    if (rc != MDBX_SUCCESS) die("get", rc);
    printf("Value: %s\n", (int)data.iov_len, (char *)data.iov_base);
    mdbx_txn_abort(txn);

    mdbx_env_close(env);
}
```

```
    return 0;
}
```

Let's walk through it step by step.

- `mdbx_env_create` creates an empty environment object — a "connection" to the future database.
- `mdbx_env_open` opens the database file; creation is controlled by the `mode` parameter: a non-zero `0664` means "create if the database does not exist yet". Do not pass `MDBX_CREATE` here — it is for `mdbx_dbi_open()`, and its bit coincides with `MDBX_NOMETASYNC`. `MDBX_NOSUBDIR` means the file name is given directly rather than as a prefix for the `data+lock` pair.

Important: the `MDBX_CREATE` bit = `0x40000` is the same bit as `MDBX_NOMETASYNC`. If you pass `MDBX_CREATE` in the `flags` of `mdbx_env_open()`, the database will be created (because of the non-zero `mode`), but the application will **silently** get weakened durability (`NOMETASYNC`). Creation is enabled only via `mode != 0`.

- All changes happen **inside a transaction** — before commit they are not visible to other readers.
- `mdbx_dbi_open(txn, NULL, 0, &dbi)` opens the **main table** (in detail — chapter 3).
- `mdbx_put` stores a pair; `mdbx_get` reads it.
- `MDBX_val` is a structure `{ iov_base, iov_len }`: a pointer and the length of the byte window.

Fragment from `examples/c++/01-hello.c++` — the same open → put → get → close cycle on the C++ API: RAII objects (`env_managed`, transactions) close the environment themselves, and keys and values are passed as `mdbx::slice`:

```
#include <iostream>
#include <string>

#include <mdbx.h++>
#include "common/common.h++"

int main() {
    try {
        const std::string path = std::string(example::tmpdir()) + "/mdbx-01-hello.mdbx";
        mdbx::env::remove(path); // erase leftovers from the previous run

        // Creation: the constructor with create_parameters creates the DB if absent.
        mdbx::env_managed env(path, mdbx::env_managed::create_parameters(), mdbx::env::operate_parameters());
        {
            auto txn = env.start_write();
            auto table = txn.open_map(nullptr); // main table
            txn.insert(table, mdbx::slice("key"), mdbx::slice("hello, libmdbx"));
            txn.commit();
        }
        {
            auto rtxn = env.start_read();
            auto table = rtxn.open_map(nullptr);
            std::cout << "got: " << rtxn.get(table, mdbx::slice("key")).as_string() << "\n";
        }
    }
}
```

Full code: [01-hello.c++](#) · [C version](#)

3.2.5 2.5. Building and running the tests

In the full source:

```
cmake -S . -B build -DMDBX_ENABLE_WERROR=ON
cmake --build build -j
ctest --test-dir build
```

The stochastic stress test `mdbx_test` with a fixed `--prng-seed` is useful — it is reproducible and helps debug rare errors (in detail in Volume V, chapter 30).

3.2.6 2.6. Nuance: data and lock files

One "database" physically consists of **two files**: the data (`demo.mdbx`) and the lock file (`demo.mdbx-lck`). The lock file contains the writer mutex and the reader table — interprocess infrastructure. Do not delete `.lck` while the database is open; when copying a database, remember the file pair (in detail — Volume III, chapter 19).

Examples for this chapter: [examples/c++/01-hello.c++](#) · [C version](#); end-to-end project: [config-store-02.c++](#).

3.2.7 2.7. Summary of chapter 2

- Amalgamation (`mdbx.c` + `mdbx.h`) or full source with CMake.
- Key build options: `MDBX_LOCKING` , `MDBX_CHECKING` , `MDBX_ENABLE_BIGFOOT` .
- Minimal cycle: `create` → `open` → `txn_begin` → `dbi_open` → `put/get` → `commit` → `close`.
- The database is a file pair: data + locks.
- Enable debug checks deliberately — they slow things down.

3.2.8 2.8. Exercises

1. Build the minimal example and run it twice — make sure the value survives the restart.
2. Add a second key and read both in a single transaction.
3. What happens if you pass `mode = 0` on the first run (when the database does not exist yet)?

3.2.9 2.9. Chapter 2 checklist

- [] I can build the library two ways: as an amalgamation (`mdbx.c` + `mdbx.h`) and from the full source with CMake;
- [] I know the key build options (`MDBX_LOCKING` , `MDBX_CHECKING` , `MDBX_ENABLE_BIGFOOT`) and why production builds want `MDBX_CHECKING=0` and `NDEBUG` ;
- [] I can build and run the minimal example and make sure the value survives a restart;
- [] I know the minimal cycle: `create` → `open` → `txn_begin` → `dbi_open` → `put/get` → `commit` → `close`;
- [] I remember that a database is a file pair (data + `.lck`) and never delete the lock file while the database is open.

3.2.10 2.10. What's next

The next chapter is about what you actually write and read: the basic data model of libmdbx. We will cover `MDBX_val` , the main and named tables (`MDBX_dbi`), limits, integer keys, and zero-length keys/values. These concepts define the schema of the end-to-end configurator project: which tables to create and what to store in them.

3.3 Chapter 3. Basic data model

3.3.1 3.1. MDBX_val: key and value as byte strings

All data in libmdbx is byte sequences. Neither type, nor encoding, nor structure interests the engine. A window onto such data is `MDBX_val` :

```
typedef struct MDBX_val {
    size_t iov_len; /* length in bytes */
    void *iov_base; /* pointer to data */
} MDBX_val;
```

Important: `iov_base` usually points **directly into the mmap region** of the database. Therefore:

- the value can be read without copying;
- but after the transaction ends the pointer may become invalid — copy the data if you intend to keep it longer than the transaction.

Nuance (C++): `mdbx::slice` and `mdbx::buffer`. In the C++ API two types play the role of `MDBX_val` . `mdbx::slice` is a non-owning view (a `std::string_view`-like window over `MDBX_val`): it merely references bytes owned elsewhere and never copies them. The constructor from `std::string` is `explicit` for a reason: `mdbx::slice(std::to_string(k))` written inline in a call argument is safe, because the temporary string lives until the end of the full-expression (i.e. until the whole `insert()/get()` call — which copies the bytes into the database — completes), whereas storing such a slice in a variable and using it later is a dangling reference. `mdbx::buffer` is the opposite, an owning container: by default it **copies** the content (the `make_reference=true` overloads fall back to `slice` behavior), so `mdbx::buffer(std::to_string(k))` puts no lifetime requirements on the source.

3.3.2 3.2. Main table and named tables (dbi)

Inside a single database file live **several trees**. Each tree is a "table" (in API terms — `MDBX_dbi`, map / sub-DB). There are:

- the **main table** — descriptors of all other tables (name → tree root, flags, counters);
- **named tables** — your data;
- service ones (for example, the GC tree for free pages — in detail in Volume III).

Opening a table in a transaction:

```
/* main table */
mdbx_dbi_open(txn, NULL, 0, &dbi_main);
/* named table (created on first open with MDBX_CREATE) */
mdbx_dbi_open(txn, "users", MDBX_CREATE, &dbi_users);
```

`MDBX_dbi` is not a global object but a **binding to a table within a specific transaction**: the handle is valid where it was opened and is inherited by nested transactions. Using the handle from another transaction/thread is an error (`MDBX_BAD_DBI`).

Nuance: reopening a table. For an already-existing table `mdbx_dbi_open()` requires its persistent flags (`MDBX_INTEGERKEY`, `MDBX_DUPSORT`, `MDBX_DUPFIXED`, ...); a mismatch yields `MDBX_INCOMPATIBLE`. If the flags are unknown in advance (the table may have been created by other code), open it with the `MDBX_DB_ACCEDE` flag: the table opens with its actual flags, and you can discover them via `mdbx_dbi_flags()`. An analogous `MDBX_ACCEDE` flag exists for `mdbx_env_open()` — it opens a database already used by another process in an unknown mode without an `MDBX_INCOMPATIBLE` error (Volume II, chapter 9).

3.3.3 3.3. Limits

Parameter	Value
Key	up to ~half a page (depends on the page size and table mode); integer keys — 8 bytes
Value	up to <code>MDBX_MAXDATASIZE</code> ≈ 2 GB
Tables	<code>MDBX_MAX_DBI</code> = 32765
Readers	configurable (<code>mdbx_env_set_maxreaders()</code>), up to <code>MDBX_READERS_LIMIT</code> = 32767

The page size is chosen when the database is created: 256...65536 bytes, 4096 by default, and it **does not change** afterwards. The choice affects performance (Volume IV).

The engine reports the actual limits through the `mdbx_limits_*` family — they depend on the page size and the build, so prefer functions over hardcoded numbers: `mdbx_limits_dbsize_min()`, `mdbx_limits_dbsize_max()`, `mdbx_limits_keysize_min()`, `mdbx_limits_keysize_max()`, `mdbx_limits_valsize_min()`, `mdbx_limits_valsize_max()`, `mdbx_limits_pgsz_min()`, `mdbx_limits_pgsz_max()`, `mdbx_limits_txnsize_max()`, `mdbx_limits_pairsz4page_max()`, `mdbx_limits_valsize4page_max()`. Handy aliases for individual values — `mdbx_env_get_maxkeysize(ex)` and `mdbx_env_get_maxvalsize(ex)`.

3.3.4 3.4. Integer keys and native byte order

For numeric keys use the special types: `MDBX_INTEGERKEY` (a table with integer keys `uint32_t`/`uint64_t` in **native byte order**) and `MDBX_INTEGERDUP` (integer multivalues). Native byte order means: the key 42 in memory as a `uint64_t` — without byte swapping. This is critical for comparisons and cursors.

A non-standard sort order is set by a **comparator** — a callback of type `MDBX_cmp_func`. Custom key/value comparators are passed when opening a table in the extended variants `mdbx_dbi_open_ex()`/`mdbx_dbi_open_ex2()` (the `keycmp`/`datacmp` arguments). The standard comparators for binary keys/values are `mdbx_cmp()`/`mdbx_dcmp()`; pick the one suitable for a given set of table flags via `mdbx_get_keycmp(flags)`/`mdbx_get_datacmp(flags)`.

3.3.5 3.5. Zero-length keys and values (difference from LMDB)

Unlike LMDB, libmdbx allows **empty keys** (length 0) and **empty values** (length 0). This is convenient for markers and presence flags. An empty value is `iov_len == 0` (the pointer may be NULL). Distinguish between "no key" (`MDBX_NOTFOUND`) and "the key exists with an empty value" (`MDBX_SUCCESS` with `iov_len == 0`).

Fragment from `examples/c++/02-data-model.c++` — integer keys (`key_mode::ordinal`) and a zero-length value (`mdbx::slice()`):

```
// Main table (name == nullptr): usual keys and values.
auto main = txn.open_map(nullptr);
txn.insert(main, mdbx::slice("key"), mdbx::slice("value"));
std::cout << "main[key] = " << txn.get(main, mdbx::slice("key")).as_string() << "\n";

// Named table: a separate key space within the same DB.
auto named = txn.create_map("named", mdbx::key_mode::usual, mdbx::value_mode::single);
txn.insert(named, mdbx::slice("config"), mdbx::slice("42"));
std::cout << "named[config] = " << txn.get(named, mdbx::slice("config")).as_string() << "\n";

// Integer keys: a table with key_mode::ordinal (MDBX_INTEGERKEY),
// keys are uint64_t in native byte order.
auto ord = txn.create_map("ordinal", mdbx::key_mode::ordinal, mdbx::value_mode::single);
const uint64_t k1 = 1, k2 = 1000;
txn.insert(ord, buffer::key_from_u64(k1), mdbx::slice("one"));
txn.insert(ord, buffer::key_from_u64(k2), mdbx::slice("thousand"));
std::cout << "ordinal[1000] = " << txn.get(ord, buffer::key_from_u64(k2)).as_string() << "\n";

// An empty key (zero length) and an empty value.
txn.insert(main, mdbx::slice(), mdbx::slice("empty-key-value"));
txn.insert(named, mdbx::slice("empty"), mdbx::slice());
std::cout << "empty value length = " << txn.get(named, mdbx::slice("empty")).size() << "\n";
```

Full code: [02-data-model.c++](#) · [C version](#)

Examples for this chapter: [examples/c++/02-data-model.c++](#) · [C version](#).

3.3.6 3.6. Summary of chapter 3

- `MDBX_val` = a byte window `{len, ptr}`; the pointer may live in mmap.
- One file — many tables (trees); the main table stores the descriptors.
- `MDBX_dbi` — a binding to a table in a specific transaction.
- Limits: key $\sim\frac{1}{2}$ page, value ~ 2 GB, tables 32765.
- Integer keys — native byte order; empty keys and values are allowed.

3.3.7 3.7. Exercises

1. Create two named tables ("settings", "counters") and put a key into each.
2. Try to read a key from a handle opened in another transaction — what does the API return?
3. Check the behavior with an empty value: put `iov_len=0`, read it back — how do you tell an "empty value" from "no key"?

3.3.8 3.8. Chapter 3 checklist

- [] I understand that `MDBX_val` is a byte window `{iov_len, iov_base}` with no type or encoding;
- [] I can open the main and named tables via `mdbx_dbi_open()` and know the role of `MDBX_CREATE`;
- [] I understand that `MDBX_dbi` is valid only in its own transaction, and remember `MDBX_BAD_DBI` and `MDBX_DB_ACCEDE`;
- [] I check limits through the `mdbx_limits_*` functions rather than hardcoded numbers;
- [] I know why `MDBX_INTEGERKEY` exists (native byte order) and how to tell an empty value (`iov_len == 0`) from a missing key (`MDBX_NOTFOUND`).

3.3.9 3.9. What's next

Now that the data model is clear, we can operate on it. Chapter 4 covers the basic CRUD set — `mdbx_put`, `mdbx_get`, `mdbx_del`, `mdbx_replace` — with write flags, handling of expected codes, and a full example. These operations form the core of `cfg_set` / `cfg_get` in the end-to-end configurator.

3.4 Chapter 4. Basic CRUD operations

3.4.1 4.1. mdbx_put — insert and update

```
int mdbx_put(MDBX_txn *txn, MDBX_dbi dbi,
            const MDBX_val *key, const MDBX_val *data, unsigned flags);
```

Flags:

Flag	Meaning
0 / MDBX_UPSERT	Insert or replace the value
MDBX_NOOVERWRITE	Insert only if the key is absent; otherwise MDBX_KEYEXIST
MDBX_NODUPDATA	(DUPSORT) do not add a duplicate value
MDBX_CURRENT	Update the cursor's current value

3.4.2 4.2. mdbx_get — read by key

```
int mdbx_get(MDBX_txn *txn, MDBX_dbi dbi,
            const MDBX_val *key, MDBX_val *data);
```

Returns `MDBX_SUCCESS` (found) or `MDBX_NOTFOUND` (no such key). For DUPSORT tables you need `MDBX_GET_BOTH` (to search for a specific value) — in detail in Volume II, chapter 7.

The extended variant `mdbx_get_ex(txn, dbi, &key, &data, &values_count)` additionally returns the **number of values for the key** (1 for a regular table; for DUPSORT — how many values are associated with the key; handy without a separate cursor and `mdbx_cursor_count()`).

3.4.3 4.3. mdbx_del — deletion

```
int mdbx_del(MDBX_txn *txn, MDBX_dbi dbi,
            const MDBX_val *key, const MDBX_val *data);
```

If `data == NULL` — the whole key is deleted. If data is given and the table is DUPSORT — the specific value is deleted.

Fragment from `examples/c++/03-crud.c++` — write flags on the C++ API: `insert` corresponds to `MDBX_NOOVERWRITE` (on a repeat it throws `mdbx::key_exists`), `upsert` — `MDBX_UPSERT`, `update` — `MDBX_CURRENT`. The flags `MDBX_NODUPDATA` / `MDBX_ALLDUPS` for multivalues are in Volume II, chapters 6–7:

```
// NOOVERWRITE: inserting a unique key.
txn.insert(table, mdbx::slice("key"), mdbx::slice("value1"));

// Re-inserting the same key -> key_exists (MDBX_KEYEXIST).
try {
    txn.insert(table, mdbx::slice("key"), mdbx::slice("value2"));
    std::cerr << "FAIL: duplicate insert did not throw\n";
    return EXIT_FAILURE;
} catch (const mdbx::key_exists &) {
    std::cout << "insert duplicate -> MDBX_KEYEXIST as expected\n";
}

// try_insert does not throw; it reports the result via the done flag.
auto ins = txn.try_insert(table, mdbx::slice("key"), mdbx::slice("value2"));
std::cout << "try_insert duplicate -> done=" << (ins.done ? 1 : 0) << "\n";

// UPSERT: insert or overwrite.
txn.upsert(table, mdbx::slice("key"), mdbx::slice("value2"));
std::cout << "after upsert: " << txn.get(table, mdbx::slice("key")).as_string() << "\n";

// CURRENT (update): update only an existing key.
txn.update(table, mdbx::slice("key"), mdbx::slice("value3"));
std::cout << "update existing: " << txn.get(table, mdbx::slice("key")).as_string() << "\n";
```

Full code: [03-crud.c++](#) · [C version](#)

3.4.4 4.4. mdbx_replace — conditional replacement

`mdbx_replace` combines put+get into a single operation: replace a value, learn the old one, insert only if the current value matches. It is used for atomic "compare-and-swap" without extra lookups.

```
int mdbx_replace(MDBX_txn *txn, MDBX_dbi dbi,
                const MDBX_val *key, const MDBX_val *new_data,
                const MDBX_val *old_data /* may be NULL */,
                MDBX_val *old_value /* may be NULL */, unsigned flags);
```

3.4.5 4.5. Reading: mdbx_get vs a cursor

`mdbx_get` is a point lookup by exact key. If you need to traverse data, find a range, or the "nearest greater key" — use a cursor (Volume II, chapter 6). For exact reads `mdbx_get` is enough.

3.4.6 4.6. Full example: CRUD

```
#include <stdio.h>
#include <string.h>
#include <mdbx.h>

static void die(const char *what, int rc) {
    fprintf(stderr, "%s: %s (%d)\n", what, mdbx_strerror(rc), rc);
    exit(1);
}

int main(void) {
    MDBX_env *env = NULL; MDBX_txn *txn = NULL; MDBX_dbi dbi;
    MDBX_val key, data, out;
    int rc;

    rc = mdbx_env_create(&env);
    if (rc) die("create", rc);
    rc = mdbx_env_open(env, "./crud.mdbx", MDBX_NOSUBDIR, 0664);
    if (rc) die("open", rc);
    rc = mdbx_txn_begin(env, NULL, MDBX_TXN_READWRITE, &txn);
    if (rc) die("txn", rc);
    rc = mdbx_dbi_open(txn, "kv", MDBX_CREATE, &dbi);
    if (rc) die("dbi", rc);

    /* insert */
    const char *k1 = "alpha", *v1 = "1";
    key.iov_base = (void *)k1; key.iov_len = strlen(k1);
    data.iov_base = (void *)v1; data.iov_len = strlen(v1);
    rc = mdbx_put(txn, dbi, &key, &data, MDBX_NOOVERWRITE);
    if (rc != MDBX_SUCCESS && rc != MDBX_KEYEXIST) die("put", rc);

    /* update */
    const char *v2 = "2";
    data.iov_base = (void *)v2; data.iov_len = strlen(v2);
    rc = mdbx_put(txn, dbi, &key, &data, 0); /* UPSERT */
    if (rc) die("put2", rc);

    /* get */
    out.iov_base = NULL; out.iov_len = 0;
    rc = mdbx_get(txn, dbi, &key, &out);
    if (rc) die("get", rc);
    printf("alpha = %.*s\n", (int)out.iov_len, (char *)out.iov_base);

    /* replace: read the old value and write the new one */
    MDBX_val old = {0, NULL};
    const char *v3 = "3";
    data.iov_base = (void *)v3; data.iov_len = strlen(v3);
    rc = mdbx_replace(txn, dbi, &key, &data, NULL, &old, 0);
    if (rc) die("replace", rc);
    printf("old value: %.*s\n", (int)old.iov_len, (char *)old.iov_base);

    /* del */
    rc = mdbx_del(txn, dbi, &key, NULL);
    if (rc) die("del", rc);

    /* verify the deletion */
    rc = mdbx_get(txn, dbi, &key, &out);
    printf("get after deletion: %s (%d)\n",
           rc == MDBX_NOTFOUND ? "NOTFOUND (expected)" : "unexpected", rc);

    mdbx_txn_commit(txn);
    mdbx_env_close(env);
    return 0;
}
```

Warning: `old_value` from `mdbx_replace` and `out` from `mdbx_get` point into mmap. Do not keep these pointers after `commit` / `abort` — the data may be reused.

3.4.7 4.7. Error handling

Return codes: `MDBX_SUCCESS` (0) — success; `MDBX_RESULT_TRUE` / `MDBX_RESULT_FALSE` — special results of some operations; negative `MDBX_*` — errors.

Common "expected" codes (these are not failures but normal responses):

- `MDBX_KEYEXIST` — the key already exists (with `MDBX_NOOVERWRITE`);
- `MDBX_NOTFOUND` — the key (or value) was not found;
- `MDBX_BAD_DBI` — invalid/closed table handle.

String description: `mdbx_strerror(rc)`. There is also a thread-safe variant `mdbx_strerror_r()` — use it in multithreaded code (Volume II, chapter 12).

Examples for this chapter: [examples/c++/03-crud.c++](#) · [C version](#); end-to-end project: [config-store-04.c++](#).

3.4.8 4.8. Summary of chapter 4

- put/get/del/replace — basic operations; flags control behavior on conflicts.
- `mdbx_get` — point reads; for traversal/ranges — cursors.
- `MDBX_KEYEXIST` / `MDBX_NOTFOUND` are expected states, not errors.
- `mdbx_replace` — atomic "read old + write new" replacement in a single call.
- Pointers into mmap are valid only within a transaction.

3.4.9 4.9. Exercises

1. Add "expected error" handling: insert a key again with `NOOVERWRITE` and handle `MDBX_KEYEXIST` without exiting the program.
2. Write a loop that inserts 10,000 keys in a **single** transaction and measures the time.
3. What does `mdbx_del` do with the `data` argument in an ordinary (non-DUPSORT) table?

3.4.10 4.10. Chapter 4 checklist

- [] I can perform all the basic operations: `mdbx_put`, `mdbx_get`, `mdbx_del`, and `mdbx_replace`;
- [] I understand the difference between the write flags `MDBX_NOOVERWRITE`, `MDBX_UPSERT`, and `MDBX_CURRENT`, and choose them deliberately;
- [] I treat `MDBX_KEYEXIST` and `MDBX_NOTFOUND` as normal states, not failures;
- [] I understand that `mdbx_replace` is an atomic "compare and swap" in a single call;
- [] I remember that pointers into mmap (`out`, `old_value`) must not be used after commit/abort.

3.4.11 4.11. What's next

You can already perform single operations, but the real power of libmdbx is in transactions. Chapter 5 introduces `begin/commit/abort`, the difference between read-only and read-write transactions, and the MVCC intuition — why readers never block the writer. For the configurator this is the foundation: atomic writes of groups of settings and safe concurrent access to the database.

3.5 Chapter 5. Transactions — first steps

3.5.1 5.1. What a transaction is and why you need it

A transaction is a group of operations executed **atomically** and **in isolation**:

- either all changes are applied, or none are (atomicity);
- until the transaction is committed, its changes are not visible to others (isolation).

Example: a money transfer — debit account A and credit account B. If you do this as two separate operations, you can lose data between them on a failure. In a transaction — it is safe.

3.5.2 5.2. begin / commit / abort

```
int mdbx_txn_begin(MDBX_env *env, MDBX_txn *parent,
                  unsigned flags, MDBX_txn **txn);
int mdbx_txn_commit(MDBX_txn *txn);
int mdbx_txn_abort(MDBX_txn *txn);
```

- `mdbx_txn_begin(env, NULL, flags, &txn)` — start a transaction. `flags == 0` (same as `MDBX_TXN_READWRITE`) — read-write; for a read-only transaction pass `MDBX_TXN_RDONLY`.
- `mdbx_txn_commit(txn)` — apply the changes (for write) or simply close (for read).
- `mdbx_txn_abort(txn)` — discard the changes and close.

Tip: always close a transaction — either commit or abort. An unclosed read transaction can "freeze" page reclamation (Volume III, chapter 16).

3.5.3 5.3. Read-only vs read-write

Read-only transactions do not block each other and do not require the writer mutex. You can have many of them at once, in any threads.

There is **one read-write transaction** in the database at any moment (the global writer mutex). If two threads try to start a write transaction at the same time — one waits for the other.

3.5.4 5.4. Why readers do not block each other (MVCC intuitively)

libmdbx uses **MVCC** (multiversion concurrency control). Simplified:

1. Every transaction sees a "snapshot" of the database — the state at the moment it started.
2. A write transaction does not modify existing pages but creates **new versions** (Copy-on-Write).
3. A reader that started earlier keeps seeing the old versions as long as its snapshot is alive.

Therefore readers never conflict either with each other or with the writer. The price — old page versions take up space while old snapshots are alive (in detail — Volume III, chapters 14 and 16).

Fragment from `examples/c++/04-transactions.c++` — a read-only observer in a separate thread sees a stable MVCC snapshot while a write transaction overwrites the key and then does an `abort` (rollback):

```
{
    auto txn = env.start_write();
    auto table = txn.open_map(nullptr);
    txn.insert(table, mdbx::slice("key"), mdbx::slice("v1"));
    txn.commit();
}

// Read-only observer in a separate thread: a fixed snapshot of the database.
std::thread observer(observer_run, std::ref(env));
while (stage.load() != 1)
    std::this_thread::yield();

// The write transaction overwrites the key but does not commit yet.
{
    auto wtxn = env.start_write();
    auto table = wtxn.open_map(nullptr);
    wtxn.upsert(table, mdbx::slice("key"), mdbx::slice("v2"));
    stage = 2; // the observer checks the snapshot with an uncommitted write
    while (stage.load() != 3)
        std::this_thread::yield();
    wtxn.abort(); // rollback: v2 is not saved
}
```

Full code: [04-transactions.c++](#) · [C version](#)

3.5.5 5.5. Basic rule: one transaction — one thread

A transaction is bound to the thread that created it ("sticky thread"). You cannot use a transaction object from another thread — that is a discipline violation (`MDBX_THREAD_MISMATCH`). However, there is the `MDBX_NOSTICKYTHREADS` mode for thread pools and coroutines (Volume II, chapter 11).

3.5.6 5.6. Full example with transactions

```
#include <stdio.h>
#include <string.h>
#include <mdbx.h>

static void die(const char *what, int rc) {
    fprintf(stderr, "%s: %s (%d)\n", what, mdbx_strerror(rc), rc);
    exit(1);
}

int main(void) {
    MDBX_env *env = NULL; MDBX_txn *txn = NULL; MDBX_dbi dbi;
    MDBX_val key, data, out;
    int rc;

    rc = mdbx_env_create(&env);
    if (rc) die("create", rc);
    rc = mdbx_env_open(env, "/.txn.mdbx", MDBX_NOSUBDIR, 0664);
    if (rc) die("open", rc);

    /* ---- Write transaction: two keys atomically ---- */
    rc = mdbx_txn_begin(env, NULL, MDBX_TXN_READWRITE, &txn);
    if (rc) die("txn_w", rc);
    rc = mdbx_dbi_open(txn, "kv", MDBX_CREATE, &dbi);
    if (rc) die("dbi", rc);

    const char *ka = "a", *va = "1";
    const char *kb = "b", *vb = "2";
    key.iov_base = (void *)ka; key.iov_len = strlen(ka);
    data.iov_base = (void *)va; data.iov_len = strlen(va);
    mdbx_put(txn, dbi, &key, &data, 0);
    key.iov_base = (void *)kb; key.iov_len = strlen(kb);
    data.iov_base = (void *)vb; data.iov_len = strlen(vb);
    mdbx_put(txn, dbi, &key, &data, 0);
    rc = mdbx_txn_commit(txn);
    if (rc) die("commit", rc);

    /* ---- Read-only transaction ---- */
    rc = mdbx_txn_begin(env, NULL, MDBX_TXN_RDONLY, &txn);
    if (rc) die("txn_r", rc);
    out.iov_base = NULL; out.iov_len = 0;
    key.iov_base = (void *)ka; key.iov_len = strlen(ka);
    rc = mdbx_get(txn, dbi, &key, &out);
    if (rc) die("get", rc);
    printf("a = %s\n", (int)out.iov_len, (char *)out.iov_base);
    mdbx_txn_abort(txn); /* read-only: just close it */

    /* ---- Abort: changes are discarded ---- */
    rc = mdbx_txn_begin(env, NULL, MDBX_TXN_READWRITE, &txn);
    if (rc) die("txn_w2", rc);
    const char *kc = "c", *vc = "3";
    key.iov_base = (void *)kc; key.iov_len = strlen(kc);
    data.iov_base = (void *)vc; data.iov_len = strlen(vc);
    mdbx_put(txn, dbi, &key, &data, 0);
    mdbx_txn_abort(txn); /* c will not appear in the database */

    /* verify that "c" is absent */
    rc = mdbx_txn_begin(env, NULL, MDBX_TXN_RDONLY, &txn);
    if (rc) die("txn_r2", rc);
    rc = mdbx_get(txn, dbi, &key, &out);
    printf("after abort: %s\n", rc == MDBX_NOTFOUND ? "no key (correct)" : "unexpected");
    mdbx_txn_abort(txn);

    mdbx_env_close(env);
    return 0;
}
```

Keep reading after commit — `mdbx_txn_commit_embark_read()` . A commit in libmdbx "seats" the transaction on a fresh snapshot (the detent — a reader held on a committed snapshot; the mechanism is detailed in Volume III); the dedicated function `mdbx_txn_commit_embark_read()` does the same and immediately returns a **new read-only transaction on the already-committed state** — with no "commit → new begin" gap. Useful for read-your-writes patterns and queue processing (Volume V, chapter 32). The transaction identifier (the ordinal number in the database) is available via `mdbx_txn_id()` .

3.5.7 5.7. End-to-end project: the beginning

Our application configurator. The idea: store settings as "section → key → value" tables.

```
/* config_store.h — the end-to-end project skeleton (chapters 2-5) */
#ifndef CONFIG_STORE_H
#define CONFIG_STORE_H
#include <mdbx.h>

typedef struct {
    MDBX_env *env;
    MDBX_dbi main;
} config_store_t;

int cfg_open(config_store_t *cs, const char *path);
int cfg_set(config_store_t *cs, const char *key, const char *value);
int cfg_get(config_store_t *cs, const char *key, char *buf, size_t buflen);
int cfg_close(config_store_t *cs);

#endif
```

The implementation comes in the following chapters; by the end of Volume II this will be a full-fledged application with indexes and multithreading.

Examples for this chapter: [examples/c++/04-transactions.c++](#) · [C version](#); end-to-end project: [config-store-05.c++](#).

3.5.8 5.8. Summary of chapter 5

- A transaction = an atomic, isolated group of operations.
- begin (0/ `MDBX_TXN_READWRITE` — write; `MDBX_TXN_RDONLY` — read-only), commit, abort.
- MVCC: readers see a snapshot and do not block the writer.
- One transaction — one thread (by default).
- Unclosed read transactions "freeze" page reclamation.

3.5.9 5.9. Exercises

1. Write a `cfg_set` function that uses `MDBX_NOOVERWRITE` for creation and `0` for updates — how do you combine them in a single transaction?
2. Measure: how many write transactions per second does your disk sustain (without batching)?
3. Why is starting a read-write transaction "just in case" a bad idea (hint: the writer mutex)?

3.5.10 5.10. Chapter 5 checklist

- [] I understand the transaction properties: atomicity (all or nothing) and isolation;
- [] I can start and correctly close every kind of transaction: `MDBX_TXN_READWRITE`, `MDBX_TXN_RDONLY`, `mdbx_txn_abort()`;
- [] I understand why there is always a single write transaction and how MVCC lets readers proceed without blocking the writer;
- [] I know the rule "one transaction — one thread" and the `MDBX_THREAD_MISMATCH` code;
- [] I can demonstrate on an example that abort discards changes and that an unclosed read transaction stalls page reclamation.

3.5.11 5.11. What's next

Volume I is complete: you have a working configurator skeleton. Volume II turns it into a full-fledged application: cursors for traversal and ranges (chapter 6), DUPSORT and secondary indexes (chapters 7–8), environment configuration (chapter 9), durability modes (chapter 10), multithreading (chapter 11), and error handling (chapter 12). We start with cursors — without them you cannot traverse a table or search a range.

3.6 Volume summary

You have learned the basic model of libmdbx: what it is, how to build it, how to open a database, and how to perform CRUD in transactions. Your end-to-end "configurator" project works.

What's next: Volume II — cursors, multivalues and DUPSORT, secondary indexes, environment configuration, durability modes, multithreading, and error handling.

4. Volume II. Practical Usage

Level: for developers who have already mastered Volume I. **Goal of the volume:** you will learn to build applications on libmdbx — cursors, multivalues and DUPSORT, secondary indexes, environment configuration, durability modes, multithreading, and robust error handling. **End-to-end project:** we grow the application configurator into a full-fledged application with indexes and multithreading.

Chapter pattern: concept → mechanism → practice → nuance.

4.1 Chapter 6. Cursors

4.1.1 6.1. What is a cursor and why do you need one

`mdbx_get` only does point lookups by an exact key. For everything else — traversing a table, range search, "the nearest greater key", iteration with deletion — you need a **cursor**.

A cursor is a "pointer" to a position inside the tree (more precisely, a stack of positions from the root to the leaf). It lets you move through the ordered keys in both directions and perform operations relative to the current position.

4.1.2 6.2. Opening and closing

```
int mdbx_cursor_open(MDBX_txn *txn, MDBX_dbi dbi, MDBX_cursor **cursor);
int mdbx_cursor_close(MDBX_cursor *cursor);
```

A cursor belongs to its transaction: it is created inside it and lives no longer than it. After `commit` / `abort` the cursor must not be closed — use it only until the transaction ends.

4.1.3 6.3. Positioning

```
int mdbx_cursor_get(MDBX_cursor *cur, MDBX_val *key, MDBX_val *data, MDBX_cursor_op op);
```

Key operations:

Operation	Meaning
<code>MDBX_FIRST</code> / <code>MDBX_LAST</code>	Move to the first / last key
<code>MDBX_NEXT</code> / <code>MDBX_PREV</code>	Next / previous key
<code>MDBX_SET</code>	Find the exact key (position on it)
<code>MDBX_SET_RANGE</code>	Find the first key $\geq$ the given one ("nearest greater or equal")
<code>MDBX_SET_LOWERBOUND</code>	Like <code>SET_RANGE</code> , but requires a valid data argument for DUPSORT
<code>MDBX_SET_UPPERBOUND</code>	Find the first key $>$ the given one
<code>MDBX_GET_BOTH</code> / <code>GET_BOTH_RANGE</code>	(DUPSORT) find a specific value / the first $\geq$ the value

After `MDBX_LAST`, when the data runs out, the operation returns `MDBX_NOTFOUND`, and the cursor lands "at the end" — the `eof` state. This is a normal situation for finishing a traversal.

For frequent scenarios there are dedicated convenience functions instead of a "cursor_get(op) + code check" pair: `mdbx_cursor_on_first()`, `mdbx_cursor_on_last()` and their dup variants `mdbx_cursor_on_first_dup()` / `mdbx_cursor_on_last_dup()`; to test "end of traversal" — the logical `mdbx_cursor_eof()`. The distance between two cursor positions can be found via `mdbx_cursor_distance()` — useful to size up a range before processing it.

Fragment from [examples/c++/05-cursors.c++](#) — forward iteration (FIRST/NEXT) and `SET_RANGE` ("nearest greater or equal"):

```
auto cur = rxn.open_cursor(table);

// Forward iteration: FIRST then NEXT until the end.
size_t forward_count = 0;
std::string forward;
for (auto r = cur.to_first(); r; r = cur.to_next(false)) {
    forward += r.key.as_string() + " ";
    ++forward_count;
}
std::cout << "forward (" << forward_count << "): " << forward << "\n";

// SET_RANGE: the first key not less than the given one.
auto r = cur.to_key_greater_or_equal(mdbx::slice("k5"), false);
std::cout << "set_range(k5) -> ";
if (r)
    std::cout << r.key.as_string() << "\n";
else
    std::cout << "<none>\n";
r = cur.to_key_greater_or_equal(mdbx::slice("k5x"), false);
std::cout << "set_range(k5x) -> ";
if (r)
    std::cout << r.key.as_string() << "\n";
else
    std::cout << "<none>\n";
```

Full code: [05-cursors.c++](#) · [C version](#)

4.1.4 6.4. Example: table traversal and range search

```
#include <stdio.h>
#include <string.h>
#include <mdbx.h>

static void die(const char *w, int rc) {
    fprintf(stderr, "%s: %s (%d)\n", w, mdbx_strerror(rc), rc);
    exit(1);
}

int main(void) {
    MDBX_env *env = NULL; MDBX_txn *txn = NULL; MDBX_dbi dbi;
    MDBX_cursor *cur = NULL;
    MDBX_val k, d;
    int rc;

    rc = mdbx_env_create(&env); if (rc) die("create", rc);
    rc = mdbx_env_open(env, "./cur.mdbx", MDBX_NOSUBDIR, 0664);
    if (rc) die("open", rc);
    rc = mdbx_txn_begin(env, NULL, MDBX_TXN_READWRITE, &txn);
    if (rc) die("txn", rc);
    rc = mdbx_dbi_open(txn, "kv", MDBX_CREATE, &dbi);
    if (rc) die("dbi", rc);

    /* fill the table */
    const char *words[] = {"alpha", "bravo", "charlie", "delta", "echo"};
    for (int i = 0; i < 5; i++) {
        k.iov_base = (void *)words[i]; k.iov_len = strlen(words[i]);
        d.iov_base = (void *)"x"; d.iov_len = 1;
        rc = mdbx_put(txn, dbi, &k, &d, 0); if (rc) die("put", rc);
    }

    /* full traversal */
    rc = mdbx_cursor_open(txn, dbi, &cur); if (rc) die("cursor", rc);
    printf("All keys:\n");
    while ((rc = mdbx_cursor_get(cur, &k, &d, MDBX_NEXT)) == MDBX_SUCCESS)
        printf(" %.*s\n", (int)k.iov_len, (char *)k.iov_base);
    /* at the end MDBX_NOTFOUND is returned — this is normal */
    if (rc != MDBX_NOTFOUND) die("next", rc);

    /* range: the first key >= "c" and all the following */
    printf("Range >= c:\n");
    const char *from = "c";
    k.iov_base = (void *)from; k.iov_len = strlen(from);
    rc = mdbx_cursor_get(cur, &k, &d, MDBX_SET_RANGE);
    if (rc == MDBX_SUCCESS) {
        do {
            printf(" %.*s\n", (int)k.iov_len, (char *)k.iov_base);
        } while ((rc = mdbx_cursor_get(cur, &k, &d, MDBX_NEXT)) == MDBX_SUCCESS);
    }
    if (rc != MDBX_NOTFOUND) die("range", rc);

    mdbx_cursor_close(cur);
    mdbx_txn_commit(txn);
    mdbx_env_close(env);
    return 0;
}
```

4.1.5 6.5. Cursor after deletion — UB (critical warning)

Warning: after `mdbx_cursor_del()` the cursor position is undefined. Using the cursor without re-positioning is undefined behavior. Always do `NEXT` (or `PREV`) after deletion and check `MDBX_NOTFOUND`.

4.1.6 6.6. Cloning cursors

`mdbx_cursor_clone()` duplicates the cursor position (for parallel processing on a single snapshot), `mdbx_cursor_bind()` binds an existing cursor to another transaction/table.

4.1.7 6.7. The safe-delete pattern in DUPSORT

Deleting values in a DUPSORT table with a single cursor during iteration is dangerous (known bugs in 0.12.x, details in Volume V). The safe pattern is **two cursors**: one positions, the other deletes.

Fragment (C, illustration): the full compilable version is below and in [examples/c++/06-dupsort-delete.c++](#).

```
/* delete all values of the "target" key in a DUPSORT table */
MDBX_cursor *it, *del;
mdbx_cursor_open(txn, dbi, &it);
mdbx_cursor_open(txn, dbi, &del);
const char *tgt = "target";
k.iov_base = (void *)tgt; k.iov_len = strlen(tgt);
rc = mdbx_cursor_get(it, &k, &d, MDBX_SET); /* first dup */
while (rc == MDBX_SUCCESS) {
    MDBX_val dk = k, dv = d;
    /* second cursor on the same position */
    rc = mdbx_cursor_get(del, &dk, &dv, MDBX_GET_BOTH);
    if (rc != MDBX_SUCCESS) break;
    rc = mdbx_cursor_del(del, MDBX_NODUPDATA);
    if (rc) break;
    rc = mdbx_cursor_get(it, &k, &d, MDBX_NEXT_DUP);
}
mdbx_cursor_close(it);
mdbx_cursor_close(del);
```

For bulk deletion of whole ranges, better use `mdbx_cursor_bunch_delete()` ("bunch deletion") — it cuts out whole pages and branches rather than iterating elements.

Fragment from [examples/c++/06-dupsort-delete.c++](#) — the safe-delete pattern with two cursors: `it` iterates over duplicates (`NEXT_DUP`), `del` positions via `GET_BOTH` and deletes the current value:

```
auto txn = env.start_write();
auto multi = txn.open_map("multi", mdbx::key_mode::usual, mdbx::value_mode::multi);
dump_key(txn, multi, "values before delete (target):", mdbx::slice("target"));

// The "two cursors" pattern: 'it' iterates over the key's duplicates,
// 'del' deletes the current pair (the 'it' position stays valid).
const mdbx::slice target("target");
auto it = txn.open_cursor(multi);
auto del = txn.open_cursor(multi);
size_t deleted = 0;
// Keep the first dup of the key, delete the rest.
auto pos = it.to_key_exact(target);
if (pos)
    pos = it.to_current_next_multi(false);
for (; pos; pos = it.to_current_next_multi(false)) {
    del.to_exact_key_value_equal(pos.key, pos.value, false); // MDBX_GET_BOTH
    if (del.erase(false)) // MDBX_CURRENT: delete only the current value
        ++deleted;
}
```

Full code: [06-dupsort-delete.c++](#) · [C version](#)

Examples for this chapter: [examples/c++/05-cursors.c++](#) · [C version](#); [examples/c++/06-dupsort-delete.c++](#) · [C version](#); end-to-end project: [config-store-06.c++](#).

4.1.8 6.8. Summary of chapter 6

- A cursor is a position in the ordered tree; `mdbx_cursor_get(op)` controls movement.
- `MDBX_SET_RANGE` is the foundation of range queries.

- After `cursor_del` the position is invalid — re-position.
- For deletion in DUPSORT — two cursors or `bunch_delete`.

4.1.9 6.9. Exercises

1. Write a function that walks all keys from end to start (`MDBX_LAST` + `MDBX_PREV`).
2. Find "the key after X" in two ways: `SET_RANGE` and `SET_UPPERBOUND` .
3. Delete every second key during a traversal and explain why the positioning must be refreshed.

4.1.10 6.10. Chapter 6 checklist

- [] I can open and close a cursor and understand that it lives no longer than its transaction;
- [] I use `MDBX_FIRST` / `MDBX_LAST` / `MDBX_NEXT` / `MDBX_PREV` and `MDBX_SET_RANGE` fluently for range queries;
- [] I remember that the position is undefined after `mdbx_cursor_del()` and always re-position;
- [] I can apply the "two cursors" pattern for safe deletion in DUPSORT;
- [] I can walk a table from end to start (`MDBX_LAST` + `MDBX_PREV`) and tell end-of-traversal (`MDBX_NOTFOUND` , `mdbx_cursor_eof()`) from an error.

4.1.11 6.11. What's next

The next chapter extends the data model: the `MDBX_DUPSORT` flag turns a table into a multimap "key → ordered set of values". We will cover the storage forms, the `MDBX_DUPFIXED`, `MDBX_INTEGERDUP`, `MDBX_REVERSEDUP` flags, navigation over duplicates, and the key technique — the inverted index "field → list of IDs". It becomes the basis of the configurator's secondary indexes in chapter 8.

4.2 Chapter 7. Multivalues and DUPSORT

4.2.1 7.1. What is DUPSORT

A regular table stores one key → one value. The `MDBX_DUPSORT` flag turns the table into a **multimap**: one key → **an ordered set of values**. The value plays the role of a "second key" with its own sort order.

Example: user tags `user:1001` → { "admin", "staff", "vip" }.

4.2.2 7.2. Storage forms

Internally (details in Volume III) the representation of values is chosen by their count:

- a few values per key — a dense **sub-page** (a nested page inside the leaf);
- many values — a separate **nested B+tree**;
- fixed-size values — specialized **dupfix pages** with dense packing.

An important consequence: a "multivalue" never goes to overflow pages, so it is limited by the key's frame (about half a page).

4.2.3 7.3. DUPSORT flags

Flag	Meaning
<code>MDBX_DUPSORT</code>	The table is a multimap
<code>MDBX_DUPFIXED</code>	All values have a fixed length (requires equal lengths! otherwise <code>MDBX_BAD_VALSIZE</code>)
<code>MDBX_INTEGERDUP</code>	Values are <code>uint32_t</code> / <code>uint64_t</code> in native order (requires <code>DUPFIXED</code> + <code>DUPSORT</code>)
<code>MDBX_REVERSEDUP</code>	Reverse sort order of values

4.2.4 7.4. Operations with multivalues

```
/* add a value to a key (MDBX_NODUPDATA protects against duplicates) */
mdbx_put(txn, dbi, &key, &data, MDBX_NODUPDATA);

/* find a specific value */
mdbx_cursor_get(cur, &key, &data, MDBX_GET_BOTH); /* exactly */
mdbx_cursor_get(cur, &key, &data, MDBX_GET_BOTH_RANGE); /* the first >= data */

/* navigation over the key's values */
mdbx_cursor_get(cur, &key, &data, MDBX_FIRST_DUP);
mdbx_cursor_get(cur, &key, &data, MDBX_LAST_DUP);
mdbx_cursor_get(cur, &key, &data, MDBX_NEXT_DUP);
mdbx_cursor_get(cur, &key, &data, MDBX_PREV_DUP);

/* number of the key's values */
size_t count;
mdbx_cursor_count(cur, &count);
```

4.2.5 7.5. Inverted indexes: the "key → list of IDs" pattern

The classic use of DUPSORT is an **inverted index**: field value → list of record identifiers. This is a "cheap secondary index" without a separate link table.

Fragment (C, illustration): the full compilable version is below and in [examples/c++/08-inverted-index.c++](#).

```
/* index "role" -> list of user_id */
MDBX_dbi idx_role;
mdbx_dbi_open(txn, "idx_role", MDBX_CREATE | MDBX_DUPSORT, &idx_role);

/* when creating a user: role = "admin", id = 1001 */
MDBX_val k = { (void *)"admin", 5 };
MDBX_val v = { &user_id_u64, sizeof(user_id_u64) };
mdbx_put(txn, idx_role, &k, &v, MDBX_NODUPDATA);
```

Getting the list of admins is a cursor over `idx_role` with the key "admin" and `NEXT_DUP`.

Fragment from [examples/c++/08-inverted-index.c++](#) — building an inverted index "word → list of document IDs" over a DUPSORT table (`value_mode::multi`):

```
{
    auto txn = env.start_write();
    auto docs = txn.create_map("docs", mdbx::key_mode::usual, mdbx::value_mode::single);
    // Index: key — a word, values — document IDs (multivalues).
    auto words = txn.create_map("words", mdbx::key_mode::usual, mdbx::value_mode::multi);

    static const struct {
        const char *id;
        const char *text;
    } entries[] = {{"doc0", "the quick brown fox"}, {"doc1", "quick red fox"}, {"doc2", "lazy brown dog"}};

    for (const auto &e : entries) {
        txn.insert(docs, mdbx::slice(e.id), mdbx::slice(e.text));
        std::istringstream stream(e.text);
        std::string word;
        while (stream >> word)
            txn.upsert(words, mdbx::slice(word), mdbx::slice(e.id)); // UPSERT appends the ID to the word
    }
    txn.commit();
}
```

Full code: [08-inverted-index.c++](#) · [C version](#)

4.2.6 7.6. Full example: a secondary index on DUPSORT

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <mdbx.h>

static void die(const char *w, int rc) {
    fprintf(stderr, "%s: %s (%d)\n", w, mdbx_strerror(rc), rc);
    exit(1);
}

int main(void) {
    MDBX_env *env = NULL; MDBX_txn *txn = NULL;
    MDBX_dbi users, by_role;
    MDBX_val k, v, d;
    MDBX_cursor *cur = NULL;
```

```

int rc;

rc = mdbx_env_create(&env); if (rc) die("create", rc);
rc = mdbx_env_open(env, "./idx.mdbx", MDBX_NOSUBDIR, 0664);
if (rc) die("open", rc);
rc = mdbx_txn_begin(env, NULL, MDBX_TXN_READWRITE, &txn);
if (rc) die("txn", rc);

mdbx_dbi_open(txn, "users", MDBX_CREATE | MDBX_INTEGERKEY, &users);
mdbx_dbi_open(txn, "by_role", MDBX_CREATE | MDBX_DUPSORT, &by_role);

/* user_id -> name; by_role["admin"] -> {id,...} */
struct { uint64_t id; const char *name; const char *role; } rows[] = {
    {1001, "Anna", "admin"}, {1002, "Boris", "user"},
    {1003, "Victor", "admin"}, {1004, "Galina", "user"},
};
for (int i = 0; i < 4; i++) {
    k.iov_base = &rows[i].id; k.iov_len = sizeof(uint64_t);
    v.iov_base = (void *)rows[i].name; v.iov_len = strlen(rows[i].name);
    mdbx_put(txn, users, &k, &v, 0);

    k.iov_base = (void *)rows[i].role; k.iov_len = strlen(rows[i].role);
    v.iov_base = &rows[i].id; v.iov_len = sizeof(uint64_t);
    mdbx_put(txn, by_role, &k, &v, MDBX_NODUPDATA);
}

/* print all admins via the index */
const char *role = "admin";
k.iov_base = (void *)role; k.iov_len = strlen(role);
mdbx_cursor_open(txn, by_role, &cur);
rc = mdbx_cursor_get(cur, &k, &d, MDBX_SET);
if (rc == MDBX_SUCCESS) {
    printf("Admins:\n");
    do {
        printf("  user id = %llu\n",
            (unsigned long long)*(uint64_t *)d.iov_base);
    } while ((rc = mdbx_cursor_get(cur, &k, &d, MDBX_NEXT_DUP)) == MDBX_SUCCESS);
}
mdbx_cursor_close(cur);

mdbx_txn_commit(txn);
mdbx_env_close(env);
return 0;
}

```

Examples for this chapter: [examples/c++/07-dupsort.c++](#) · C version; [examples/c++/08-inverted-index.c++](#) · C version.

4.2.7 7.7. Summary of chapter 7

- `MDBX_DUPSORT` = key → ordered set of values.
- `DUPFIXED` requires equal lengths; `INTEGERDUP` — integer values in native order.
- Navigation over values: `FIRST_DUP`/`LAST_DUP`/`NEXT_DUP`/`PREV_DUP`/`GET_BOTH`.
- An inverted index on `DUPSORT` is a cheap secondary index.

4.2.8 7.8. Exercises

1. Build an index "by age" and print all users older than 30.
2. Explain why `MDBX_NODUPDATA` is needed when adding to a `DUPSORT` index.
3. What happens on an attempt to `DUPFIXED`-insert a value of a different length?

4.2.9 7.9. Chapter 7 checklist

- [] I understand that `MDBX_DUPSORT` means "key → ordered set of values", with the value acting as a second key;
- [] I know when `MDBX_DUPFIXED` (equal value lengths) and `MDBX_INTEGERDUP` are required;
- [] I can locate and iterate the values of a key: `MDBX_GET_BOTH`, `MDBX_GET_BOTH_RANGE`, `MDBX_FIRST_DUP` / `NEXT_DUP`, `mdbx_cursor_count()`;
- [] I add values with `MDBX_NODUPDATA` to avoid duplicates;
- [] I can build an inverted index "word → list of IDs" over a `DUPSORT` table.

4.2.10 7.10. What's next

Multivalues are a half-finished product; chapter 8 assembles them into a complete technique: the secondary index. We will build the "main table + index tables" schema, learn to keep them consistent in a single transaction, compare a DUPSORT index with a separate table, and meet composite keys and deletion via an index. For the end-to-end project this is a step toward a "mini-ORM" with a users entity.

4.3 Chapter 8. Secondary indexes

4.3.1 8.1. Why secondary indexes are needed in a KV database

There are no SQL indexes in a key-value database: data is searched only by the primary key. To search "by another field", we build the index **ourselves** — a separate table whose key is the sought field and whose value is the primary key of the original record.

4.3.2 8.2. Pattern: main table + index tables

```
users:      user_id -> {name, email, ...}      (primary)
idx_email:  email   -> user_id                 (index)
idx_by_role: role    -> user_id (DUPSORT)      (index)
```

Keeping consistency is **on your side**: on every change to `users` update all indexes in the **same transaction** (otherwise there will be drift).

Fragment from `examples/c++/09-secondary-index.c++` — the main table and the index are updated in one transaction: on user insertion — and on deletion (the record + its index references):

```
{
    auto txn = env.start_write();
    auto users = txn.create_map("users", mdbx::key_mode::usual, mdbx::value_mode::single);
    // Index: key — a role, values — user IDs (multivalues).
    auto by_role = txn.create_map("by_role", mdbx::key_mode::usual, mdbx::value_mode::multi);

    // Inserting users and maintaining the index — in one transaction.
    txn.insert(users, mdbx::slice("user1"), mdbx::slice("Alice|admin"));
    txn.upsert(by_role, mdbx::slice("admin"), mdbx::slice("user1"));
    txn.insert(users, mdbx::slice("user2"), mdbx::slice("Bob|dev"));
    txn.upsert(by_role, mdbx::slice("dev"), mdbx::slice("user2"));
    txn.insert(users, mdbx::slice("user3"), mdbx::slice("Carol|admin"));
    txn.upsert(by_role, mdbx::slice("admin"), mdbx::slice("user3"));
    txn.commit();
}

// Deleting a user together with their index entry.
{
    auto txn = env.start_write();
    auto users = txn.open_map("users", mdbx::key_mode::usual, mdbx::value_mode::single);
    auto by_role = txn.open_map("by_role", mdbx::key_mode::usual, mdbx::value_mode::multi);
    txn.erase(users, mdbx::slice("user2"));
    txn.erase(by_role, mdbx::slice("dev"), mdbx::slice("user2")); // a specific key's value
    txn.commit();
}
```

Full code: `09-secondary-index.c++`

Nuance: opening a table with unknown flags. If a table may have been created by other code (and its persistent flags — `MDBX_DUPSORT`, `MDBX_INTEGERKEY`, etc. — are unknown in advance), pass `MDBX_DB_ACCEDE` when opening it: instead of `MDBX_INCOMPATIBLE` the table opens with its actual flags. You can discover them via `mdbx_dbi_flags()` (the exact variant — `mdbx_dbi_flags_ex()` — additionally returns the table state bits: `MDBX_DBI_CREAT` / `MDBX_DBI_DIRTY` / `MDBX_DBI_FRESH` / `MDBX_DBI_STALE`). The same applies to reopening tables in a new transaction: an `MDBX_dbi` handle from an old transaction does not carry over; and `mdbx_dbi_open()` in a new transaction without the same flags returns `MDBX_INCOMPATIBLE`.

4.3.3 8.3. DUPSORT index vs a separate table

- **DUPSORT index** ("field → list of IDs") is ideal for "one-to-many": the recipient list is the set of values of one key. Fast add/remove of a single ID.
- **Separate table** ("composite key → marker") — when you need more data about the relationship or composite conditions.

4.3.4 8.4. Composite indexes

A composite key is a concatenation of fields in a single key. Byte order is critical: for a numeric field inside a composite key to compare correctly, it must be converted to **big-endian** order (or use integer types of native order with a separate comparator).

Fragment from [examples/c++/10-composite-key.c++](#) — packing two fields into a composite key with big-endian byte order: the lexicographic order of the key matches the numeric order of the pair:

```
// Packing (x, y) into a uint64_t so that the byte-wise (lexicographic) order
// of the key matches the numeric order of the pair. For that, a big-endian
// representation of unsigned fixed-width fields is used.
uint64_t pack(uint32_t x, uint32_t y) {
    const uint64_t value = (uint64_t(x) << 32) | uint64_t(y);
    #if defined(__BYTE_ORDER__) && __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
        return __builtin_bswap64(value);
    #else
        return value;
    #endif
}
```

Full code: [10-composite-key.c++](#)

4.3.5 8.5. Deleting by index

You cannot delete via an index directly — find the primary key via the index, then delete the record and update the indexes. Use the safe iteration pattern (chapter 6).

4.3.6 8.6. Mini-ORM over libmdbx (end-to-end project)

We extend the configurator: add a "users" entity with an index by name.

```
/* schema: users(id -> name), idx_name(name -> id) */
int user_add(config_store_t *cs, uint64_t id, const char *name) {
    MDBX_txn *txn; MDBX_dbi users, idx;
    MDBX_val k, v;
    int rc;

    rc = mdbx_txn_begin(cs->env, NULL, MDBX_TXN_READWRITE, &txn);
    if (rc) return rc;
    mdbx_dbi_open(txn, "users", MDBX_CREATE | MDBX_INTEGERKEY, &users);
    mdbx_dbi_open(txn, "idx_name", MDBX_CREATE | MDBX_DUPSORT, &idx);

    k.iov_base = &id; k.iov_len = sizeof(id);
    v.iov_base = (void *)name; v.iov_len = strlen(name);
    rc = mdbx_put(txn, users, &k, &v, MDBX_NOOVERWRITE);
    if (rc == MDBX_KEYEXIST) { mdbx_txn_abort(txn); return rc; }

    /* index: name -> id */
    k.iov_base = (void *)name; k.iov_len = strlen(name);
    v.iov_base = &id; v.iov_len = sizeof(id);
    rc = mdbx_put(txn, idx, &k, &v, MDBX_NODUPDATA);

    rc = mdbx_txn_commit(txn);
    return rc;
}
```

Nuance: atomicity guarantees exactly "all or nothing": if a crash happens between operations, the transaction rolls back entirely. This is the price of correct indexes — update them only in one transaction together with the main record.

Examples for this chapter: [examples/c++/09-secondary-index.c++](#); [examples/c++/10-composite-key.c++](#); end-to-end project: [config-store-08.c++](#).

4.3.7 8.7. Summary of chapter 8

- A secondary index is a "field → primary key" table that you maintain yourself.
- All changes (data + indexes) go in one transaction.
- A DUPSORT index for "one-to-many"; composite keys for several fields.
- Delete a record together with its index entries; otherwise the index goes stale.
- An inverted index over DUPSORT is a cheap alternative to a separate link table.

4.3.8 8.8. Exercises

1. Add an index "by city" and write a lookup function.
2. Write `user_rename`, which atomically changes the name and rebuilds `idx_name`.
3. Why do composite keys require care with byte order?

4.3.9 8.9. Chapter 8 checklist

- [] I understand that a secondary index is a "field → primary key" table that I maintain myself;
- [] I update the data and all indexes in one transaction, remembering the price of breaking this rule;
- [] I choose deliberately between a DUPSORT index ("field → list of IDs") and a separate table;
- [] I know why numeric fields in composite keys are packed big-endian;
- [] I delete a record together with its index entries; otherwise the index goes stale;
- [] I remember `MDBX_DB_ACCEDE` for the case when a table's flags are unknown in advance.

4.3.10 8.10. What's next

Indexes and data lead to the next question: how to configure the environment itself. Chapter 9 is about file geometry (`mdbx_env_set_geometry()`), the `maxreaders / maxdbs` limits, environment flags, `MDBX_opt_*` runtime options, and statistics. A configurator that grows from a test database to a working one needs a managed file size and predictable limits — otherwise `MDBX_MAP_FULL` arrives sooner or later.

4.4 Chapter 9. Environment configuration

4.4.1 9.1. File geometry

Geometry determines how the database file grows and shrinks:

```
int mdbx_env_set_geometry(MDBX_env *env,
    intptr_t size_lower, intptr_t size_now, intptr_t size_upper,
    intptr_t growth_step, intptr_t shrink_threshold, unsigned pagesize);
```

Parameters (`-1` = "default/do not change"):

Parameter	Meaning
<code>size_lower</code>	Lower bound of the file size (do not shrink below)
<code>size_now</code>	Initial size at creation
<code>size_upper</code>	Hard limit ; on reaching it with no space left — <code>MDBX_MAP_FULL</code>
<code>growth_step</code>	File growth step
<code>shrink_threshold</code>	Threshold at which the file may be truncated
<code>pagesize</code>	Page size (256...65536; set before the first open)

Warning: set geometry **once, before** `env_open` (or at creation). Changing `upper` "on the fly" is not reliably possible. The engine picks the `upper` value for a new database itself (~golden section of RAM, bounded by the mmap limit ≈ 140 TB on 64-bit); `TOO_LARGE` / `ENOMEM` are possible with an obviously excessive `upper` (e.g. under ASAN/Valgrind) — always set a reasonable `upper` explicitly.

Historical note. `mdbx_env_set_mapsize(env, size)` is a legacy wrapper over `mdbx_env_set_geometry()`, equivalent to `mdbx_env_set_geometry(env, size, size, size, -1, -1, -1)`. Like geometry, it resizes the database "on the fly" (including after `mdbx_env_open()`), but pins the lower/current/upper bounds to a single value — the database loses the ability to auto-grow above `size`. Prefer `mdbx_env_set_geometry()` with separate parameters. Likewise `mdbx_env_sync()` is a legacy synonym for `mdbx_env_sync_ex(env, force=true, nonblock=false)` — a blocking full synchronization (chapter 10).

4.4.2 9.2. maxreaders / maxdbs / pagesize

```
mdbx_env_set_maxreaders(env, readers); /* reader-table slots, before open */
mdbx_env_set_maxdbs(env, dbs);      /* limit of named tables */
/* the page size is set via the pagesize argument of mdbx_env_set_geometry() — there is no separate function */
```

Current values are read back with the getter pairs `mdbx_env_get_maxreaders()` / `mdbx_env_get_maxdbs()`.

`maxreaders` — how many threads can hold read transactions at the same time. Do not set it too low in applications with thread pools.

4.4.3 9.3. Environment flags

```
mdbx_env_set_flags(env, flags, onoff);
mdbx_env_get_flags(env, &flags);
```

Key flags: `MDBX_RDONLY`, `MDBX_WRITEMAP`, `MDBX_NOMETASYNC`, `MDBX_SAFE_NOSYNC`, `MDBX_UTTERLY_NOSYNC`, `MDBX_NOSTICKYTHREADS`, `MDBX_EXCLUSIVE`.

Two more important flags are passed **at** `mdbx_env_open()`, not via `set_flags`:

- `MDBX_ACCEDE` — open a database **already used by another process in an unknown mode**: instead of an `MDBX_INCOMPATIBLE` error the environment opens in a mode compatible with the current usage (applies to the durability flags, `MDBX_LIFORECLAIM` and `MDBX_NORDAHEAD`). Has no effect if the current process is the only one or all opens are read-only.
- `MDBX_EXCLUSIVE` — exclusive open: succeeds only if the database is not opened by anyone else.

A couple of useful environment functions: `mdbx_env_warmup()` — "warm up" the database file into RAM (the page cache) before starting work; `mdbx_env_get_fd()` — the data file descriptor (for your own fsync/backup purposes).

4.4.4 9.4. Runtime options (MDBX_opt_*)

Fine tuning via `mdbx_env_set_option / mdbx_env_get_option` :

Option	Meaning
<code>MDBX_opt_rp_augment_limit</code>	Limit of list accumulation when searching sequences in the GC
<code>MDBX_opt_gc_time_limit</code>	Time limit for GC search inside a write transaction (1/65536 s)
<code>MDBX_opt_txn_dp_limit</code>	Dirty-page limit of a transaction (by default $\approx 1/42$ of RAM)
<code>MDBX_opt_loose_limit</code>	Loose-pages cache (default 64)
<code>MDBX_opt_writethrough_threshold</code>	Threshold for choosing <code>0_DSYNC</code> vs <code>fdatsync</code> (default 2 dirty pages; ignored on Windows)
<code>MDBX_opt_prefault_write_enable</code>	Pre-fault writing for WRITEMAP
<code>MDBX_opt_sync_bytes</code> / <code>MDBX_opt_sync_period</code>	Auto-synchronization (defaults: ~ 1.05 GB / 42.42 s; explicit 0 = disabled)
<code>MDBX_opt_merge_threshold</code>	Page merge threshold (16.16 format; default 33%, range [12.5%..50%])
<code>MDBX_opt_prefer_waf_insteadof_balance</code>	Prefer the dirty neighbor when merging (default true since 2026-01-04)

Fragment from `examples/c++/11-geometry-options.c++` — setting geometry before `open` (via `create_parameters().set_geometry(geo)`) and runtime options after:

```
// Geometry: lower/current/upper bounds of the DB size, the growth step,
// and the shrink threshold. Here — a compact database for the example.
mdbx::env::geometry geo;
geo.size_lower = 8 * mdbx::env::geometry::MB;
geo.size_now = 16 * mdbx::env::geometry::MB;
geo.size_upper = 64 * mdbx::env::geometry::MB;
geo.growth_step = 8 * mdbx::env::geometry::MB;
geo.shrink_threshold = 2 * mdbx::env::geometry::MB;

mdbx::env_managed env(path, mdbx::env_managed::create_parameters().set_geometry(geo),
                      mdbx::env::operate_parameters());

// Runtime options: set and read back.
env.set_extra_option(mdbx::env::extra_runtime_option::writethrough_threshold, 1 << 20);
env.set_extra_option(mdbx::env::extra_runtime_option::merge_threshold_dot16, 65536 / 3);
env.set_sync_threshold(256 * 1024); // sync_bytes = 256 KiB
env.set_extra_option(mdbx::env::extra_runtime_option::prefault_write_enable, 1);
```

Full code: [11-geometry-options.c++](#) · [C version](#)

4.4.5 9.5. Choosing the page size

Page	When
4 KB	Default; universal
8 KB	Insurance against a large fragmented GC
64 KB	Large values (fewer overflow trips); more expensive small writes

Note: a value that does not fit into a single page is stored as one unbroken piece on consecutive pages and needs a run of **consecutive** free pages — allocating it is costlier and, under fragmentation, may end in `MDBX_MAP_FULL` . So when choosing the page size, keep the actual (typical and maximum) value sizes in mind: the larger the share of values that do not fit into one page, the more attractive a large page. If a long value cannot be made smaller, split it into chunks that fit into one page (`mdbx_limits_valsize4page_max()`); manual splitting is needed only until the planned **streamed API for long values** ("Streaming BLOBs") arrives. Details — [Restrictions & gotchas](#).

4.4.6 9.6. Statistics

```
mdbx_env_info_ex(env, &info, sizeof(info)); /* geometry, metas, counters */
mdbx_env_stat_ex(env, txn, &stat, sizeof(stat)); /* sizes, pages */
mdbx_dbi_stat_ex(txn, dbi, &dst, sizeof(dst), 0); /* table statistics */
```

Fragment from `examples/c++/12-env-stat.c++` — reading environment statistics (the analogues of `mbdx_env_stat_ex()` / `mbdx_env_info_ex()`):

```
{
    auto txn = env.start_write();
    auto table = txn.open_map(nullptr);
    for (int i = 0; i < 100; ++i) {
        const auto key = std::to_string(i);
        const auto value = "value-" + std::to_string(i);
        txn.insert(table, mdbx::slice(key), mdbx::slice(value));
    }
    txn.commit();
}

const auto stat = env.get_stat(); // analogue of mdbx_env_stat_ex()
std::cout << "stat: ps=" << stat.ms_psize << " depth=" << stat.ms_depth
    << " leaf=" << stat.ms_leaf_pages << " branch=" << stat.ms_branch_pages
    << " overflow=" << stat.ms_overflow_pages << " entries=" << stat.ms_entries << "\n";

const auto info = env.get_info(); // analogue of mdbx_env_info_ex()
std::cout << "info: recent_txnid=" << info.mi_recent_txnid
    << " latter_reader_txnid=" << info.mi_latter_reader_txnid << " geo.current=" << info.mi_geo.current
    << "\n";
```

Full code: `12-env-stat.c++` · [C version](#)

Examples for this chapter: `examples/c++/11-geometry-options.c++` · [C version](#); `examples/c++/12-env-stat.c++` · [C version](#); end-to-end project: `config-store-09.c++`.

4.4.7 9.7. Summary of chapter 9

- Geometry (lower/now/upper/growth/shrink/pagesize) is set before open.
- `upper` — a hard limit; too low → `MDBX_MAP_FULL`, obviously excessive (e.g. under ASAN/Valgrind) → `TOO_LARGE` / `ENOMEM`. The engine picks the default for a new database itself (≈ golden section of RAM).
- `maxreaders` / `maxdbs` / `pagesize` — before the first open.
- Options `MDBX_opt_*` — fine tuning of GC/spill/synchronization.

4.4.8 9.8. Exercises

1. Set the geometry "from 1 MB to 4 GB, step 64 MB" before open and check `env_info`.
2. What does `env_open` return with an obviously excessive `upper` (e.g. 140 TB on 64-bit)? Record the error code.
3. Create a database with a 64 KB page and compare with 4 KB when reading large values.

4.4.9 9.9. Chapter 9 checklist

- [] I set the geometry (`size_lower` / `size_now` / `size_upper` / `growth_step` / `shrink_threshold`) before `env_open`;
- [] I understand that `size_upper` is a hard limit: exhausting it yields `MDBX_MAP_FULL`, while an excessive `upper` yields `TOO_LARGE` / `ENOMEM`;
- [] I know where `maxreaders`, `maxdbs`, and the page size are set;
- [] I distinguish the key environment flags (`MDBX_WRITEMAP`, `MDBX_SAFE_NOSYNC`, `MDBX_EXCLUSIVE`, `MDBX_ACCEDE`) and the main `MDBX_opt_*` options;
- [] I can take statistics via `mbdx_env_info_ex()` / `mbdx_env_stat_ex()` and use it for diagnostics.

4.4.10 9.10. What's next

A configured environment is not the whole story: you must also decide what happens on commit. Chapter 10 covers durability modes: weak/steady meta, what `MDBX_NOMETASYNC` and `MDBX_SAFE_NOSYNC` risk, why auto-synchronization is needed, and how the modes differ on macOS, Linux, and Windows. The mode choice directly determines the configurator's speed and reliability.

4.5 Chapter 10. Durability modes

4.5.1 10.1. What "committing" means

A strict commit must: (1) write data to disk (new page versions) and (2) write the meta — make the snapshot "visible" for future opens. Between these events we distinguish:

- **weak meta** — data is in the file but not guaranteed on durable storage;
- **steady meta** — data flushed to disk; the snapshot survives a system crash.

4.5.2 10.2. Modes

Mode	Behavior	Risk
<code>MDBX_SYNC_DURABLE</code> (default)	Data → flush → meta → flush	None: full ACID
<code>MDBX_NOMETASYNC</code>	Data flushed, meta — deferred	Loss of recent commits on a crash
<code>MDBX_SAFE_NOSYNC</code>	Nothing flushed immediately, the previous steady is kept	Rollback to the last steady; file growth (quasi-long-reader effect)
<code>MDBX_UTTERLY_NOSYNC</code>	No flushes, no steady guarantees	The database may not survive a crash
<code>MDBX_WRITEMAP</code>	Writing via mmap (+msync)	Combines with the modes above

Fragment from `examples/c++/13-sync-modes.c++` — changing the sync mode between opens of the same database: full durability (`robust_synchronous`) → `NOMETASYNC` (`half_synchronous_weak_last`):

```
// Run 1: full durability (sync before the commit returns).
{
    auto env = example::env_open(path, mdbx::env::operate_parameters().robust_synchronous());
    auto txn = env.start_write();
    auto table = txn.open_map(nullptr);
    txn.insert(table, mdbx::slice("key"), mdbx::slice("v1"));
    txn.commit();
    env.sync_to_disk(true); // explicit flush to disk
    std::cout << "durable: ok\n";
}

// Run 2: NOMETASYNC (metadata is not synchronized on every commit).
{
    auto env = example::env_open(path, mdbx::env::operate_parameters().half_synchronous_weak_last());
    auto txn = env.start_read();
    auto table = txn.open_map(nullptr);
    std::cout << "nometasync: " << txn.get(table, mdbx::slice("key")).as_string() << "\n";
    txn.abort();
}
```

Full code: [13-sync-modes.c++ · C version](#)

4.5.3 10.3. Which mode when

- **Default** — maximum reliability (ACID). For databases where losing a commit is unacceptable.
- **`MDBX_NOMETASYNC`** — high-load writing where the "tail" of the latest commits may be lost.
- **`MDBX_SAFE_NOSYNC`** — writing can speed up several times (up to 10×), but: the file grows (pages newer than steady are not reused until a new steady-point), on a crash — rollback to the last steady.
- **`MDBX_UTTERLY_NOSYNC`** — only for non-critical data/caches.
- **`MDBX_WRITEMAP`** — for large transactions and fine-grained write tuning; combines with the flags above (e.g., `SAFE_NOSYNC | WRITEMAP`).

4.5.4 10.4. Auto-synchronization

`MDBX_SAFE_NOSYNC` without growth management is a source of uncontrolled file growth. Manage it:

```
mdbx_env_set_syncbytes(env, 512ULL * 1024 * 1024); /* flush after 512 MB of writes */
mdbx_env_set_syncperiod(env, 3 << 16); /* 3 seconds in 16.16 format: 3<<16 = 3·65536 units (raw value 3 ≈ 46 μs) */
```

And manually from a separate thread: `mdbx_env_sync_ex(env, force, nonblock)` — with `nonblock=true` the call returns immediately (asynchronously), completion is checked with the cheap `mdbx_env_sync_poll(env)`.

4.5.5 10.5. Platform-specific notes

- **macOS:** by default `fcntl(F_FULLFSYNC)` — maximum durability but slow; `MDBX_APPLE_SPEED_INSTEADOF_DURABILITY` switches the priority to speed.
- **Linux:** `boot_id` is used to control the rollback of weak metas (important in LXC).
- **Windows:** `LockFileEx` file locks are slower than named mutexes (affects small transactions); `MDBX_WRITE_MAP` helps. Write-through is always used; `MDBX_NOMETASYNC` switches to "lazy + flush" (the writethrough threshold is ignored).

4.5.6 10.6. Example: switching the mode

Fragment (C, illustration):

```
MDBX_env *env;
mdbx_env_create(&env);
mdbx_env_set_geometry(env, -1, -1, 8LL * 1024 * 1024 * 1024, -1, -1, -1);
mdbx_env_set_flags(env, MDBX_SAFE_NOSYNC, 1); /* enable */
mdbx_env_set_syncbytes(env, 256ULL * 1024 * 1024);
mdbx_env_open(env, "./safe.mdbx", MDBX_NOSUBDIR, 0664);
```

Examples for this chapter: [examples/c++/13-sync-modes.c++](#) · [C version](#); end-to-end project: [config-store-10.c++](#).

4.5.7 10.7. Summary of chapter 10

- Durability modes differ in what gets flushed: data, meta, nothing.
- `SAFE_NOSYNC` — speed at the cost of file growth and rollback to steady on a crash.
- Auto-synchronization (`syncbytes` / `syncperiod`) is mandatory for `SAFE_NOSYNC`.
- Platform differences (`F_FULLFSYNC`, `boot_id`, `LockFileEx`) affect the choice.

4.5.8 10.8. Exercises

1. Compare commit speed in `DURABLE` vs `SAFE_NOSYNC` on 10,000 small transactions.
2. Observe file growth in `SAFE_NOSYNC` without auto-sync — and after configuring `syncbytes`.
3. Explain why `UTTERLY_NOSYNC` is "maximum risk": what exactly may not survive a crash.

4.5.9 10.9. Chapter 10 checklist

- [] I understand that a strict commit is "data to disk + meta to disk" and distinguish weak/steady meta;
- [] I know the risks of each mode: `MDBX_SYNC_DURABLE`, `MDBX_NOMETASYNC`, `MDBX_SAFE_NOSYNC`, `MDBX_UTTERLY_NOSYNC`, and combinations with `MDBX_WRITE_MAP`;
- [] for `MDBX_SAFE_NOSYNC` I configure auto-synchronization (`syncbytes` / `syncperiod`) to keep file growth under control;
- [] I can switch modes and call `mdbx_env_sync_ex()` manually;
- [] I remember the platform nuances: `F_FULLFSYNC` on macOS, `boot_id` on Linux, `LockFileEx` on Windows.

4.5.10 10.10. What's next

Durability modes set the rules for writing; the next chapter is about working with the database from several threads. Chapter 11 covers sticky threads, the `MDBX_NOSTICKYTHREADS` flag and its pitfalls, TLS reader slots, `fork()`, transaction cloning, and parking of long-lived readers. A configurator with a thread pool or coroutines cannot do without this knowledge.

4.6 Chapter 11. Multithreading

4.6.1 11.1. Thread model: sticky threads by default

By default a transaction "sticks" to the thread that created it. Other threads cannot use this object — you get `MDBX_THREAD_MISMATCH`. Readers (different transactions in different threads) work in parallel without locks.

Fragment from `examples/c++/14-multithreading.cpp` — a writer commits new records, while readers on clones of one snapshot (`base.clone()`) keep seeing a stable picture:

```
std::atomic<bool> writer_done{false};
std::thread writer([&] {
    for (int batch = 0; batch < 2; ++batch) {
        auto txn = env.start_write();
        auto table = txn.open_map(nullptr);
        for (int i = 0; i < 10; ++i) {
            const int k = 10 + batch * 10 + i;
            txn.insert(table, mdbx::slice(std::to_string(k)), mdbx::slice("w"));
        }
        txn.commit();
        std::this_thread::yield();
    }
    writer_done = true;
});

// Two readers: each works with its own clone of the base snapshot.
std::atomic<size_t> snapshot_count[2];
std::atomic<size_t> fresh_count[2];
std::vector<std::thread> readers;
for (int r = 0; r < 2; ++r) {
    readers.emplace_back([&, r] {
        auto clone = base.clone(); // a snapshot clone for this thread
        auto table = clone.open_map(nullptr);
        snapshot_count[r] = clone.get_map_stat(table).ms_entries;
        clone.abort();
    });
}
```

Full code: [14-multithreading.cpp](#)

4.6.2 11.2. MDBX_NOSTICKYTHREADS

The `MDBX_NOSTICKYTHREADS` flag allows transferring a transaction between threads — needed for thread pools, coroutines, async runtimes (tokio and the like), where an operation may continue in another thread.

Warning: with `NOSTICKYTHREADS`, functions requiring the environment write lock (`mdbx_env_set_option`, `set_flags`, `set_geometry`, `sync`, `stat`, `defrag`, `close`), may deadlock if a write transaction "moved" to another thread that waits for that lock. Serialize such calls (channel/mutex), or keep them in the thread that owns the transaction.

4.6.3 11.3. Thread registration and TLS

A reader slot is bound to a thread via TLS. On proper thread exit the destructor cleans up the slot. If a thread exits "abnormally" (or the TLS destructor did not run — known glibc bugs #21031/#21032), the slot may "leak". For threads without TLS — explicit registration:

```
mdbx_thread_register(env);
/* ... */
mdbx_thread_unregister(env);
```

Leak diagnostics — `mdbx_reader_check(env, &dead)` (Volume V, chapter 30).

4.6.4 11.4. fork()

After `fork()` the child **does not inherit** `mmap` and record locks. Using the environment in the child process is possible only after:

```
mdbx_env_resurrect_after_fork(env);
```

Called once in the child, not in the parent.

4.6.5 11.5. Cloning transactions

`mdbx_txn_clone()` duplicates a read-only transaction — several handlers on one snapshot without rescanning the RLT (the RLT — reader lock table — is libmdbx's internal registry of readers; detailed in Volume III, ch. 14) — useful for parallel processing.

4.6.6 11.6. Parking and eviction

A long-lived reader "pins" the **detent** (the oldest active snapshot; details — Volume III, chapter 14) and prevents page recycling. Solutions:

```
mdbx_txn_park(txn); /* release the reader slot while keeping the handle */
mdbx_txn_unpark(txn); /* resume (when evicted — with restart_if_ousted=true) */
```

If the writer lacks space, parked readers are **ousted**: the slot moves from `PARKED` to `OUSTED`; on the next access the reader either restarts on a fresh snapshot, or gets `MDBX_OUSTED`.

Warning: after parking the snapshot is not held — dereferencing pointers obtained before parking is forbidden (pages may be reused).

4.6.7 11.7. Several environments; `MDBX_EXCLUSIVE`

- You cannot open the same database twice in one process (protection against races; legacy mode — `MDBX_DBG_LEGACY_MULTIOPEN`).
- Several different databases (different files) in one process — allowed.
- `MDBX_EXCLUSIVE` — exclusive open (only this process).

Examples for this chapter: [examples/c++/14-multithreading.c++](#); [examples/c++/15-fork-resurrect.c++](#); [examples/c++/16-parking.c++](#); end-to-end project: [config-store-11.c++](#).

4.6.8 11.8. Summary of chapter 11

- Sticky threads by default; `NOSTICKYTHREADS` for pools/coroutines, but with a deadlock risk in write functions.
- TLS destructors clean reader slots; on abnormal exits — `reader_check` / registration.
- `resurrect_after_fork` is mandatory in the child.
- Parking/eviction — the solution for long reads.

4.6.9 11.9. Exercises

1. Write an example with two threads: one writes, the other reads. Verify the reader is not blocked.
2. Reproduce the scenario: a write transaction started in thread A, thread B calls `env_sync` with `NOSTICKYTHREADS` — capture the deadlock.
3. Think of when `txn_clone` saves resources compared to several `txn_begin`.

4.6.10 11.10. Chapter 11 checklist

- [] I understand that a transaction is bound to its thread by default and recognize `MDBX_THREAD_MISMATCH`;
- [] I know when `MDBX_NOSTICKYTHREADS` is justified and which write functions may deadlock with it;
- [] I understand how TLS reader slots are cleaned up and how `mdbx_thread_register()` / `mdbx_thread_unregister()` help;
- [] after `fork()` I call `mdbx_env_resurrect_after_fork()` in the child, not in the parent;
- [] I can clone read-only transactions (`mdbx_txn_clone()`) for parallel processing of one snapshot;

- [] I use `mdbx_txn_park()` / `mdbx_txn_unpark()` for long-lived readers and remember that a parked snapshot is not held.

4.6.11 11.11. What's next

One practical skill remains: reacting to errors. Chapter 12 systematizes the return codes: expected states (`MDBX_KEYEXIST` , `MDBX_NOTFOUND`), space problems (`MDBX_MAP_FULL` vs `MDBX_TXN_FULL`), corruption and discipline-violation errors, and retry loops for concurrent scenarios. After it, the configurator will withstand both crashes and contention.

4.7 Chapter 12. Error handling

4.7.1 12.1. Return codes

- `MDBX_SUCCESS` (0) — success.
- Negative `MDBX_*` — errors (`MDBX_NOTFOUND` , `MDBX_KEYEXIST` , `MDBX_BAD_DBI` , ...).
- Special results: `MDBX_RESULT_TRUE` (operation performed), `MDBX_RESULT_FALSE` (no data/not required).
- System codes (errno-like) may come from the OS — they are negative and reported as-is.

4.7.2 12.2. `mdbx_strerror_r` vs `mdbx_strerror`

`mdbx_strerror(rc)` uses a shared buffer — **not thread-safe**. In multithreaded code use:

```
char buf[128];
mdbx_strerror_r(rc, buf, sizeof(buf));
```

4.7.3 12.3. `MDBX_MAP_FULL`: what to do

`MDBX_MAP_FULL` — the database hit `upper` , and there is nothing to reuse (the GC is empty/frozen). The sequence:

1. **Abort the current write transaction** (continuing after a geometry change → `MDBX_BAD_TXN`).
2. Check long readers (`mdbx_stat -r` retained; `mdbx_env_info_ex`).
3. Set/raise `upper` **before creating/at open**.
4. Install an HSR callback (HSR — Handle-Slow-Readers, the mechanism for evicting stuck readers; Volume V, chapter 29).

4.7.4 12.4. `MAP_FULL` vs `TXN_FULL`

- `MDBX_MAP_FULL` — the file space is exhausted (geometry).
- `MDBX_TXN_FULL` — the internal limit of the transaction's dirty/retired pages is exhausted. The cure — commit earlier, split transactions, check `mdbx_txn_info()` .

4.7.5 12.5. Expected codes

- `MDBX_KEYEXIST` — the key already exists (with `NOOVERWRITE`).
- `MDBX_NOTFOUND` — there is no key/value. These are normal states, not errors: handle them with branching, not panic.

4.7.6 12.6. Corruption and recovery errors

- `MDBX_BAD_TXN` , `MDBX_BAD_DBI` — an invalid handle (discipline violation/closed transaction).
- `MDBX_EBADSIGN` — corrupted signature.
- `MDBX_WANNA_RECOVERY` — the database requires recovery at read-only open; open read-write or use `mdbx_env_open_for_recovery()` .
- `MDBX_MVCC_RETARDED` — a reader older than the actual snapshot.
- `MDBX_LAGGARD_READER` — a reader lagged behind (see HSR).

4.7.7 12.7. Discipline violations

- `MDBX_THREAD_MISMATCH` — the transaction is used from a foreign thread.
- `MDBX_TXN_OVERLAPPING` — overlapping read/write transactions in one thread.
- `MDBX_BUSY` — an ownership conflict.
- `MDBX_BAD_RSLT` — an invalid reader slot. These are signals of architectural errors, not "random glitches".

4.7.8 12.8. Retry loop strategies

For concurrent scenarios ("compare-and-replace"):

Fragment (C, illustration):

```
for (;;) {
    mdbx_txn_begin(env, NULL, MDBX_TXN_READWRITE, &txn);
    /* read the condition */
    rc = mdbx_get(txn, dbi, &key, &val);
    if (rc) { mdbx_txn_abort(txn); return rc; }
    if (!condition_met(val)) { mdbx_txn_abort(txn); return MDBX_RESULT_FALSE; }
    /* write the new value */
    rc = mdbx_put(txn, dbi, &key, &newval, 0);
    if (rc) { mdbx_txn_abort(txn); return rc; }
    rc = mdbx_txn_commit(txn);
    if (rc == MDBX_MAP_FULL) { /* raise upper/wait for readers */ continue; }
    return rc;
}
```

Fragment from `examples/c++/17-error-handling.cpp` — a retry loop on environment "busyness" (`MDBX_BUSY` under process contention / `EAGAIN` inside the process): opening in exclusive mode while the database is already open is repeated a bounded number of times:

```
// Retry strategy: attempting to open the environment in exclusive mode while
// it is already open by the current process yields "busyness" (MDBX_BUSY under
// process contention or the system EAGAIN inside the process).
// Such a failure is caught and handled with a bounded retry.
const int max_attempts = 3;
int attempt = 0;
for (; attempt < max_attempts; ++attempt) {
    try {
        mdbx::env_managed busy(path, mdbx::env::operate_parameters().exclusive());
        std::cerr << "FAIL: exclusive open unexpectedly succeeded\n";
        return EXIT_FAILURE;
    } catch (const std::exception &ex) {
        std::cout << "busy attempt " << (attempt + 1) << ": failed (" << ex.what() << ")\n";
    }
}
std::cout << "busy: persists after " << attempt << " attempts (expected while env is open)\n";
```

Full code: [17-error-handling.cpp](#) · C version

Examples for this chapter: [examples/c++/17-error-handling.cpp](#) · C version; end-to-end project: [config-store-12.cpp](#).

4.7.9 12.9. Summary of chapter 12

- Success = 0; errors are negative; `RESULT_TRUE/FALSE` — special results.
- `mdbx_strerror_r` — the thread-safe variant.
- `MAP_FULL` is cured with geometry+HSR; `TXN_FULL` — with early commits.
- `KEYEXIST` / `NOTFOUND` — expected states.
- Discipline codes (`THREAD_MISMATCH` and others) indicate architectural errors.

4.7.10 12.10. Exercises

1. Write a `MDBX_KEYEXIST` handler in the end-to-end project: on conflict — read and decide.
2. Describe how `MAP_FULL` differs from `TXN_FULL`, with examples.
3. Make a retry loop around `cfg_set` with a "read-check-write" strategy.

4.7.11 12.11. Chapter 12 checklist

- [] I read return codes correctly: 0 — success, negative — errors, `MDBX_RESULT_TRUE` / `MDBX_RESULT_FALSE` — special results;
- [] in multithreaded code I use `mdbx_strerror_r()` rather than `mdbx_strerror()` ;
- [] I know the sequence of actions for `MDBX_MAP_FULL` (abort, check readers, raise `upper` before opening, HSR) and can tell it apart from `MDBX_TXN_FULL` ;
- [] I handle `MDBX_KEYEXIST` and `MDBX_NOTFOUND` with branching, not panic;
- [] I treat discipline codes (`MDBX_THREAD_MISMATCH` , `MDBX_TXN_OVERLAPPING`) as signals of architectural mistakes;
- [] I can write a "read-check-write" retry loop for concurrent scenarios.

4.7.12 12.12. What's next

Volume II provided the full practical toolkit, but many of its rules ("close read transactions", "geometry before open", "`SAFE_NOSYNC` grows the file") looked like demands without explanations. Volume III opens libmdbx's internal mechanisms: B+tree and mmap, MVCC, the commit pipeline, the free-page GC, geometry, and crash recovery. With them, the rules turn into an understanding of "why".

4.8 Volume summary

Volume II gave you a practical toolkit: cursors, DUPSORT and indexes, environment configuration, durability modes, multithreading, and error handling. The end-to-end configurator project is now a full-fledged application with indexes.

What's next: Volume III — internal mechanisms: B+tree and mmap, MVCC, the commit pipeline, GC, geometry, nested transactions, the lock file, durability and recovery.

5. Volume III. Internal Mechanisms

Level: for those who understand architecture; a bridge between "how to use" and "how to configure and debug". **Goal of the volume:** you explain libmdbx's behavior through its internals: B+tree and mmap, MVCC, commit pipeline, GC, geometry, nested transactions, lock file, and recovery.

5.1 Chapter 13. Storage Architecture: B+tree and mmap

5.1.1 13.1. B+tree: Branches, Leaves, Overflow

libmdbx stores data in a **B+tree** — a balanced tree where all values live in leaves, while internal nodes (branch) contain only separator keys and pointers to child pages.



Properties:

- search — descent from root to leaf, $O(\log N)$;
- all leaves at the same level are linked — range traversal goes through neighbors;
- on leaf overflow — **split** (halving), recursively upward, including a new root;
- on emptying below the threshold — **rebalance/merge** with a neighbor (threshold `merge_threshold`).

Values that do not fit in a leaf (larger than roughly half a page) go to **overflow pages**: a contiguous run of `P_LARGE` pages; the leaf keeps a pointer `{pgno, npages}`.

When the value does not fit even a single overflow page (`npages > 1`), the run must be a sequence of **consecutive** pages of the needed length. This is the main practical point: allocating such a run is costlier than allocating any single free page (the engine may scan and merge GC records or append at the file end), and rewriting the value rewrites the whole run (CoW of all its pages, fragmentation, GC churn). The problem can be softened by splitting the long value into chunks that fit into a single page (`mdbx_limits_valsize4page_max()`); manual splitting is needed only until the planned **streamed API for long values** ("Streaming BLOBs") arrives. Details and mitigations — [Restrictions & gotchas](#).

5.1.2 13.2. Why B+tree Rather than B-tree or LSM

- **B-tree** — data in both leaves and branches: a search may stop at any level, but traversal requires moving up/down. A B+tree with data only in leaves gives a predictable descent and linear traversal.
- **LSM** (LevelDB/RocksDB) is optimized for writes (append-only + background compaction), but reads suffer from multi-level lookups. A B+tree gives honest $O(\log N)$ for both reads and writes without background processes.

5.1.3 13.3. Page — the Unit of Storage and I/O

The page size is 256...65536 bytes (4096 by default), chosen when the database is created and then fixed. Types:

Type	Purpose
branch	Internal node: separators and pointers
leaf	Leaf: key→value pairs
large / overflow	Contiguous run of large values
meta	One of the three meta pages (database state snapshot)
dupfix / subpage	Dense packing of multivalues

Every page carries the **transaction label** (`txnid`) of the transaction that created its current version — this is the basis of CoW and MVCC.

A user-supplied **canary** (`MDBX_canary`, 32 bytes — four `uint64_t`: `x`, `y`, `z`, `v`; the `v` field is always set to the transaction number) can be associated with the database: `mdbx_canary_put(txn, &canary)` writes it into the meta, `mdbx_canary_get(txn, &canary)` reads it. Updated values become visible to other processes only after the commit. This is not table data but an arbitrary application marker (e.g., a schema version) that survives database opens and is available even when mounted read-only.

5.1.4 13.4. mmap: Virtual Memory as a Window into the File

The file is mapped into the address space in its entirety. Reading is a pointer dereference; the kernel (page cache) pulls in the page. There is no separate buffer cache in the library.

Consequences:

- copy-free reads (you get an address straight into mmap);
- the OS page cache manages residency;
- **database size is limited by the address space** (on 32-bit — practically ~1–3 GB);
- when the database $\gg$ RAM, PTE overhead in the kernel grows.

`madvise(MADV_NOHUGEPAGE)` (opting out of THP) is applied to the mmap region — huge pages interfere with accurate residency estimation.

Fragment from `examples/c++/18-tree-height.c++` — tree estimation via `mdbx_env_stat_ex()`: height, leaf/branch/overflow pages and the number of entries; a value larger than the page size creates overflow pages:

```
{
    auto txn = env.start_write();
    auto table = txn.open_map(nullptr);
    for (int i = 0; i < 5000; ++i)
        txn.insert(table, mdbx::slice("k" + std::to_string(i)), mdbx::slice("v"));
    // A value larger than the page size (4096) goes to overflow pages.
    const std::string big(64 * 1024, 'x');
    txn.insert(table, mdbx::slice("big"), mdbx::slice(big));
    txn.commit();
}

const auto stat = env.get_stat(); // mdbx_env_stat_ex()
std::cout << "stat: depth=" << stat.ms_depth << " leaf=" << stat.ms_leaf_pages
          << " branch=" << stat.ms_branch_pages << " overflow=" << stat.ms_overflow_pages
          << " entries=" << stat.ms_entries << "\n";
```

Full code: [18-tree-height.c++](#) · [C version](#).

5.1.5 13.5. The Reachability Invariant

Every non-meta page up to the `first_unallocated` pointer is reachable **exactly once**: either it belongs to some tree (an ordinary, the main, or the GC table), or it is listed in a GC record. Integrity auditing and the database verification tool (`mdbx_chk`) are built on this invariant.

Examples for this chapter: [examples/c++/18-tree-height.c++](#) · [C version](#).

5.1.6 13.6. Summary of Chapter 13

- B+tree: data in leaves, branches — separators; split/merge maintains the balance.
- Large values — overflow runs of pages.
- mmap instead of a buffer cache: copy-free reads, the cost — address space/PTE.
- The reachability invariant — the foundation of integrity.

5.1.7 13.7. Exercises

1. Explain why `mdbx_get` on 64-bit is practically independent of the database size.
2. What changes for a reader if the tree grows from height 3 to height 5?

5.1.8 13.8. Chapter 13 checklist

- [] I can explain why all values live in the leaves of the B+tree while branches hold only separators, and what this gives for search and range traversal.
- [] I know where a value that does not fit in a leaf goes (an overflow run) and what the leaf keeps in its place.
- [] I understand why a value spanning several overflow pages needs a contiguous run of pages, and what it costs for allocation and rewriting.
- [] I understand why libmdbx does not maintain its own buffer cache, and its consequences (copy-free reads, residency managed by the kernel, PTE growth when the DB $\gg$ RAM).
- [] I can name the address-space limit on database size and its consequence on 32-bit platforms.
- [] I know the purpose of the canary (`MDBX_canary`) and that the `v` field always holds the transaction number.
- [] I can state the reachability invariant and explain what is built on it (integrity auditing, `mdbx_chk`).

5.2 Chapter 14. MVCC and Snapshots

5.2.1 14.1. Page Versioning, Reader Snapshot

Each page is marked with the number of the transaction that created its current version. On start, a reader fixes a snapshot — the state of the database as of the moment the transaction began. Everything it sees afterwards is a consistent picture of that moment, regardless of subsequent commits.

5.2.2 14.2. Trio of Meta Pages and Two-Phase Commit

The database always has **three meta pages** (`NUM_METAS = 3`). This allows keeping two valid snapshots ("fresh" and "steady" — with data guaranteed to be flushed to disk) and one trailing slot for overwriting.

The meta update is **two-phase**: first one half is invalidated (writing the transaction number while zeroing the second), then the fields are filled in and the second half is validated. A reader sees either the old intact meta or the new one — never a "half-updated" one.

5.2.3 14.3. Finite State Machine of the Trio

The state of each meta is encoded independently; the full transition table covers **216 (6³)** combinations and is exhaustively checked by internal validation. This protects against "impossible" states after failures.

5.2.4 14.4. Reader Lock Table (RLT)

A reader registers its snapshot number in the **reader lock table** (in the lock file). The slot contains: the snapshot number, the owner's pid/tid, and fields describing how many pages the reader "pins".

The oldest active snapshot — the **detent** — is computed by scanning the RLT (lock-free, with caching). The detent is the watershed for page reuse (chapter 16).

5.2.5 14.5. Wait-free Reads and safe64

On read operations the reader **takes no locks** — it simply walks the mmap. The only "writing" action is registering a slot when the transaction starts.

64-bit fields (transaction numbers in the RLT and in the meta) are read through a special atomic protocol, **safe64**: values are written "low word first, then high word", and reads are retried when a "torn" value is detected. This is protection against torn 64-bit reads on weak memory models.

5.2.6 14.6. The Writer: Global Mutex, Front Txnid, Dirty List

At any moment there is **one write transaction** (the global writer mutex in the lock file). On start, the writer gets a **preliminary number** (`front txnid = txnid + 1`): new and CoW-copied pages are marked with it and are invisible to readers until commit. Modified pages accumulate in the **dirty list** (DPL).

Fragment from [examples/c++/19-readers-lag.c++](#) — a reader holds a snapshot; the writer's commits increase its "lag", and `mdbx_reader_list` shows reader slots with txnid and lag:

```
// Commits after the snapshot was created increase the reader's lag.
for (int i = 3; i < 8; ++i) {
    auto txn = env.start_write();
    auto wtable = txn.open_map(nullptr);
    txn.insert(wtable, mdx::slice("k" + std::to_string(i)), mdx::slice("v"));
    txn.commit();
}

// Enumerate the readers: a reader slot has lag > 0.
struct visitor {
    int operator()(const mdx::env::reader_info &ri, int) {
        std::cout << "reader slot: pid=" << ri.pid << " tid=" << ri.thread << " txnid=" << ri.transaction_id
                    << " lag=" << ri.transaction_lag << "\n";
        return mdx::continue_loop;
    }
} v;
env.enumerate_readers(v);
```

Full code: [19-readers-lag.c++](#).

Examples for this chapter: [examples/c++/19-readers-lag.c++](#).

5.2.7 14.7. Summary of Chapter 14

- MVCC: a page is marked with a txnid; a reader sees its own snapshot.
- The meta trio with a two-phase update and a finite state machine of 216 states.
- RLT and detent — the basis of safe reuse.
- Wait-free reads; safe64 against torn reads.
- One writer; front txnid hides uncommitted changes.

5.2.8 14.8. Exercises

1. Why exactly three meta pages, and not two?
2. What happens if the writer crashes in the middle of a two-phase meta update?

5.2.9 14.9. Chapter 14 checklist

- [] I can explain why a reader sees a consistent picture regardless of subsequent commits (the snapshot + the `txnid` label on pages).
- [] I know why exactly three meta pages are needed and which two valid snapshots they keep.
- [] I can describe the two-phase meta update and explain why a reader never sees a "half-updated" meta.

- [] I understand how the detent is computed by scanning the RLT and why it is the watershed for page reuse.
- [] I know that reads are wait-free and take no locks, and why the `safe64` protocol is needed.
- [] I can explain why the writer's uncommitted pages are invisible to readers (`front_txn_id` , the dirty list).

5.3 Chapter 15. Copy-on-Write and the Commit Pipeline

5.3.1 15.1. Page Lifecycle on Write

The page state is determined by comparing its label `mp->txn_id` with the transaction number:

State	Condition	Meaning
frozen	<code>txn_id < txn->txn_id</code>	Old version, visible to readers; cannot be changed — only copied
spilled	<code>txn_id == txn->txn_id</code>	Already written to disk by the current writer
shadowed	<code>txn_id > txn->txn_id</code>	A newer version exists; this one is obsolete
modifiable	<code>txn_id == front_txn_id</code>	New version of the current writer; can be modified in place

The touch mechanism (CoW): if a page is "frozen" — we allocate a new one (from GC or the file tail), copy the contents, mark it with `front_txn_id` , put it in the dirty list; the old one is released. The parent that pointed to the old page also becomes "modifiable" — CoW propagates upward to the root. Cursors that referred to the old version are moved to the new one.

5.3.2 15.2. Dirty Page List (DPL)

Within one transaction a page enters the dirty list **once**, no matter how many times it is modified — this is the key factor holding back WAF (write amplification factor, the ratio of bytes written to bytes of useful data; detailed in Volume IV, chapter 21).

5.3.3 15.3. Spill

If the volume of dirty pages exceeds the limit (`dp_limit` , default $\approx 1/42$ of RAM), **spill** kicks in: some pages are written to disk ahead of time. Pages referenced by cursors are not subject to spill; the selection order is governed by divisor options (minimum/maximum share).

Nuance: if a spilled page is then modified again — it will be written once more (additional amplification). This is a "memory vs WAF" trade-off.

5.3.4 15.4. Loose Pages and Refund

- **Loose pages** — a small cache of dirty pages freed in the current transaction (limit `loose_limit` , default 64) for fast reuse without consulting the GC.
- **Refund** — returning freed tail pages to unallocated space (rather than to the GC); reduces WAF and enables online shrinkage.

5.3.5 15.5. The Commit Pipeline

1. Close/verify cursors.
2. **GC processing:** freed pages are placed into GC-tree records.
3. **Refund:** tail frees are returned to unallocated space.
4. **Spill:** on dirty-list overflow — early write-out.
5. **Audit** (in enhanced checking modes): reconciliation of page sums.
6. **Two-phase meta update** + synchronization according to the durability mode.
7. Release the writer lock, update the detent cache, clean out dead readers.

Each stage is measurable via `mdbx_txn_commit_ex()` → `MDBX_commit_latency` (`preparation/gc_wallclock/audit/write/sync/ending/whole`).

Fragment from [examples/c++/20-commit-latency.c++](#) — collecting `MDBX_commit_latency` by stage via `commit_get_latency()` for a series of commits and averaging:

```
for (int i = 0; i < commits; ++i) {
    auto txn = env.start_write();
    auto table = txn.open_map(nullptr);
    txn.insert(table, mdbx::slice("k" + std::to_string(i)), mdbx::slice("v"));
    const auto lat = txn.commit_get_latency(); // returns MDBX_commit_latency
    prep += lat.preparation;
    gc += lat.gc_wallclock;
    audit += lat.audit;
    write += lat.write;
    sync += lat.sync;
    ending += lat.ending;
    whole += lat.whole;
}

std::cout << "avg latency (us): preparation=" << (prep / commits) << " gc=" << (gc / commits)
    << " audit=" << (audit / commits) << " write=" << (write / commits) << " sync=" << (sync / commits)
    << " ending=" << (ending / commits) << " whole=" << (whole / commits) << "\n";
```

Full code: [20-commit-latency.c++](#).

Examples for this chapter: [examples/c++/20-commit-latency.c++](#).

5.3.6 15.6. Summary of Chapter 15

- CoW: frozen → copy → modifiable; the path to the root is rewritten entirely.
- DPL deduplicates pages within a transaction — the main lever of WAF.
- Spill — a memory/WAF trade-off; loose and refund reduce writes.
- Commit pipeline: GC → refund → spill → audit → meta → sync → unlock.

5.3.7 15.7. Exercises

1. Why does changing a single byte in a leaf rewrite the whole path to the root?
2. How can `mdbx_txn_commit_ex` help see what "got stuck" — spill or sync?

5.3.8 15.8. Chapter 15 checklist

- [] Given a page's `mp->txnid` label, I can determine its state (frozen/spilled/shadowed/modifiable) and what may be done with it.
- [] I can explain why CoW rewrites the path to the root and how cursors are moved to new page versions.
- [] I know that a page enters the dirty list once per transaction, and why this holds back WAF.
- [] I can describe when spill kicks in (`dp_limit` ≈ 1/42 of RAM), which pages it never touches, and why a re-modified spilled page is bad.
- [] I distinguish loose pages (fast reuse, `loose_limit` = 64) from refund (returning tail pages to unallocated space).
- [] I can reproduce the order of the commit pipeline and name the stages visible in `MDBX_commit_latency`.

5.4 Chapter 16. GC — the Garbage Collector Inside the Database

5.4.1 16.1. Why There Is No Free-List

libmdbx has no classic free-page list. Freed pages are accounted for **persistently, inside the data file**, in a special **GC tree** (a separate table whose root is in the meta).

5.4.2 16.2. Format of a GC Record

- **Record key** — the number of the transaction that freed the pages.
- **Value** — a list of page numbers (compressed, as ranges).
- One record is physically limited (~1000 numbers at a 4 KB page).

The GC format is frozen and has not changed since v11.3 — an important compatibility guarantee.

5.4.3 16.3. Detent: Safety of Reuse

Pages from a GC record with key τ can be reused only when $\tau \leq \text{detent}$ — the oldest active snapshot. Above the detent, records stay "frozen" until the corresponding reader finishes.

The detent is computed by scanning the RLT (lock-free, with caching of the result).

5.4.4 16.4. FIFO vs LIFO

Policy	Mechanism	Effect
FIFO (default)	Pages are taken from the oldest suitable record	The list lives longer, cools down on disk
LIFO (<code>MDBX_LIFORECLAIM</code>)	The newest frees come first	Minimally short circulation cycle; pages are still "warm"; on systems with a write-back cache — several-fold write performance growth

Nuance: `MDBX_LIFORECLAIM` gives almost no effect with `SAFE_NOSYNC` / `UTTERLY_NOSYNC` — there the cycle length is determined by the frequency of `env_sync()`. Vector (SIMD) kernels are used to find dense sequences: SSE2/AVX2/AVX512/NEON.

5.4.5 16.5. BigFoot

One GC record holds ~1000 page numbers (~4 MB at 4 KB). If a transaction frees more (for example, when replacing a huge value), the records form a **chain over consecutive txnids** — the BigFoot mode. Readers must be newer than the whole chain. The price — more records, a taller GC tree.

BigFoot is enabled by default in 64-bit builds (`MDBX_ENABLE_BIGFOOT`); it is fully compatible with the database format.

5.4.6 16.6. Recursiveness of GC Updates

The GC is also a CoW tree: modifying the GC requires allocating pages, which affects the list of pages that goes into the GC. That is why an operational reserve of free pages is formed before an update: shortage → irrational database growth, surplus → overhead.

5.4.7 16.7. `rp_augment_limit` and `gc_time_limit`

Searching for dense sequences (for large values) can be expensive under fragmentation. Two cost limiters:

- `rp_augment_limit` — the limit of list accumulation during search; exceeding it → it is cheaper to append new pages to the file tail;
- `gc_time_limit` — a time limit (in 1/65536 seconds) on the search during a writing transaction.

Rule: decrease `rp_augment_limit` only together with `gc_time_limit`; too small an `rp_augment` does not cure GC growth but aggravates it for the sake of long-record insert speed.

Fragment from `examples/c++/22-gc-limits.c++` — setting and reading GC limits: `rp_augment_limit` (page reserve for reclamation) and `gc_time_limit` (1/65536 seconds per sequence search):

```
env.set_extra_option(opt::rp_augment_limit, 128 * 1024);
// gc_time_limit is specified in 1/65536 fractions of a second (16.16 fixed point):
// 2500 ≈ 38 ms for searching page sequences in the GC within a transaction.
env.set_extra_option(opt::gc_time_limit, 2500);

// A load with long values: makes the GC work harder.
{
    auto txn = env.start_write();
    auto table = txn.open_map(nullptr);
    for (int i = 0; i < 2000; ++i) {
        const auto key = "k" + std::to_string(i);
        const std::string value(4096 + (i % 7) * 512, 'v');
        txn.insert(table, mdbx::slice(key), mdbx::slice(value));
    }
    txn.commit();
    auto del = env.start_write();
    auto dtable = del.open_map(nullptr);
```

```

for (int i = 0; i < 2000; i += 2)
    del.erase(dtable, mdbx::slice("k" + std::to_string(i)));
del.commit();
}

std::cout << "rp_augment_limit = " << env.extra_option(opt::rp_augment_limit) << "\n";
std::cout << "gc_time_limit = " << env.extra_option(opt::gc_time_limit) << "\n";

```

Full code: [22-gc-limits.c++](#).

5.4.8 16.8. Early GC Cleanup (2025) and Non-Deferred Cleanup (devel)

- **Early cleanup (2025):** recycled GC records start to be processed earlier.
- **Non-deferred cleanup (devel, 0.14.x):** recycled GC records are removed immediately after reading rather than at commit; overhead becomes proportional to the volume of operations. This also opened the path to explicit copy-free defragmentation (0.14.2+).

Examples for this chapter: [examples/c++/21-gc-observe.c++](#); [examples/c++/22-gc-limits.c++](#).

5.4.9 16.9. Summary of Chapter 16

- GC — a tree of "txnid → page list" records inside the file (not a free-list).
- Reuse only newer than the detent.
- FIFO by default; `MDBX_LIFORECLAIM` — LIFO.
- BigFoot — chains for large frees.
- `rp_augment_limit` / `gc_time_limit` — control over search cost.

5.4.10 16.10. Exercises

1. Explain why a long-lived reader "freezes" reclamation, using the concept of the detent.
2. When does LIFO give a win, and when not?

5.4.11 16.11. Chapter 16 checklist

- [] I can explain why libmdbx has no free-list and where freed pages are accounted for persistently.
- [] I describe the format of a GC record: key — the freeing txnid, value — a list of pages as ranges; the format is frozen since v11.3.
- [] I can explain the reuse condition via the detent and what happens to GC records above the detent.
- [] I know the difference between FIFO and LIFO and when `MDBX_LIFORECLAIM` gives a win and when it does not.
- [] I understand the BigFoot mechanism: chains of records over consecutive txnids and the requirement that readers be newer than the whole chain.
- [] I can explain why GC updates are recursive (the GC is itself a CoW tree) and why an operational reserve of pages is needed.
- [] I know the purpose of `rp_augment_limit` and `gc_time_limit` and the rule of decreasing them together.

5.5 Chapter 17. Growth, Shrinkage, and Defragmentation of the Database

5.5.1 17.1. Why the Database Grows

- **Long-lived readers** freeze the detent → GC does not reuse pages → the writer takes new ones from the file tail. Even a put→del cycle "swells".
- `MDBX_SAFE_NOSYNC` — a permanent quasi-long-lived reader (the last steady commit).
- **Fragmentation** and a shortage of contiguous sequences for large values.
- **Forgotten readers** (thread crash, wrong TLS).

5.5.2 17.2. Geometry: growth_step, upper, shrink_threshold

The file grows automatically, in multiples of `growth_step`, up to `upper`. Truncation is possible only down to the last used page; a single occupied page near the end can block shrinkage indefinitely. Shrinkage requires hysteresis (shrink step > growth step), otherwise the file "breathes".

5.5.3 17.3. MDBX_MAP_FULL and Resolution Mechanisms

When the GC is empty/frozen and the file has hit `upper`, the following fire, when possible: a new steady point (detent shift) → the HSR callback (HSR — Handle-Slow-Readers, the mechanism for evicting stuck readers; detailed in Volume V, ch. 29) → eviction of parked readers → and only then `MDBX_MAP_FULL`.

Fragment from `examples/c++/23-map-full.c++` — reproduction of `MDBX_MAP_FULL` (in the example, a rigid geometry limits the file to 512 KB) and resolution by increasing `size_upper`:

```
size_t inserted = 0;
try {
    auto txn = env.start_write();
    auto table = txn.open_map(nullptr);
    for (;;) {
        const auto key = "k" + std::to_string(inserted);
        const std::string value(1024, 'x');
        txn.insert(table, mdbx::slice(key), mdbx::slice(value));
        ++inserted;
    }
} catch (const mdbx::db_full &ex) {
    std::cout << "MDBX_MAP_FULL after " << inserted << " inserts\n";
}

// Increase the upper bound — and continue from the same place.
mdbx::env::geometry bigger;
bigger.size_upper = 8 * 1024 * 1024;
try {
    env.set_geometry(bigger);
} catch (const mdbx::db_unable_extend &) {
    // On 32-bit Windows the address range right after the mapping may be
    // occupied: reopen the same database with the enlarged geometry.
    mdbx::env::geometry reopened = geo;
    reopened.size_upper = bigger.size_upper;
    env.close();
    env = mdbx::env_managed(path,
        mdbx::env_managed::create_parameters().set_geometry(reopened),
        mdbx::env::operate_parameters());
}
```

Raising `upper` on a live database requires remapping: on most platforms this happens in place, but on 32-bit Windows there may be no free address range next to the current mapping — then `MDBX_UNABLE_EXTEND_MAPSIZE` is returned, and the portable path is to close and reopen the database with the enlarged geometry (the new mapping is created with the required `upper` right away).

Full code: [23-map-full.c++](#).

5.5.4 17.4. Auto-Compaction (Implicit Shrink)

On every commit, freed tail pages are returned to unallocated space (refund). When the free tail exceeds `shrink_threshold` (+ a small reserve), the commit additionally drops the "excess" range via `madvise(MADV_DONTNEED/REMOVE)` and truncates the file.

Limitations: it does not work while long-lived readers hold the tail or while several processes use the file; with `SAFE_NOSYNC` it is limited by the frequency of steady points.

5.5.5 17.5. Explicit Defragmentation

`mdbx_env_defrag()` / the `mdbx_defrag` utility (0.14.2+) — careful rearrangement of pages toward the beginning of the file in cycles; each cycle is a separate committed transaction. Parameters: `defrag_atleast` / `defrag_enough` (minimum/sufficient reduction), `time_atleast` / `time_limit` (1/65536 s), `acceptable_backlash`, `preferred_batch`. The result code is `MDBX_defrag_enough_threshold`.

Guaranteed file reduction — also `mdbx_copy -c` (compacted copy) or `dump+load (-a)`.

5.5.6 17.6. Estimating Capacity

Dividing the file size by the item size **overestimates** the real capacity: page granularity, the path to the root, GC, and hysteresis add overhead. Estimate for the worst case of reuse freezing.

Examples for this chapter: [examples/c++/23-map-full.cpp](#); [examples/c++/24-defrag.cpp](#).

5.5.7 17.7. Summary of Chapter 17

- Growth = pages not being reused (readers/SAFE_NOSYNC/fragmentation).
- `upper` — the hard limit; `MAP_FULL` is resolved by steady/HSR/eviction.
- Implicit shrink — free compaction as part of commits.
- Explicit defragmentation in cycles; guaranteed shrinkage — `copy -c / dump+load`.

5.5.8 17.8. Exercises

1. Why is "10 GB in the database, so 10 GB of data" an estimation error?
2. Describe a scenario where auto-compaction will not work.

5.5.9 17.9. Chapter 17 checklist

- [] I name the causes of database growth: long-lived readers, `MDBX_SAFE_NOSYNC`, fragmentation, forgotten readers.
- [] I understand why truncation is possible only down to the last used page and why shrink-step hysteresis is needed.
- [] I can reproduce the chain that resolves `MDBX_MAP_FULL`: steady point → HSR callback → eviction of parked readers → the error.
- [] I know the limitations of auto-compaction (implicit shrink) and when it will not work.
- [] I can describe explicit cycle-based defragmentation and the guaranteed file-reduction paths (`mdbx_copy -c` , `dump+load`).
- [] I can explain why dividing the file size by the item size overestimates the real capacity.

5.6 Chapter 18. Nested Transactions

5.6.1 18.1. Why They Are Needed

A nested transaction is a child write-transaction inside a parent one, allowing changes to be grouped with the ability to **partially roll back**.

5.6.2 18.2. Implementation

A child transaction works on the same transaction number and front `txnid`. It inherits from the parent: the page space and the dirty list (with a shadow of the parent's), retired lists (lists of pages already freed but still held back from reuse — see 19.4), GC buffers, table descriptors, and cursors (via the mechanism of "paired" cursors).

5.6.3 18.3. Commit (join) vs Abort (undo)

- **Commit:** the child's dirty pages are merged into the parent's list; GC counters are transferred. If the child is "clean" — a fast path without merging.
- **Abort:** all changes are discarded; retired pages are "un-retired"; handles and cursors are restored.

Fragment from [examples/c++/25-nested-txn.cpp](#) — a nested transaction: `start_nested()` + `abort()` rolls back its changes (undo); the symmetric `commit()` merges them into the parent (join):

```
// A nested transaction with a rollback (undo).
{
    auto nested = txn.start_nested();
    auto ntable = nested.open_map(nullptr);
```

```

    nested.insert(nstable, mdbx::slice("k2"), mdbx::slice("v2"));
    nested.abort(); // roll back the nested one
}
std::cout << "parent sees k1 after nested abort: " << txn.get(table, mdbx::slice("k1")).as_string() << "\n";
try {
    (void)txn.get(table, mdbx::slice("k2"));
    std::cerr << "FAIL: k2 survived nested abort\n";
    return EXIT_FAILURE;
} catch (const mdbx::not_found &) {
    std::cout << "k2 rolled back by nested abort: absent\n";
}

```

Full code: [25-nested-txn.c++](#).

5.6.4 18.4. Fate of Tables

- A table **created** in a nested transaction: stays on commit, disappears on abort.
- A table **dropped** in a nested transaction: is deleted on commit, "comes back to life" on abort.

5.6.5 18.5. Anti-Pattern: a Nested Transaction in Every Function

Calling `mdbx_txn_begin(parent, ...)` in every small function is an anti-pattern: the meaning of grouping is lost and overhead grows. Nest deliberately, along the boundaries of logical operations.

5.6.6 18.6. Workaround for Lack of Space

Nested transactions are a known workaround for lack-of-space scenarios (you can partially roll back without losing previous changes).

Nuance: nested transactions do not combine with `MDBX_WRITEMAP`.

Examples for this chapter: [examples/c++/25-nested-txn.c++](#).

5.6.7 18.7. Summary of Chapter 18

- A nested transaction = a child write-transaction with partial rollback.
- Commit — merging into the parent; Abort — full rollback.
- The fate of created/dropped tables follows the CoW rules.
- Do not nest "for every function"; do not combine with `WRITEMAP`.

5.6.8 18.8. Exercises

1. Write an example of "a group of updates with a rollback of the last step" via a nested transaction.
2. Check the fate of a table created and dropped in a nested transaction upon abort.

5.6.9 18.9. Chapter 18 checklist

- [] I can explain that a child transaction works on the same transaction number, and what it inherits from the parent.
- [] I distinguish commit (join): merging the child's dirty pages into the parent, from abort (undo): full rollback and "un-retiring" pages.
- [] I know the fate of tables created and dropped in a nested transaction, on commit and on abort.
- [] I can demonstrate a partial rollback via `start_nested()` + `abort()`.
- [] I can justify why a nested transaction "in every function" is an anti-pattern.
- [] I know that nested transactions are a workaround for lack of space, and that they do not combine with `MDBX_WRITEMAP`.

5.7 Chapter 19. The Lock File and Interprocess Synchronization

5.7.1 19.1. Why a Separate File

Next to the data file lives the lock file (`.lck`): the global writer mutex, the reader table (RLT), statistics, caches of the oldest snapshot. It is needed for coordination **between processes**.

Fragment from `examples/c++/26-two-processes.c++` — interprocess access: a writer process commits data, a reader process (a child via `fork()`) opens the same database and reads it:

```
// Writer: the parent process commits data and closes the environment.
{
    auto env = example::env_open(path);
    auto txn = env.start_write();
    auto table = txn.open_map(nullptr);
    txn.insert(table, mdbx::slice("k"), mdbx::slice("v1"));
    txn.commit();
    std::cout << "parent wrote: k=v1" << std::endl;
}

const pid_t pid = fork();
if (pid < 0) {
    std::cerr << "fork failed\n";
    return EXIT_FAILURE;
}

if (pid == 0) {
    // Reader: the child opens the same database and reads the data.
    auto env = example::env_open(path);
    auto txn = env.start_read();
    auto table = txn.open_map(nullptr);
    std::cout << "child reads: " << txn.get(table, mdbx::slice("k")).as_string() << std::endl;
    txn.abort();
    _exit(EXIT_SUCCESS);
}
```

Full code: [26-two-processes.c++](#).

5.7.2 19.2. Lock Implementations

Build option `MDBX_LOCKING` :

Flavour	Primitives	LCK signature
POSIX2008	Robust mutexes (Linux default)	0x8017
POSIX2001	Shared mutexes	0x8017
SYSV	Semaphores (macOS default)	0xF18D
WIN32FILES	Windows (<code>LockFileEx</code>)	0xF10C
POSIX1988	(legacy)	0xFC29

Nuance: `POSIX2001` and `POSIX2008` share one signature — **this is not a typo**. Both flavors store the same kind of interprocess primitive in the LCK — a `pthread_mutex_t`, so their on-disk formats are byte-identical and the signature is common. The difference lies only in the API features used: robust mutexes appeared in POSIX.1-2008, while the 2001 mode works with plain process-shared mutexes. Consequence: a database opened in one of these two modes can be continued in the other — but not with `SYSV`, `WIN32FILES`, or `POSIX1988` (their signatures differ, and an LCK of such a format is rejected).

The actual compatibility check at open time uses not the raw signature but the derived `MDBX_LOCK_FORMAT` hash — it mixes the flavor's signature with `sizeof(reader_slot_t)` and the offsets of the key `lck_t` fields, so the constants above are just the per-primitive-format "base", while the full identity also accounts for the layout of the remaining fields.

The LCK format is versioned **separately** from the data format: a structure change breaks only shared access to an already-open database, not the data file itself.

5.7.3 19.3. Why LockFileEx on Windows Instead of Named Mutexes

File locks (`LockFileEx`) were chosen deliberately: they work on network drives and protect against incompetent actions. The price — in naive benchmarks with many small transactions libmdbx may lag behind LMDB (acquiring/releasing a file lock — hundreds of microseconds).

5.7.4 19.4. Reader Slot (RLT)

txnid (atomic)	snapshot number (INVALID_TXNID = free)
tid	owner; PARKED = UINT64_MAX, OUSTED = UINT64_MAX-1
pid	owner process
snapshot_pages_used	first_unallocated at snapshot time
snapshot_pages_retired	how many pages it holds back from reuse

Slot liveness is checked using OS facilities (protection against pid/tid reuse). Stale slots are not scanned one by one — the whole table is reinitialized when the LCK is opened by a single process.

5.7.5 19.5. Recovering Locks After a Process Failure

After a process crash the OS releases its locks; dead reader slots are cleaned up by the liveness mechanisms and `mdbx_reader_check()`.

5.7.6 19.6. Mode Without a Lock File

A "single process, exclusive" mode without `.lck` is possible (a build option) — when interprocess access is not needed.

Examples for this chapter: [examples/c++/26-two-processes.c++](#).

5.7.7 19.7. Summary of Chapter 19

- LCK: writer mutex + RLT + statistics; versioned separately.
- Implementations: POSIX/SysV/WIN32FILES; flavour signatures.
- LockFileEx — the choice for network drives; slower for small transactions.
- RLT slot: txnid/tid/pid + pinning fields; liveness via the OS.
- A mode without LCK is possible (single process).

5.7.8 19.8. Exercises

1. Why is deleting `.lck` while the database is open a mistake?
2. How is `mdbx_reader_check` different from "automatic" cleanup?

5.7.9 19.9. Chapter 19 checklist

- [] I can list what lives in the lock file and why it is needed for interprocess access.
- [] I know that the LCK format is versioned separately from the data format, and what that means for compatibility.
- [] I can explain why `LockFileEx` was chosen on Windows, and what we pay for it in benchmarks with small transactions.
- [] I describe an RLT reader slot and the fields recording how many pages the reader holds back from reuse.
- [] I understand how slot liveness is checked with OS facilities and what cleans dead slots after a process crash (`mdbx_reader_check()`).
- [] I know when the mode without a lock file is possible (single process, exclusive).

5.8 Chapter 20. Durability and Recovery

5.8.1 20.1. Two-Phase Meta Update: weak vs steady

- **weak** — data is in the file but not guaranteed to be flushed to persistent media;
- **steady** — data is flushed to disk; the snapshot survives a system failure.

5.8.2 20.2. Recovery Without a WAL

There is deliberately no WAL. After a crash, on open the last **intact and valid** meta is selected (a consistent trio, matching txnid halves, a checksum). "Half-written" transactions that did not manage to publish the meta simply do not exist.

`mdbx_txn_checkpoint(txn, weakening_durability, &latency)` — a **commit variant**: it commits the transaction and returns per-stage latencies (`MDBX_commit_latency`), and allows weakening durability via `weakening_durability`. The header marks it as "may be changed in future releases" — use it for diagnostics, not as a stable API. Forcing the steady point is done with `mdbx_env_sync_ex(env, force=true, ...)` (Volume II, chapter 10), not with a checkpoint.

5.8.3 20.3. Open for Recovery

`mdbx_env_open_for_recovery()` opens the database with a choice of the target meta page (`target_meta`) under an exclusive lock, to resolve a "stuck" trio. Since 0.12.7+ recovery checks **do not modify the database**. The `MDBX_WANNA_RECOVERY` code is returned on a read-only open of a database requiring recovery.

Warning: the header marks this function as internal API of the `mdbx_chk` utility, "subject to change at any time" — it is not recommended in application code; the regular path is a read-write open or `mdbx_chk`.

Fragment from `examples/c++/27-recovery.c++` — a writer crash (`_exit` without commit/abort) is rolled back on reopen (steady), and `open_for_recovery()` opens a specific meta page:

```
// Reopen: uncommitted changes were rolled back.
{
    auto env = example::env_open(path);
    auto txn = env.start_read();
    auto table = txn.open_map(nullptr);
    const std::string value(txn.get(table, mdbx::slice("key")).as_string());
    std::cout << "crashed writer changes rolled back: key=" << value << ", newkey ";
    try {
        (void)txn.get(table, mdbx::slice("newkey"));
        std::cout << "present\n";
        return EXIT_FAILURE;
    } catch (const mdbx::not_found &) {
        std::cout << "absent\n";
    }
    txn.abort();
}

// Recovery mode: open a specific meta page (0..2).
{
    auto recovery = mdbx::env_managed::open_for_recovery(path.c_str(), 0 /* target_meta */, true);
    std::cout << "open_for_recovery: ok\n";
}
```

Full code: [27-recovery.c++](#).

5.8.4 20.4. boot_id: Semantics and Behavior in LXC

`boot_id` — the OS boot identifier, taken into account when rolling back weak metas. In LXC, `boot_id` may be absent/shared across containers — the rollback control for weak metas accounts for this. This matters for failures inside containers.

5.8.5 20.5. Unified Page Cache Incoherence

A known issue, issue #269: incoherence of the unified page cache between processes. The protection — `MDBX_FORCE_CHECK_MMAP_COHERENCY` (a build option, default 0); introduced in the 0.11.5–0.11.6 series; the trigger counter is visible in diagnostics.

5.8.6 20.6. mdbx_chk: Integrity Checking

`mdbx_chk` — deep validation: metas/trio, trees, ordering, the reachability invariant, GC. Modes: `-w` (read-write, rollback to steady), `-d` (page-by-page traversal — without it you cannot find lost/double-used pages), `-i` (ignore false ordering errors with custom comparators), `-0/-1/-2` (a specific meta), `-vvvvv` (histograms). Exit code: 0 = clean.

5.8.7 20.7. What Survives a Power Failure — by Mode

Mode	After a failure
<code>MDBX_SYNC_DURABLE</code>	All committed data is in place
<code>MDBX_NOMETASYNC</code>	Loss of the last commits (meta deferred)
<code>MDBX_SAFE_NOSYNC</code>	Rollback to the last steady
<code>MDBX_UTTERLY_NOSYNC</code>	No guarantees; corruption possible

Examples for this chapter: [examples/c++/27-recovery.c++](#).

5.8.8 20.8. Summary of Chapter 20

- Two-phase meta: weak/steady; recovery — selecting the last intact meta.
- Open-for-recovery with `target_meta`; since 0.12.7 checks do not modify the database.
- `boot_id` matters in LXC; page-cache incoherence — `#269 + FORCE_CHECK`.
- `mdbx_chk` — the main verification tool; see the modes above.

5.8.9 20.9. Exercises

1. Describe what will happen to the database after kill -9 in the middle of a commit, in `DURABLE` mode.
2. Why is `mdbx_chk -d` needed to find lost-unused pages?

5.8.10 20.10. Chapter 20 checklist

- [] I distinguish weak and steady snapshots and know which one survives a system failure.
- [] I can explain how recovery works without a WAL: selecting the last intact and valid meta; "half-written" transactions simply do not exist.
- [] I know the purpose of `mdbx_env_open_for_recovery()` with `target_meta` and why it is not recommended in application code.
- [] I understand the role of `boot_id` when rolling back weak metas, and its specifics in LXC.
- [] I know about the unified page cache incoherence (`#269`) and the `MDBX_FORCE_CHECK_MMAP_COHERENCY` protection.
- [] I can pick the right `mdbx_chk` modes (`-w`, `-d`, `-i`, `-0/-1/-2`) for the task and understand why `-d` is needed.
- [] For each durability mode, I can say what will survive a power failure.

5.9 Volume Conclusion

You understand the internals of libmdbx: from the page and B+tree to the meta, GC, and recovery. Now you can explain any observed behavior through the architecture.

What's next: Volume IV — performance and optimization: WAF, choosing a configuration for the scenario, micro-optimizations, get-cached, bulk operations, benchmarks.

libmdbx v0.15.0-150-g6ca39a94 — built 2026-10-19 · [online version](#) · [PDF version](#)

6. Volume IV. Performance and Optimization

Level: for experienced developers. **Goal of the volume:** you know how to tune libmdbx for a specific scenario, understand the sources of write amplification and bottlenecks, and can measure and interpret metrics.

6.1 Chapter 21. WAF — write amplification factor

6.1.1 21.1. What is WAF and why it matters

WAF (Write Amplification Factor) — the ratio of bytes actually written to disk to the amount of data accepted from the application. WAF = 100 means: for every byte you write, the disk receives 100 bytes. A high WAF means SSD wear, a shorter disk lifetime and worse throughput.

6.1.2 21.2. Sources of amplification

1. **Page granularity.** CoW does not modify an old page, it creates a new one. Changing one byte in a leaf = writing a whole page (e.g., 4 KB).
2. **Path to the root.** A new version of a leaf changes the pointer in its parent → the parent also becomes new → and so on up to the root. A point update = $(\text{height}+1) \times \text{pagesize}$ bytes written.
3. **GC writes and meta.** Freed pages are described in the GC-tree; two-phase meta adds another 1–3 pages per commit.
4. **Spill.** A spilled page that is modified again is written once more.
5. **Merge/rebalance.** Merging half-empty pages can rewrite neighboring ones.

6.1.3 21.3. Batching — the main lever

Within a single transaction a page enters the dirty list **once**. N operations on M unique pages write $M \times \text{pagesize}$ bytes (+GC+meta). Therefore:

- one operation per transaction: $\text{WAF} = \text{height}+1 \approx 4-5$ (and higher);
- 10 000 operations in a single transaction: WAF approaches "1 page per many changes".

Conclusion: batching (100–10 000 operations per commit) is the most powerful lever for reducing WAF.

6.1.4 21.4. Influence of tree height

B+tree height $\sim \log_B(N)$, where B is the average number of children in a node (hundreds). Increasing the data volume by 2^k times adds $\sim k$ levels. For point-update the specific WAF grows **logarithmically** with volume — slowly.

6.1.5 21.5. Spill as a trade-off

When the dirty list overflows `dp_limit` ($\approx 1/42$ of RAM), some pages are flushed early. If such a page is modified again — it is written once more (extra WAF). This is a "memory vs WAF" trade-off.

6.1.6 21.6. `prefer_waf_insteadof_balance` and `merge_threshold`

The `prefer_waf_insteadof_balance` option intentionally sacrifices page fill uniformity for a smaller number of written pages (reuse an already-modified neighbor page); it is enabled by default. The merge threshold is `MDBX_opt_merge_threshold` (16.16 %; default 33%, allowed range [12.5%..50%]). Both are direct levers of WAF.

6.1.7 21.7. WAF calculation for typical scenarios

For a point-update in a tree of height h on a 4 KB page: write $\approx (h+1) \times 4096$ bytes per operation. At $h=4$ and a 32-byte value: $20 \text{ KB} / 32 \text{ B} \approx \text{WAF} \approx 640$ (single operations!). With batching of 10 000 records to the same 5 000 unique pages: $5\,000 \times 4096 / (10\,000 \times 32) \approx \text{WAF} \approx 64$. As the batch grows further, WAF keeps decreasing.

Warning: numbers without context are useless — always state the sync mode, page size, tree height and transaction size.

Fragment from `examples/c++/28-waf-batching.c++` — measuring written pages (`newly+cow` from `mi_pgop_stat`, the same counters as `mdbx_stat -p`) for different write batch sizes:

```
constexpr int total = 1000;
const auto initial = env.get_info().mi_pgop_stat;
for (int off = 0; off < total; off += (int)batch_size) {
    auto txn = env.start_write();
    auto table = txn.open_map(nullptr);
    for (int i = off; i < off + (int)batch_size && i < total; ++i)
        txn.insert(table, mdbx::slice("k" + std::to_string(i)), mdbx::slice("value"));
    txn.commit();
}
const auto final = env.get_info().mi_pgop_stat;
const uint64_t pages = (final.newly - initial.newly) + (final.cow - initial.cow);
std::cout << "batch=" << batch_size << ": pages=" << pages << " ops=" << total << " pages/op=" << (double)pages / total
<< "\n";
```

Full code: `28-waf-batching.c++`.

Examples for this chapter: `examples/c++/28-waf-batching.c++`.

6.1.8 21.8. Chapter 21 summary

- WAF = written to disk / accepted from the application.
- Sources: page granularity, path to the root, GC/meta, spill, merge.
- Batching is the main lever: a page is written once per transaction.
- Growth with volume is logarithmic; large values give a linear contribution.

6.1.9 21.9. Exercises

1. Calculate WAF for a point-update at height 3 and a 4 KB page.
2. How will a batch of 1 000 updates on the same 500 pages affect it?

6.1.10 21.10. Chapter 21 checklist

- [] I can compute the write for a point-update as `(height+1) × pagesize` bytes and derive the WAF from it.
- [] I name all the sources of amplification: page granularity, path to the root, GC/meta, spill, merge/rebalance.
- [] I can explain why a page enters the dirty list once per transaction and why batching is the main lever for reducing WAF.
- [] I know that the specific WAF grows logarithmically with volume and can explain why ($\text{height} \sim \log_B(N)$).
- [] I understand the spill trade-off: `dp_limit` $\approx 1/42$ of RAM, and a spilled page being written again.
- [] I know the effect of `prefer_waf_insteadof_balance` and `MDBX_opt_merge_threshold`.
- [] I present WAF numbers only with context: sync mode, page size, tree height, transaction size.

6.2 Chapter 22. Choosing a configuration for the scenario

6.2.1 22.1. Scenario 1: High write throughput (Ethereum-like)

- Mode: `MDBX_SAFE_NOSYNC` + auto-sync (`syncbytes / syncperiod`), or `NOMETASYNC`.
- Geometry: `upper` with headroom; large `growth_step` (fsync less often).

- Page: 4–8 KB.
- Batching: tens to thousands of operations per commit.
- Expected numbers: hundreds of thousands to millions of inserts/s (depends on hardware and key size).

6.2.2 22.2. Scenario 2: Read-heavy with rare writes (cache, index)

- Mode: default (`DURABLE`); writes are rare, reliability matters.
- Page: matched to the value size; short keys.
- `maxreaders` — according to the number of reader threads.
- Reads scale linearly across cores (wait-free).

6.2.3 22.3. Scenario 3: Embedded mobile application (Isar-like)

- Small database; 4 KB page.
- `MDBX_EXCLUSIVE` when exclusive access is needed.
- Careful read-only transactions; parking for long reads.
- Low WAF (little churn) — extends flash lifetime.

6.2.4 22.4. Scenario 4: Analytics, full DB scan

- Full-scan reading; important: a 64 KB page for sequential reads? (No — everything is read; what matters is locality and the absence of long writers.)
- Cursors + `get_batch`; range estimation via `estimate_*`.

6.2.5 22.5. Scenario 5: Multi-process reading (web server + workers)

- Several processes open one database read-only; one — write.
- `maxreaders` accounts for all processes; the LCK-file is shared.
- Be careful with `boot_id` in containers; page-cache coherence (#269).

6.2.6 22.6. Scenario 6: Low latency, small transactions

- Each transaction — one operation (minimal latency).
- Mode: `NOMETASYNC` / `SAFE_NOSYNC`, so as not to pay `fsync` for each one.
- On Windows remember `LockFileEx` (small transactions are expensive).
- Overhead guideline: ~0.5 μ s per transaction on top of I/O.

Fragment from `examples/c++/29-scenario-configs.c++` — typical configurations as functions returning `operate_parameters`: flags and options for each scenario of the chapter:

```
// Write-heavy: lazy sync, enough named tables.
params write_heavy() { return params().lazy_weak_tail().set_max_maps(64); }

// Read-heavy: maximum durability, many reader slots.
params read_heavy() { return params().robust_synchronous().set_max_readers(1024).set_max_maps(64); }

// Mobile device: small file size, gentle with flash memory.
params mobile() { return params().lazy_weak_tail().set_max_maps(64); }

// Analytics: batched writes, do not fuss with balancing.
params analytics() { return params().lazy_weak_tail().set_max_maps(64); }

// Multi-process read: many reader slots, stable geometry.
params multi_process_read() { return params().set_max_readers(4096).set_max_maps(64); }

// Low latency: NOMETASYNC (data on disk, metadata may lag).
params low_latency() { return params().half_synchronous_weak_last().set_max_maps(64); }
```

Full code: [29-scenario-configs.c++](#).

Examples for this chapter: [examples/c++/29-scenario-configs.c++](#).

6.2.7 22.7. Chapter 22 summary

- The sync mode is chosen by durability requirements.
- Geometry is set with headroom; do not undersize `upper`.
- Batching scales to the scenario (latency vs throughput).
- Platform specifics (Windows LockFileEx, LXC `boot_id`) matter.

6.2.8 22.8. Exercises

1. Pick a configuration for the "analytics" scenario and justify the page size choice.
2. What changes for "low latency" on Windows compared to Linux?

6.2.9 22.9. Chapter 22 checklist

- ☐ For each of the six scenarios I can justify the choice of sync mode (from `DURABLE` to `SAFE_NOSYNC` / `NOMETASYNC`).
- ☐ I know that geometry is set with headroom and why `upper` should not be undersized.
- ☐ I can pick a batch size matching the latency/throughput balance of a scenario.
- ☐ I understand when `maxreaders` matters (read-heavy, multi-process reading).
- ☐ I remember the platform traps: LockFileEx on Windows, `boot_id` in containers, page-cache coherence (#269).
- ☐ I can explain why switching to a large page for full-scan is not a silver bullet.

6.3 Chapter 23. Micro-optimizations

6.3.1 23.1. SIMD search for free pages

Searching for dense sequences in the GC uses vector kernels: SSE2/AVX2/AVX512/NEON (speedup $\times 4/\times 8/\times 16$; chosen at runtime). This is an internal optimization — you only influence it via `rp_augment_limit` / `gc_time_limit`.

6.3.2 23.2. Branchless binary search (CMOV)

Binary search over the separators of branch pages uses unconditional moves (CMOV) — no branching, predictability. The gain is noticeable in hot search paths.

6.3.3 23.3. Radix-sort and sorting networks

Used in internal procedures (sorting page lists, PNL merging).

6.3.4 23.4. C11 atomics for weak memory models

On ARM/AArch64/PPC/MIPS/RISC-V careful atomics are used (acquire/release, CAS, safe64). Correctness is guaranteed by the build (atomicity checking).

6.3.5 23.5. Minimizing system calls

Hot paths avoid syscalls; I/O batching via bulk iovec; `mdbx_copy` uses bulk writes.

6.3.6 23.6. Prefault write and mincore

`MDBX_opt_prefault_write_enable` — proactive page writing to eliminate page-faults and disk reads on first access in `MDBX_WRITEMAP`. Residency is tracked via `mincore` / `madvise`. The win is when the DB > RAM and pages are frequently evicted.

6.3.7 23.7. Auto-appending split and merging with a dirty neighbor

- Auto-appending split — efficient addition of keys at the end (append scenarios).
- Merge with an already-dirty neighbor — fewer CoW copies (related to `prefer_waf_insteadof_balance`).

Fragment from [examples/c++/30-append-prefault.c++](#) — batch loading of sorted keys: `prefault_write_enable + txn.append() (MDBX_APPEND)`:

```
static uint64_t load_append(const std::string &path, int count) {
    mdbx::env::remove(path);
    auto env = example::env_open(path);
    env.set_extra_option(mdbx::env::extra_runtime_option::prefault_write_enable, 1);

    const auto t0 = std::chrono::steady_clock::now();
    auto txn = env.start_write();
    auto table = txn.open_map(nullptr);
    for (int i = 0; i < count; ++i)
        txn.append(table, mdbx::slice(key_of(i)), mdbx::slice("value")); // MDBX_APPEND
    txn.commit();
    const auto t1 = std::chrono::steady_clock::now();
    return uint64_t(std::chrono::duration_cast<std::chrono::microseconds>(t1 - t0).count());
}
```

Full code: [30-append-prefault.c++](#).

6.3.8 23.8. When these optimizations matter, and when they do not

SIMD/CMOV give from a few percent to tens of percent on specific paths. For most applications what decides is **batching and the sync mode**, not micro-optimizations. Do not optimize blindly — measure.

Examples for this chapter: [examples/c++/30-append-prefault.c++](#).

6.3.9 23.9. Chapter 23 summary

- Internal optimizations (SIMD, CMOV, radix, atomics) are already built in.
- Your levers: prefault, auto-append, merge policy.
- Micro-optimizations are secondary to batching and the sync mode.

6.3.10 23.10. Exercises

1. When is enabling prefault write justified? State the condition (DB size vs RAM).
2. Why is "optimizing without measuring" an anti-pattern?

6.3.11 23.11. Chapter 23 checklist

- [] I know which micro-optimizations are already built in (SIMD kernels, CMOV, radix-sort, C11 atomics), and that the SIMD search is influenced only via `rp_augment_limit` / `gc_time_limit`.
- [] I can state the condition under which prefault write is justified: DB > RAM, frequent page evictions, `MDBX_WRITEMAP`.
- [] I understand the benefit of the auto-appending split for append scenarios and of merging with an already-dirty neighbor (related to `prefer_waf_insteadof_balance`).
- [] I know that hot paths minimize syscalls (bulk iovec, bulk writes in `mdbx_copy`).
- [] I can explain why batching and the sync mode matter more than micro-optimizations, and why optimizing must follow measurements.

6.4 Chapter 24. Cache lookup (get-cached)

6.4.1 24.1. What is get-cached

`mdbx_cache_get()` — a read service for "recurring" keys. For a key, the address of the value in mmap + version tags are stored. On a repeated read, instead of a full descent of the B+tree a **lazy (early) exit** is performed: we descend only while we encounter pages modified after the last confirmation.

6.4.2 24.2. Mechanism

Cache entry: `{trunk_txnid, last_confirmed_txnid, offset, length}` (`offset == 0` = "key absent").

The first call is a full search + filling the entry. Subsequent ones:

1. the descent stops at the first page not modified after `last_confirmed_txnid`;
2. if nothing changed — `MDBX_CACHE_HIT` after a few cheap comparisons;
3. if there are changes — confirmation of freshness (`MDBX_CACHE_CONFIRMED`) or a full search (`MDBX_CACHE_REFRESHED`).

6.4.3 24.3. Statuses

Status	Meaning
HIT / CONFIRMED / REFRESHED	The result is obtained
DIRTY	The value is on a dirty (uncommitted) page; valid only in the current write transaction
BEHIND	Reader is older than the version in the cache; bypass the cache
UNABLE	ABA situation; search bypassing the cache
RACE	Concurrent update of the entry; search bypassing
ERROR (<code>MDBX_CACHE_ERROR</code>)	An error unrelated to the key being absent

Fragment from [examples/c++/31-get-cached.c++](#) — displaying all `get_cached` cache statuses: HIT / CONFIRMED / REFRESHED, DIRTY, BEHIND, UNABLE, RACE and ERROR :

```
const char *status_name(int status) {
    switch (status) {
        case MDBX_CACHE_ERROR:
            return "ERROR";
        case MDBX_CACHE_BEHIND:
            return "BEHIND";
        case MDBX_CACHE_UNABLE:
            return "UNABLE";
        case MDBX_CACHE_RACE:
            return "RACE";
        case MDBX_CACHE_DIRTY:
            return "DIRTY";
        case MDBX_CACHE_HIT:
            return "HIT";
        case MDBX_CACHE_CONFIRMED:
            return "CONFIRMED";
        case MDBX_CACHE_REFRESHED:
            return "REFRESHED";
        default:
            return "?";
    }
}
```

Full code: [31-get-cached.c++](#).

6.4.4 24.4. When it gives a speedup

The cache is effective for **recurring** keys with a small number of changes (configuration, counters, markers, lookaside tables). For unique keys there is no benefit (the first search is full anyway). The ratio of a full search to the HIT path easily reaches 10^2 – 10^3 .

6.4.5 24.5. When it does not work

- Unique keys (no repeats).
- Active write transactions yield `DIRTY` (not cached).
- Multithreading requires `MDBX_NOSTICKYTHREADS` for the multithreaded variant; the single-threaded variant (`mdbx_cache_get_SingleThreaded`) is cheaper.

6.4.6 24.6. History

The request for a multithreaded lockfree get-cached — Gabriel RABHI ("Ghost Body Object"). The multithreaded variant is implemented; the status on the current canon — to be clarified as the API evolves.

Examples for this chapter: [examples/c++/31-get-cached.c++](#).

6.4.7 24.7. Chapter 24 summary

- get-cached is a lazy cache with an early exit and version tags.
- HIT/.../RACE are result statuses; `DIRTY` in write transactions.
- Effective for recurring "hot" keys; up to 1000× faster than a search.
- The multithreaded variant requires `NOSTICKYTHREADS`.

6.4.8 24.8. Exercises

1. Estimate: a loop of 1 000 000 reads of one key — how much will the cache save?
2. Why is `DIRTY` not cached?

6.4.9 24.9. Chapter 24 checklist

- [] I describe the cache entry `{trunk_txnid, last_confirmed_txnid, offset, length}` and the meaning of `offset == 0`.
- [] I can explain the lazy-exit mechanics: the descent stops at the first page not modified after `last_confirmed_txnid`.
- [] I distinguish the statuses `HIT / CONFIRMED / REFRESHED / DIRTY / BEHIND / UNABLE / RACE / ERROR` and know when the search bypasses the cache.
- [] I understand that `DIRTY` is valid only within the current write transaction and is not cached.
- [] I can estimate when the cache gives a speedup (recurring "hot" keys) and when it does not (unique keys).
- [] I know the `MDBX_NOSTICKYTHREADS` requirement for the multithreaded variant and the cheaper single-threaded variant (`mdbx_cache_get_SingleThreaded`).

6.5 Chapter 25. Bulk operations

6.5.1 25.1. bunch_delete: deleting in bunches

`mdbx_cursor_bunch_delete()` with an action mode (`MDBX_bunch_action_t`) deletes the current value / the whole multivalue / everything before or after the position / everything in a row. Instead of iterating over elements the engine **cuts out whole pages and branches** — the cost is proportional to the number of pages, not elements.

6.5.2 25.2. get_batch: batched reading

`mdbx_cursor_get_batch()` — symmetric bulk reading in batches (bulk iovec).

6.5.3 25.3. GET/PUT_MULTIPLE

Working with multivalues in batches: `MDBX_GET_MULTIPLE / MDBX_PUT_MULTIPLE` for DUPFIXED tables.

6.5.4 25.4. estimate_range/distance/move

Heuristic estimation without scanning: `mdbx_estimate_range()`, `mdbx_estimate_distance()`, `mdbx_estimate_move()`. It is built from the common pages of the stacks of two positions. Accuracy: in the worst case a deviation of up to **4× at each level** of the tree (except the first and last); in practice — a few percent.

Fragment from `examples/c++/32-bulk-ops.c++` — bulk operations: `get_batch` (reading in batches), `estimate` (estimation of the number of records) and `bunch_delete` (deleting in bunches with page cutting-out):

```
// Batched reading: up to 100 pairs per call.
size_t pairs = 0, chunks = 0;
cur.to_first();
bool is_last = false;
while (!is_last) {
    auto batch = cur.get_batch(100, mdbx::cursor::move_operation::next, &is_last);
    pairs += batch.size();
    ++chunks;
}
std::cout << "batch reads: " << pairs << " pairs in " << chunks << " chunks\n";

// Estimation of the number of records (approximately).
cur.to_first();
auto est = cur.estimate(mdbx::cursor::move_operation::last);
std::cout << "estimate: ~" << est.approximate_quantity << " keys\n";

// Deleting in bunches: everything from the current position to the end.
cur.to_key_exact(mdbx::slice("k500"));
const size_t removed = cur.erase_bunch(mdbx::cursor::bunch_delete::delete_after_including);
std::cout << "bunch delete removed " << removed << " items\n";
```

Full code: [32-bulk-ops.c++](#).

6.5.5 25.5. txn_clone: parallel processing on a single snapshot

`mdbx_txn_clone()` multiplies a read-only transaction: several handlers on a single snapshot without re-scanning the RLT.

6.5.6 25.6. Transaction batching

100–10 000 operations per commit is a typical range for write-heavy. More operations → lower WAF and higher throughput.

6.5.7 25.7. When a "maximum-size transaction" is an anti-pattern

- A transaction with millions of operations can overflow the dirty-page limits (`TXN_FULL`), cause a spill and a latency spike.
- A long write transaction blocks other writers and widens the risk window.
- Split into reasonable batches (e.g., 10 000 operations each) — a balance between WAF and latency.

Examples for this chapter: [examples/c++/32-bulk-ops.c++](#).

6.5.8 25.8. Chapter 25 summary

- `bunch_delete/delete_range` — bulk deletion by cutting out pages.
- `get_batch`, `GET/PUT_MULTIPLE` — batched read/write.
- `estimate_*` — cheap range estimation (up to 4×/level in the worst case).
- `txn_clone` — parallel processing on a single snapshot.
- A reasonable batch is better than a "maximum-size transaction".

6.5.9 25.9. Exercises

1. Compare row-by-row range deletion and `bunch_delete` on 100 000 keys.
2. When is `estimate_range` useful before running a query?

6.5.10 25.10. Chapter 25 checklist

- [] I can explain why `bunch_delete` is cheaper than iterating over elements: whole pages and branches are cut out, the cost is proportional to the number of pages.
- [] I know the symmetric bulk reads/writes: `get_batch`, `MDBX_GET_MULTIPLE` / `MDBX_PUT_MULTIPLE` (DUPFIXED tables).
- [] I understand how `estimate_*` works: estimation from the common pages of the stacks of two positions, up to 4× per tree level in the worst case.
- [] I can use `txn_clone` for parallel processing on a single snapshot.
- [] I can name the consequences of a "maximum-size transaction": `TXN_FULL`, spill, a latency spike, blocking other writers.
- [] I choose a reasonable batch size (e.g., ~10 000 operations) as a balance between WAF and latency.

6.6 Chapter 26. Benchmarks and measurements

6.6.1 26.1. ioarena

`ioarena` — the project's standard benchmark pipeline. It runs on ready-made configurations; it shows write/read TPS for different engines and modes.

6.6.2 26.2. What to measure

- **Write TPS** — commits/sec in the sync modes.
- **Read get/s** — operations/sec.
- **WAF** — via page counters (`mdbx_stat -p`).
- **Commit latency** — `mdbx_txn_commit_ex()` → `MDBX_commit_latency` by stages (`preparation/gc_wallclock/audit/write/sync/ending/whole`).

6.6.3 26.3. MDBX_ENABLE_PROFGC

GC profiling: collected in the LCK-file, returned in `commit_latency.gc_prof`. Key fields: `work_rtime_monotonic` / `work_xtime_cpu` (GC time), `work_rsteps` / `work_xpages` (fragmentation), `work_majflt` (page-faults), `max_reader_lag` / `max_retained_pages` (long readers), `kicks` (HSR — Handle-Slow-Readers, eviction of stuck readers; see Volume V, ch. 29).

Fragment from `examples/c++/33-profiler.c++` — reading PROFGC fields from `commit_latency.gc_prof` (`wloops`, `coalescences`, `flushes`, `kicks`, `max_reader_lag`, `max_retained_pages`); the fields are filled in builds with `MDBX_ENABLE_PROFGC`:

```
// Load that generates GC work: inserts and deletes.
for (int round = 0; round < 5; ++round) {
    auto txn = env.start_write();
    auto table = txn.open_map(nullptr);
    for (int i = 0; i < 500; ++i) {
        const auto key = "k" + std::to_string(round * 1000 + i);
        txn.insert(table, mdx::slice(key), mdx::slice("payload"));
    }
    txn.commit();
    auto del = env.start_write();
    auto dtable = del.open_map(nullptr);
    for (int i = 0; i < 500; ++i)
        del.erase(dtable, mdx::slice("k" + std::to_string(round * 1000 + i)));
    del.commit();
}

auto txn = env.start_write();
auto table = txn.open_map(nullptr);
txn.insert(table, mdx::slice("last"), mdx::slice("v"));
const auto lat = txn.commit_get_latency();

const auto &gc = lat.gc_prof;
std::cout << "gc_prof: wloops=" << gc.wloops << " coalescences=" << gc.coalescences << " flushes=" << gc.flushes
    << " kicks=" << gc.kicks << " max_reader_lag=" << gc.max_reader_lag
    << " max_retained_pages=" << gc.max_retained_pages << "\n";
```

Full code: [33-profiler.c++](#).

6.6.4 26.4. Typical benchmarking mistakes

- **Asserts:** a debug build (`MDBX_CHECKING>0`) slows things down manifold.
- **Page-cache incoherence** (#269) on Linux with several processes.
- **Methodology:** "small transactions + DURABLE on Windows" is LockFileEx, not the engine.
- **Comparison without context:** mode, page, tree height, hardware.

6.6.5 26.5. Why libmdbx may be "6–7× slower than LMDB"

Usually it means: asserts are enabled, or an incorrect benchmark (small transactions on Windows with LockFileEx), or a single-op mode is measured, where libmdbx deliberately pays for stricter guarantees (two-phase meta, checks). With a correct methodology and the same modes the gap disappears or turns in libmdbx's favor.

6.6.6 26.6. Guidelines

- Write: ~200 TPS of single durable transactions on an ordinary SSD; with batching and `SAFE_NOSYNC` — tens– hundreds of thousands and millions of operations/s.
- Read: 1–3 million gets/s (short keys, warm cache); linear scaling across cores.
- Inserts: 20K–10M/s depending on the mode and hardware.

Warning: always provide the context of a number — sync mode, page size, transaction size, hardware.

6.6.7 26.7. Read scaling

Reading is wait-free and scales linearly across cores — until memory saturates (bandwidth/TLB). The bottleneck is usually not the engine but the cache/memory.

Examples for this chapter: `examples/c++/33-profiler.cpp`.

6.6.8 26.8. Chapter 26 summary

- ioarena — the standard benchmark; TPS/get/s/WAF/latency.
- `commit_latency` + `PROFGC` — bottleneck diagnostics.
- Asserts and methodology are the main traps.
- "6–7× slower than LMDB" is almost always a measurement error.
- Guidelines: 200 TPS durable, 1–3M gets/s, 20K–10M inserts/s.

6.6.9 26.9. Exercises

1. Measure `commit_latency` on your configuration and identify the dominant stage.
2. Explain why the "one transaction per operation" comparison is incorrect for evaluating the engine.

6.6.10 26.10. Chapter 26 checklist

- [] I know what to measure: write TPS, read get/s, WAF (via page counters), commit latency by stage.
- [] I can read `MDBX_commit_latency` (`preparation/gc_wallclock/audit/write/sync/ending/whole`) and identify the dominant stage.
- [] I understand what the `gc_prof` fields (`max_reader_lag` , `max_retained_pages` , `kicks`) say about long readers and HSR kicks.
- [] I avoid the typical mistakes: measuring a build with asserts, "small transactions + DURABLE on Windows", comparison without context.
- [] I can explain why "6–7× slower than LMDB" is almost always a measurement error or the price of stricter guarantees.
- [] I know the guidelines (~200 TPS durable, 1–3M gets/s) and always provide the context of a number.

- [] I can explain why reads scale linearly across cores and where the real bottleneck is (memory/cache, not the engine).
-

6.7 Volume summary

You know how to measure and tune libmdbx: you understand WAF, choose a configuration for the scenario, use micro-optimizations, get-cached and bulk operations, and read the GC profile and latency.

What's next: Volume V — expert topics: rules and tips from the knowledge base, platform nuances, HSR, diagnostics and debugging, migration from LMDB, design patterns, roadmap.

libmdbx v0.15.0-150-g6ca39a94 — built 2026-10-19 · [online version](#) · [PDF version](#)

7. Volume V. Expert topics and edge cases

Level: for advanced users. **Volume goal:** you know the subtleties and pitfalls, can debug problems, make architectural decisions, migrate from LMDB and build robust patterns.

7.1 Chapter 27. Rules and tips from the knowledge base

The full list of rules and tips is in the knowledge base ([knowledge-base/rules-and-checklist/](#), [knowledge-base/tips/](#)). Here — grouping by topic with criticality accents.

7.1.1 27.1. Transactions and threads (critical)

- **One transaction — one thread.** Passing a transaction object to another thread is UB/an error (`MDBX_THREAD_MISMATCH`).
- **Do not pass a transaction between threads** even with `NOSTICKYTHREADS`, if another thread may synchronously call a write-function of the environment — deadlock.
- **Do not use transaction objects from two threads at the same time.**
- After `commit / abort` pointers to data in the mmap are invalid.

7.1.2 27.2. Lifecycle and integrity

- Do not open the database twice in one process; `fork()` — only with `resurrect_after_fork` in the child.
- An unclosed read-transaction "freezes" page reclamation — close/park it.
- Never copy the database file "on the fly" — use `mdbx_copy`.
- `MDBX_UTTERLY_NOSYNC` — only for non-critical data.

7.1.3 27.3. Space and growth

- Set the geometry once before open; do not lower `upper`.
- Lower `rp_augment_limit` — only together with `gc_time_limit`.
- Do not use `MDBX_ENABLE_REFUND=0` in production (debug option).
- Long values: increase the page size (up to 64 KB), split records.
- A value spanning several overflow pages needs a run of **consecutive** pages: when large values are rewritten often, watch fragmentation (`mdbx_chk` histograms, overflow page counts in `mdbx_env_stat`); for "heavy" data consider storing it outside the database.
- If a long value must be kept in the DB, split it into chunks that fit into a single page (`mdbx_limits_valsize4page_max()`): in a plain table — a chunk-number suffix in the key, in a DUPSORT table — a prefix with the multivalue number and the chunk number. Manual splitting is needed only until the planned **streamed API for long values** ("Streaming BLOBs") arrives. Details — [Restrictions & gotchas](#).

7.1.4 27.4. Operations

- Do deletions before insertions in the same transaction; mass deletions — `bunch_delete / delete_range`.
- Do not build a `std::map` over the database for ordering — the gain is usually less than the cost.
- For DUPSORT deletion while iterating — use two cursors.

7.1.5 27.5. Typical counter-examples

- Violating "one transaction — one thread" → random `BAD_RSL0T` /overlaps.
- Opening the database twice → races and corrupted registrations.
- A forgotten reader → the file grows to `MDBX_MAP_FULL`.
- A debug build in production → multiple slowdowns.

Fragment from `examples/c++/34-rules-counter.c++` — counter-example of a "forgotten" read snapshot: a thread opens a snapshot (`env.start_read()`) and holds it (`abort()` only via an external flag); while the reader is alive, the writer hits `MDBX_MAP_FULL` (GC is frozen, the file grows):

```
std::atomic<bool> reader_ready{false};
std::atomic<bool> release_reader{false};
std::thread holder([&] {
    auto rtxn = env.start_read();
    auto table = rtxn.open_map(nullptr);
    (void)rtxn.get(table, mdbx::slice("k0"));
    reader_ready = true;
    while (!release_reader.load())
        std::this_thread::yield();
    rtxn.abort();
});
while (!reader_ready.load())
    std::this_thread::yield();
```

Full code: [34-rules-counter.c++](#).

Examples for this chapter: [examples/c++/34-rules-counter.c++](#).

7.1.6 27.6. Summary of chapter 27

- One transaction — one thread: passing a transaction between threads is UB/an error (`MDBX_THREAD_MISMATCH`), even with `NOSTICKYTHREADS` a deadlock is possible.
- An unclosed read-transaction "freezes" page reclamation — the file grows to `MDBX_MAP_FULL`; close/park it.
- Do not open the database twice in one process; `fork()` — only with `resurrect_after_fork` in the child; copying — only `mdbx_copy`.
- Do not use `MDBX_UTTERLY_NOSYNC` or debug options (`MDBX_ENABLE_REFUND=0`) in production.
- Set the geometry once before open, do not lower `upper`; for long values increase the page size (up to 64 KB), split into chunks that fit into one page, or store outside the DB; values that do not fit one page need a contiguous run and are sensitive to fragmentation — until the streaming API ("Streaming BLOBs") arrives.
- Do deletions before insertions; mass deletions — `bunch_delete` / `delete_range`; DUPSORT deletion while iterating — two cursors.
- Typical counter-examples: violating thread discipline (`BAD_RSL0T`), double opening, a forgotten reader, a debug build in production.

7.1.7 27.7. Chapter 27 checklist

- [] Ensure transactions are not passed between threads and are not used from two threads simultaneously.
- [] Close/park read-transactions immediately after use.
- [] Copy the database file only via `mdbx_copy`.
- [] Set the geometry before open and do not lower `upper`.
- [] Exclude `MDBX_UTTERLY_NOSYNC` and `MDBX_ENABLE_REFUND=0` from production configuration.
- [] Perform deletions before insertions; mass deletions — via `bunch_delete` / `delete_range`.
- [] For DUPSORT deletion while iterating use two cursors.

7.2 Chapter 28. Platform-specific notes

7.2.1 28.1. Linux

- `boot_id` is taken into account when rolling back weak metas; in LXC — shared/missing.
- `/dev/shm` (tmpfs): create with headroom; `ENOSPC` from `fallocate()` can be ignored; since 0.13.8 `fallocate` protects against `SIGBUS`.
- Page cache: unified page cache incoherency (#269) — `MDBX_FORCE_CHECK_MMAP_COHERENCY`.
- `mincore` / `madvise(MADV_NOHUGEPAGE)` — opting out of THP.
- Linux < 4.x: `mdbx_env_create()` → `MDBX_INCOMPATIBLE`.

7.2.2 28.2. Windows

- `LockFileEx` is slower than named mutexes; small transactions are expensive.
- 32-bit: address-space limits; `/LARGEADDRESSAWARE` (+1 GB to `MAX_MAPSIZE32`).
- File compaction — only in a single-process scenario; extension via Native API; geometry changes suspend threads (SRWL — the Windows Slim Reader/Writer Lock, Windows' fine-grained locking primitive).
- WSL1: `ENOLCK` — operation is fundamentally impossible.

7.2.3 28.3. macOS / iOS

- `fcntl(F_FULLFSYNC)` by default (maximum durability); `MDBX_APPLE_SPEED_INSTEADOF_DURABILITY` — sacrifice durability for speed.
- SysV semaphores by default (flavour `SYSV`).
- `fcntl(F_PREALLOCATE)` reserves at the end — the DB grows at twice the standard rate.

7.2.4 28.4. Android / bionic

- TLS and atomics differ from glibc; 32-bit Android — a possible hang when rolling back a commit (historically; no problem since 0.11.7).

7.2.5 28.5. Containers

- LXC: `boot_id`; Docker — PID uniqueness.
- tmpfs: `ENOSPC` from `fallocate` (ignore).
- NFS/CIFS/SMB: only exclusive mode or cooperative read-only; copying works.

7.2.6 28.6. Wine

- Mechanisms of dynamic database resizing are not implemented — set a fixed sufficient size; `mdbx_module_handler(...)` at startup for static linking.

Fragment from [examples/c++/35-platform-notes.c++](#) — a platform caveat in code: an "absurd" geometry request is handled via try/catch, because on tmpfs/WSL1/32-bit Windows `ENOSPC` / `ENOLCK` is a normal reaction, not a fatal failure:

```
// Checking geometry bounds must not crash: we request an "absurd"
// upper bound and carefully handle the result.
mdbx::env::geometry probe;
probe.size_upper = intptr_t(8) * mdbx::env::geometry::TB;
try {
    env.set_geometry(probe);
    std::cout << "geometry probe: accepted\n";
} catch (const std::exception &ex) {
    // On tmpfs/WSL1/32-bit Windows this may fail —
    // for the example it matters that it is caught, not that it crashes the process.
    std::cout << "geometry probe: rejected (" << ex.what() << ")\n";
}
```

Full code: [35-platform-notes.c++](#).

Examples for this chapter: [examples/c++/35-platform-notes.c++](#).

7.2.7 28.7. Summary of chapter 28

- Platform differences concern locking, synchronization and address-space sizes.
- LXC/boot_id and page cache — container/Linux specifics.
- Windows LockFileEx and F_FULLFSYNC on macOS — the main platform pitfalls.

7.2.8 28.8. Chapter 28 checklist

- [] Linux: account for boot_id (LXC), tmpfs ENOSPC from fallocate, opt out of THP if needed.
- [] Windows: remember expensive small transactions and 32-bit limits (/LARGEADDRESSAWARE).
- [] macOS: consciously choose between fcntl(F_FULLFSYNC) and MDBX_APPLE_SPEED_INSTEADOF_DURABILITY.
- [] Android/bionic: verify behavior on 32-bit (the historical hang when rolling back a commit — before 0.11.7).
- [] Containers: LXC — boot_id, Docker — PID uniqueness; NFS/CIFS/SMB — only exclusive or cooperative read-only.
- [] Wine: set a fixed sufficient database size (dynamic resizing does not work).
- [] Treat ENOSPC / ENOLCK on tmpfs/WSL1/32-bit Windows as normal errors, not fatal failures.

7.3 Chapter 29. Handle-Slow-Readers (HSR)

7.3.1 29.1. What HSR is and why it is needed

When the database is full **because of long readers** (GC is frozen), the library calls the HSR callback — the only standard way for the application to influence the outcome of the "long reader versus writer" conflict.

7.3.2 29.2. Signature and parameters

```
typedef int (*MDBX_hsr_func)(const MDBX_env *env, const MDBX_txn *txn,
                             mdbx_pid_t pid, mdbx_tid_t tid,
                             uint64_t laggard, unsigned gap,
                             size_t space, int retry);
```

- laggard — the lag of the problematic reader;
- gap — the number of attempts;
- space — the volume that will be freed after the reader finishes;
- retry — the counter of repeated invocations.

7.3.3 29.3. Return values

Value	Action
0	Callback waited/decided; libmdbx re-scans RLT and retries
1	Reader transaction aborted asynchronously; clear the slot immediately
2+	Reader process killed; libmdbx clears its registration

Called only when the database is full because of readers.

7.3.4 29.4. Typical implementations

- **Logging** + return 0 (wait).
- **Graceful shutdown**: set a flag for the reader; return 1.
- **Forced**: kill the reader process, return 2.

- **Agreement to grow:** if geometry allows — increase `upper` and return 0.

7.3.5 29.5. HSR and SAFE_NOSYNC

In `SAFE_NOSYNC` a "quasi-long reader" is a steady-commit, not a live slot: HSR does not affect it. Growth must be managed via auto-sync (`syncbytes / syncperiod`).

Fragment from `examples/c++/36-hsr.c++` — the HSR callback: reports the problematic reader (pid/lag/space), asks it to release the snapshot and waits for the actual release, then returns `MDBX_RESULT_TRUE`. Registration — `env.set_HandleSlowReaders(hsr_callback)`; together with `SAFE_NOSYNC` the callback remains the main tool against long readers:

```
// The callback is invoked when the DB "hits" the limit because of readers
// holding old snapshots. Here: report the event, ask the reader to
// release the snapshot and wait for it, then return MDBX_RESULT_TRUE
// ("problem resolved" — the writer will continue).
int hsr_callback(const MDBX_env *, const MDBX_txn *, mdbx_pid_t pid, mdbx_tid_t tid, uint64_t laggard,
                unsigned gap, size_t space, int retry) noexcept {
    std::cout << "hsr invoked: laggard pid=" << pid << " tid=" << tid << " lag=" << laggard << " gap=" << gap
                << " space=" << space << " retry=" << retry << "\n";
    hsr_called = true;
    hsr_release_reader = true;
    while (!reader_released.load())
        std::this_thread::yield(); // wait for the actual release
    return MDBX_RESULT_TRUE;
}
```

Full code: [36-hsr.c++](#).

7.3.6 29.6. Example implementation

Fragment (C, illustration): the full compilable version is in [examples/c++/36-hsr.c++](#).

```
static int my_hsr(const MDBX_env *env, const MDBX_txn *txn,
                mdbx_pid_t pid, mdbx_tid_t tid,
                uint64_t laggard, unsigned gap,
                size_t space, int retry) {
    fprintf(stderr, "HSR: reader pid=%d lag=%llu space=%zu retry=%d\n",
            (int)pid, (unsigned long long)laggard, space, retry);
    if (retry > 3)
        return 2; /* kill the problematic reader */
    return 0; /* wait more */
}
/* setup */
mdbx_env_set_hsr(env, my_hsr);
```

Examples for this chapter: [examples/c++/36-hsr.c++](#).

7.3.7 29.7. Summary of chapter 29

- HSR — a callback for resolving conflicts with long readers.
- Parameters: `laggard/gap/space/retry`; returns 0/1/2+.
- Typical strategies: wait/kill/grow.
- With `SAFE_NOSYNC` manage growth via auto-sync.

7.3.8 29.8. Chapter 29 checklist

- [] Install the HSR callback (`mdbx_env_set_hsr / set_HandleSlowReaders`).
- [] Define the policy per `retry`: logging, graceful shutdown, forced kill, agreement to grow.
- [] Use the `laggard / gap / space / retry` parameters to make the decision in the callback.
- [] Remember the return semantics: 0 — wait, 1 — abort the reader, 2+ — kill the reader process.
- [] Under `SAFE_NOSYNC` configure auto-sync (`syncbytes / syncperiod`) — HSR does not affect a steady-commit.

7.4 Chapter 30. Diagnostics and debugging

7.4.1 30.1. Tools

`mdbx_chk` — integrity check. Key options:

Option	What it does
<code>-v...-vvvvv</code>	Output verbosity
<code>-q</code>	Complete silence
<code>-c</code>	Cooperative (non-exclusive) mode; full check — only exclusive
<code>-w</code>	Read-write: rollback to steady, check meta txnids
<code>-d</code>	Page-by-page B+tree walk; without it lost/double-used pages cannot be found
<code>-i</code>	Ignore false order errors (custom comparators)
<code>-s table</code>	Check only the table
<code>-0/-1/-2</code>	Specific meta; <code>-t / -T</code> — switch to it

Exit code: 0 = no errors.

`MDBX_ENABLE_PROFGC` — GC profile in `commit_latency.gc_prof`. Key fields: `max_reader_lag`, `max_retained_pages`, `work_rtime_monotonic / work_xtime_cpu`, `work_rsteps / work_xpages`, `work_majflt`, `self_*`, `wloops`, `flushes`, `kicks`.

`MDBX_commit_latency` — commit stages: `preparation`, `gc_wallclock`, `audit`, `write`, `sync`, `ending`, `whole`, `gc_cputime`.

`mdbx_txn_info()` — `txn_reader_lag`, `txn_space_used/limit_soft/limit_hard/retired/leftover/dirty`. For long readers `txn_space_retired` and `txn_space_leftover` matter.

Debug build — `MDBX_DEBUG / MDBX_CHECKING` (~1..3); `MDBX_FORCE_ASSERTIONS` deprecated. Sanitizers: `ENABLE_ASAN / ENABLE_UBSAN / ENABLE_MEMCHECK`; TSAN is not in CMake (manually).

Fragment from `examples/c++/37-diagnostics.c++` — reading `mdbx_txn_info` (via `txn.get_info(...):id/lag/used/dirty`) and the reader list (`mdbx_reader_list`, here `env.enumerate_readers(...)`) — the very data for the "DB growth" and `MDBX_MAP_FULL` algorithms above:

```
// Information about the current writing transaction.
const auto info = txn.get_info(true /* scan_rlt */);
std::cout << "txn_info: id=" << info.txn_id << " lag=" << info.txn_reader_lag << " used=" << info.txn_space_used
          << " dirty=" << info.txn_space_dirty << "\n";
txn.commit();
}

// List of readers (RLT).
struct visitor {
    int operator()(const mdbx::env::reader_info &ri, int) {
        std::cout << "reader slot=" << ri.slot << " pid=" << ri.pid << " tid=" << ri.thread
                  << " txnid=" << ri.transaction_id << " lag=" << ri.transaction_lag << "\n";
        return mdbx::continue_loop;
    }
} v;
env.enumerate_readers(v);
```

Full code: [37-diagnostics.c++](#).

7.4.2 30.2. Step-by-step algorithm: DB growth

1. `mdbx_env_info_ex()` — metas and geometry (`size_now` vs `size_upper`).
2. `mdbx_reader_list()` / `mdbx_reader_check()` — who holds snapshots.
3. `mdbx_stat -r` — retained.
4. PROFGC: `max_reader_lag`, `max_retained_pages`, `kicks`.
5. `mdbx_stat -p` — `newly` vs `cow` (new pages instead of reuse).

6. Decision: parking/HSR/steady-point/defragmentation; under `SAFE_NOSYNC` — auto-sync.

7.4.3 30.3. Step-by-step algorithm: commit slowdown

1. `mdbx_txn_commit_ex()` — decompose `whole` into stages.
2. `sync` dominates → durability mode; too frequent `fsync` → batch or `NOMETASYNC` / `SAFE_NOSYNC`.
3. `gc_wallclock` is large → PROFGC: `work_rsteps` / `work_xpages` / `work_majflt` (fragmentation/large values).
4. `write` is large with small data → early spill (`dp_limit`) or duplicated dirty pages.
5. `WITEMAP` + page-faults → `prefault_write_enable`.
6. `mdbx_stat -p` — `msync/fsync` counters.

7.4.4 30.4. Step-by-step algorithm: MDBX_MAP_FULL

1. **Abort the current write-transaction** (continuation after `resize` → `BAD_TXN`).
2. `mdbx_txn_info()` — `txn_space_limit_hard` vs used.
3. `mdbx_reader_list()` — old snapshots pin space.
4. PROFGC — `kicks`, `max_reader_lag`, `max_retained_pages`.
5. Check HSR (`mdbx_env_set_hsr`), parking, geometry (`upper` lowered?).
6. Decision: HSR (wait/kill/grow), evict parked, new steady, increase `upper` before open, defragmentation.

7.4.5 30.5. Step-by-step algorithm: deadlock / MDBX_BUSY

1. Check thread discipline: codes `THREAD_MISMATCH` / `TXN_OVERLAPPING` / `BAD_RSLT`.
2. Under `NOSTICKYTHREADS` — exclude synchronous write-functions (`set_option/set_flags/set_geometry/sync/stat/defrag/close`) from a thread not owning the writing transaction.
3. Audit `fork()` (resurrect) and repeated `env_open`.
4. `mdbx_reader_list()` will show foreign owners.

7.4.6 30.6. Step-by-step algorithm: DB corruption

1. `mdbx_chk -w -vvv` — localize (meta? tree? GC? order?).
2. Check the environment: `boot_id` (LXC?), page cache incoherency (#269, `incoherence`).
3. One meta corrupted → `mdbx_chk -1/-2 + -T` (switch to a valid one).
4. `MDBX_WANNA_RECOVERY` → read-write or `mdbx_env_open_for_recovery()` (since 0.12.7 it does not modify the database).
5. Recovery checks are safe; restore from backup (`mdbx_copy`).

7.4.7 30.7. Step-by-step algorithm: reader slot leak

1. `mdbx_reader_check(env, &dead)` — how many dead slots.
2. Cause: threads terminated without cleanup (TLS destructor; glibc #21031/#21032; DSO unload).
3. Solution: explicit `mdbx_thread_register/unregister`; the `reset+renew` pattern; `resurrect_after_fork`.

7.4.8 30.8. Checklists before production deployment

- [] Geometry set before open (`upper` adequate; the engine itself chooses the default $\approx$ golden ratio of RAM).
- [] HSR callback installed.
- [] Sync mode consciously chosen.
- [] Release build without asserts.

- [] `maxreaders` covers the number of threads.
- [] Long reads use parking/reset+renew.
- [] Post-crash test: `kill -9 + open + mdbx_chk`.

Examples for this chapter: [examples/c++/37-diagnostics.c++](#).

7.4.9 30.9. Summary of chapter 30

- Tools: `chk`, `PROFGC`, `commit_latency`, `txn_info`, `reader_check`, debug build.
- 6 step-by-step algorithms: growth, commit slowdown, `MAP_FULL`, deadlock, corruption, slot leak.
- Pre-deployment checklist is mandatory.

7.4.10 30.10. Chapter 30 checklist

- [] Master `mdbx_chk` (options `-d`, `-w`, `-0/-1/-2/-T`) as the main integrity tool.
- [] Diagnose DB growth per the algorithm in §30.2 (`mdbx_reader_list`, `mdbx_stat -r/-p`, `PROFGC`).
- [] Decompose commit slowdown into `MDBX_commit_latency` stages (`mdbx_txn_commit_ex`) — §30.3.
- [] On `MDBX_MAP_FULL` — abort the transaction, check readers, HSR and geometry — §30.4.
- [] Deadlock/`MDBX_BUSY` — audit thread discipline and `NOSTICKYTHREADS` — §30.5.
- [] DB corruption — `mdbx_chk -w -vvv`, restore from backup (`mdbx_copy`) — §30.6.
- [] Check reader slot leaks via `mdbx_reader_check` — §30.7.
- [] Pass the production-deployment checklist §30.8 (including `kill -9 + open + mdbx_chk`).

7.5 Chapter 31. Migrating from LMDB

7.5.1 31.1. Why migrate

`libmdbx` is a reworked LMDB: more durability modes, GC instead of free-list, stricter recovery guarantees, higher write performance with batching. **The data format is not compatible with LMDB:** the triple of meta pages, two-phase commit, checksums and the GC tree are own changes; LMDB files are not opened directly, migration is performed by transferring data.

7.5.2 31.2. Format compatibility

- **LMDB files are not opened** by `libmdbx` directly: the formats differ (triple of meta pages, two-phase commit, checksums, GC tree).
- **`libmdbx 0.11.x` ⇔ `0.12.x`** databases are mutually compatible (the format is frozen since v11.3).
- Migration from LMDB is a data transfer (`mdbx_dump / mdbx_load`, `mdbx_copy`), not a file rename.

7.5.3 31.3. Breaking changes

- `MDBX_NOLOCK` removed.
- `MDBX_NOTLS` → `MDBX_NOSTICKYTHREADS`.
- Other renames and new flag requirements — check against `mdbx.h`.

7.5.4 31.4. mdb_ → mdbx_ correspondence table

LMDB	libmdbx
<code>mdb_env_create</code>	<code>mdbx_env_create</code>
<code>mdb_env_open</code>	<code>mdbx_env_open</code>
<code>mdb_txn_begin</code>	<code>mdbx_txn_begin</code>
<code>mdb_dbi_open</code>	<code>mdbx_dbi_open</code>
<code>mdb_put/get/del</code>	<code>mdbx_put/get/del</code>
<code>mdb_cursor_get</code>	<code>mdbx_cursor_get</code>
<code>MDB_NOTLS</code>	<code>MDBX_NOSTICKYTHREADS</code>

Fragment from [examples/c/38-migration.c](#) — the `mdb_*` → `mdbx_*` correspondence map from the example header:

```
// Macro/name map for porting code from LMDB (the full table is in §31.4):
//  mdb_env_create   → mdbx_env_create
//  mdb_env_open     → mdbx_env_open
//  mdb_env_close    → mdbx_env_close
//  mdb_env_set_mapsize → mdbx_env_set_geometry (a different API!)
//  mdb_txn_begin    → mdbx_txn_begin
//  mdb_txn_commit   → mdbx_txn_commit
//  mdb_txn_abort    → mdbx_txn_abort
//  mdb_dbi_open     → mdbx_dbi_open
//  mdb_get/put/del  → mdbx_get/mdbx_put/mdbx_del
//  mdb_cursor_open/get → mdbx_cursor_open/mdbx_cursor_get
//  MDB_NOTLS       → MDBX_NOSTICKYTHREADS
//  MDB_NOSYNC      → MDBX_SAFE_NOSYNC or MDBX_UTTERLY_NOSYNC
//  MDB_APPEND      → MDBX_APPEND
//  MDB_INTEGERKEY   → MDBX_INTEGERKEY
//  mdb_strerror     → mdbx_strerror
```

Full code: [38-migration.c](#).

7.5.5 31.5. Behavioral differences

- Three metas vs two; two-phase commit.
- Page checksums.
- More diagnostic codes (`WANNA_RECOVERY`, `MVCC_RETARDED` and others).
- Empty keys/values are allowed.

7.5.6 31.6. Migration checklist

1. Make a backup (`mdbx_copy -c` or dump+load).
2. Open the database read-only — check with `mdbx_chk`.
3. Replace the calls per the correspondence table.
4. Run the tests under ASAN/UBSAN.
5. Tune geometry/modes for the new engine (do not blindly port LMDB settings).

7.5.7 31.7. Summary of chapter 31

- The libmdbx data format is not compatible with LMDB: the triple of meta pages, two-phase commit, checksums, GC tree; LMDB files are not opened directly.
- Migration is a data transfer (`mdbx_dump / mdbx_load`, `mdbx_copy`), not a file rename; libmdbx 0.11.x ↔ 0.12.x databases are mutually compatible (the format is frozen since v11.3).
- Breaking changes: `MDBX_NOLOCK` removed, `MDBX_NOTLS` → `MDBX_NOSTICKYTHREADS`; check other renames against `mdbx.h`.

- Function names change predictably (`mdb_env_create` → `mdbx_env_create`), but `mdb_env_set_mapsize` → `mdbx_env_set_geometry` is a different API.
- Behavioral differences: three metas vs two, page checksums, more diagnostic codes, empty keys/values allowed.
- Migration order: backup → read-only check with `mdbx_chk` → replace calls → tests under ASAN/UBSAN → re-tune geometry/modes.

7.5.8 31.8. Chapter 31 checklist

- [] Make a backup (`mdbx_copy -c` or dump+load) before any changes.
- [] Open the database read-only and verify it with `mdbx_chk`.
- [] Replace the calls per the correspondence table §31.4, including `mdb_env_set_mapsize` → `mdbx_env_set_geometry`.
- [] Replace the flags: `MDBX_NOLOCK` removed, `MDBX_NOTLS` → `MDBX_NOSTICKYTHREADS`.
- [] Run the tests under ASAN/UBSAN.
- [] Tune geometry and modes for libmdbx from scratch, do not blindly port LMDB settings.

Examples for this chapter: [examples/c/38-migration.c](#) (`mdb_` → `mdbx_` correspondence map — in the file header).

7.6 Chapter 32. Design patterns on libmdbx

7.6.1 32.1. Pattern 1: Key-value with auto-increment ID

There are two approaches to monotonic IDs: the built-in `mdbx_dbi_sequence()` (an atomic per-table counter) and the classic portable approach — a counter in a service table. The example below demonstrates the second variant: it works identically across all libmdbx versions and doesn't depend on `MDBX_LIFORECLAIM` subtleties.

Fragment from [examples/c++/39-pattern-sequence-id.c++](#) — auto-increment ID: the counter lives in the `meta` table and is incremented in the same writing transaction as the record insert:

```
uint64_t next_id(mdbx::txn_managed &txn, const mdbx::map_handle &meta) {
    constexpr auto counter_key = mdbx::slice("seq");
    uint64_t current = 0;
    try {
        current = txn.get(meta, counter_key).as_uint64();
    } catch (const mdbx::not_found &) {
        current = 0;
    }
    ++current;
    txn.upsert(meta, counter_key, mdbx::slice::wrap(current));
    return current;
}
```

Full code: [39-pattern-sequence-id.c++](#).

7.6.2 32.2. Pattern 2: Secondary index (DUPSORT)

Main table + "field → list of IDs" index (Volume II, chapter 8). Update in one transaction.

Fragment from [examples/c++/40-pattern-secondary-index.c++](#) — secondary index "field → list of IDs": the main table `users` and the index `by_role` (DUPSORT, `value_mode::multi`) are updated in one transaction:

```
auto txn = env.start_write();
// Main table: id → {name, role}.
auto users = txn.create_map("users", mdbx::key_mode::ordinal, mdbx::value_mode::single);
// Index: role → list of ids (multi-values).
auto by_role = txn.create_map("by_role", mdbx::key_mode::usual, mdbx::value_mode::multi);

struct rec {
    uint64_t id;
    const char *name;
    const char *role;
};
static const rec records[] = {{1, "alice", "admin"}, {2, "bob", "dev"}, {3, "carol", "admin"}};
for (const auto &r : records) {
    txn.upsert(users, mdbx::slice::wrap(r.id), mdbx::slice(std::string(r.name) + "|" + r.role));
    txn.upsert(by_role, mdbx::slice(r.role), mdbx::slice::wrap(r.id));
}
```

```

}
txn.commit();

```

Full code: [40-pattern-secondary-index.c++](#).

7.6.3 32.3. Pattern 3: Composite key

Concatenation of fields + comparator (or big-endian for numbers). Prefix search — `SET_RANGE`.

Fragment from [examples/c++/41-pattern-composite-key.c++](#) — composite key: fixed-width fields are packed into big-endian, for which the lexicographic order matches the numeric one (range search — `SET_RANGE`):

```

uint64_t pack(uint16_t year, uint16_t month) {
    const uint64_t value = (uint64_t(year) << 48) | (uint64_t(month) << 32);
    #if defined(__BYTE_ORDER__) && __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
        return __builtin_bswap64(value);
    #else
        return value;
    #endif
}

uint16_t decode_year(const mdbx::slice &key) {
    const auto *p = static_cast<const uint8_t *>(key.data());
    return uint16_t((uint16_t(p[0]) << 8) | p[1]);
}

uint16_t decode_month(const mdbx::slice &key) {
    const auto *p = static_cast<const uint8_t *>(key.data());
    return uint16_t((uint16_t(p[2]) << 8) | p[3]);
}

```

Full code: [41-pattern-composite-key.c++](#).

7.6.4 32.4. Pattern 4: Task queue (table-as-queue)

`MDBX_DUPSORT` + sequence: key — priority/number, value — task. Natural retrieval order.

Fragment from [examples/c++/42-pattern-queue.c++](#) — task queue: key — monotonic number (ordinal), the consumer takes the first element via a cursor and deletes it:

```

// Consumer: takes the first element via a cursor and deletes it.
auto txn = env.start_write();
auto queue = txn.open_map("queue", mdbx::key_mode::ordinal, mdbx::value_mode::single);
std::string dequeued;
auto cur = txn.open_cursor(queue);
while (true) {
    auto r = cur.to_first(false);
    if (!r)
        break;
    const auto id = r.key.as_uint64();
    const auto task = r.value.as_string();
    dequeued += std::to_string(id) + " " + std::string(task) + " ";
    cur.erase(false);
}
txn.commit();
std::cout << "dequeued: " << dequeued << "\n";

```

Full code: [42-pattern-queue.c++](#).

7.6.5 32.5. Pattern 5: Ring buffer

Bounded size: when exceeded, delete the oldest keys (cursor + `del`).

Fragment from [examples/c++/43-pattern-ring-buffer.c++](#) — ring buffer of the last N: fixed slots `0..N-1`, writing overwrites `counter % N`:

```

// Writing 12 elements into a buffer of 5 slots.
{
    auto txn = env.start_write();
    auto ring = txn.create_map("ring", mdbx::key_mode::ordinal, mdbx::value_mode::single);
    for (unsigned i = 0; i < 12; ++i) {
        const unsigned slot = i % kSlots;
        txn.upsert(ring, buffer::key_from_u64(slot), mdbx::slice("s" + std::to_string(i)));
    }
    txn.commit();
}

```

Full code: [43-pattern-ring-buffer.c++](#).

7.6.6 32.6. Pattern 6: Full-scan iterator

Cursor + `get_batch` / `bunch_delete` for bulk processing.

Fragment from [examples/c++/44-pattern-full-scan.c++](#) — full scan with the cursor `to_first()` → `to_next()` with a filter:

```
auto rtxn = env.start_read();
auto table = rtxn.open_map(nullptr);
auto cur = rtxn.open_cursor(table);

// Full scan with the "even keys" filter.
size_t scanned = 0, matched = 0;
for (auto r = cur.to_first(); r; r = cur.to_next(false)) {
    ++scanned;
    if (r.key.as_string().size() % 2 == 0)
        ++matched;
}
std::cout << "scanned " << scanned << " entries\n";
```

Full code: [44-pattern-full-scan.c++](#).

7.6.7 32.7. Pattern 7: Replication via per-table txnid

Change markers: store the transaction number of the last change; a secondary process polls and copies deltas.

Fragment from [examples/c++/45-pattern-replication.c++](#) — replication via per-table txnid: the value stores a version (`data|N`), the replica reads only records newer than its `last_seen`:

```
// The replica reads everything newer than its last txnid.
auto replicate = [&](uint64_t last_seen) -> size_t {
    auto rtxn = env.start_read();
    auto table = rtxn.open_map(nullptr);
    size_t n = 0;
    auto cur = rtxn.open_cursor(table);
    for (auto r = cur.to_first(); r; r = cur.to_next(false)) {
        if (version_of(r.value) > last_seen)
            ++n;
    }
    rtxn.abort();
    return n;
};
```

Full code: [45-pattern-replication.c++](#).

7.6.8 32.8. Pattern 8: Read-your-writes via clone / `embark_read`

`mdbx_txn_clone` / `embark_read` — own writes are visible immediately in the same logical operation without a separate transaction.

Fragment from [examples/c++/46-pattern-read-your-writes.c++](#) — read-your-writes: the same writing transaction immediately reads its own change; after `commit` it is also visible to a fresh reader:

```
// The writing transaction immediately reads its own change.
{
    auto txn = env.start_write();
    auto table = txn.open_map(nullptr);
    txn.insert(table, mdbx::slice("key"), mdbx::slice("v1"));
    std::cout << "within txn sees: " << txn.get(table, mdbx::slice("key")).as_string() << "\n";
    txn.commit();
}

// After the commit the change is visible to a fresh reader too.
{
    auto rtxn = env.start_read();
    auto table = rtxn.open_map(nullptr);
    std::cout << "after commit via fresh txn sees: " << rtxn.get(table, mdbx::slice("key")).as_string() << "\n";
    rtxn.abort();
}
```

Full code: [46-pattern-read-your-writes.c++](#).

Each pattern: problem statement → solution with code → trade-offs → alternatives (in the full version of the knowledge base — `knowledge-base/`).

7.6.9 32.9. Summary of chapter 32

- Auto-increment ID: a counter in a service table, incremented in the same write transaction as the insert; the alternative is the built-in `mdbx_dbi_sequence()`.
- A secondary (DUPSORT) index "field → list of IDs" is updated in one transaction with the main table.
- Composite key: fixed-width fields in big-endian (lexicographic order = numeric), range search — `SET_RANGE`.
- Task queue: an ordinal key (monotonic number), the consumer takes the first element via a cursor and deletes it.
- Ring buffer: fixed slots `0..N-1`, writing overwrites `counter % N`.
- Full scan — cursor `to_first()` → `to_next()`, bulk processing — `get_batch` / `bunch_delete`.
- Replication via per-table txnid (a version marker in the value) and read-your-writes via `mdbx_txn_clone` / `embark_read`.

7.6.10 32.10. Chapter 32 checklist

- [] For monotonic IDs choose `mdbx_dbi_sequence()` or a counter in a service table (the portable variant).
- [] Update the secondary index (DUPSORT, `value_mode::multi`) in the same transaction as the main record.
- [] For composite keys use big-endian/a comparator and range search via `SET_RANGE`.
- [] Implement the task queue via an ordinal key and deletion of the first element by a cursor.
- [] For bounded structures (ring buffer) delete the oldest records on overflow.
- [] Perform bulk scans/deletions via `get_batch` / `bunch_delete`.
- [] Build replication on txnid markers, the replica — on a read-only snapshot filtered by `last_seen`.

Examples for this chapter: [examples/c++/39-pattern-sequence-id.c++](#); [examples/c++/40-pattern-secondary-index.c++](#); [examples/c++/41-pattern-composite-key.c++](#); [examples/c++/42-pattern-queue.c++](#); [examples/c++/43-pattern-ring-buffer.c++](#); [examples/c++/44-pattern-full-scan.c++](#); [examples/c++/45-pattern-replication.c++](#); [examples/c++/46-pattern-read-your-writes.c++](#).

7.7 Chapter 33. Roadmap and the future: MithrilDB

7.7.1 33.1. Current state

libmdbx as of 2026 — a mature engine (0.15.x devel-canon), the DB format frozen since 2018. Fundamental GC/freelist improvements are planned only in the next generation.

7.7.2 33.2. MithrilDB

A common API for several storage formats; amalgamation simplifies distribution.

7.7.3 33.3. Replication

Prerequisites: change subscription, early GC cleanup (2025), non-linear GC processing (end-to-end tracking of page usage by readable snapshots).

7.7.4 33.4. Other roadmap items

- Change subscription (mailboxes, ring buffers).
- Encryption and compression at the engine level.
- A streaming read/write mode for long values ("Streaming BLOBs") — without manual chunking, see [Restrictions & gotchas](#).
- SWIG and cross-language interop.

7.7.5 33.5. Summary of chapter 33

- libmdbx as of 2026 — a mature engine (0.15.x devel-canon); the DB format frozen since 2018.
- Fundamental GC/freelist improvements are planned only in the next generation — MithrilDB: a common API for several storage formats, amalgamation simplifies distribution.
- Replication prerequisites: change subscription, early GC cleanup (2025), non-linear GC processing (end-to-end tracking of page usage by readable snapshots).
- Other roadmap items: engine-level encryption and compression, a streaming read/write mode for long values ("Streaming BLOBs"), SWIG and cross-language interop.

7.7.6 33.6. Chapter 33 checklist

- [] In long-term planning account for the DB format being frozen since 2018 (0.11.x ↔ 0.12.x compatibility).
 - [] Do not expect fundamental GC/freelist improvements in the current generation — watch MithrilDB.
 - [] For replication rely on the available prerequisites (change subscription, early GC cleanup).
 - [] Treat roadmap features (encryption, compression, the streaming mode for long values / Streaming BLOBs, SWIG) as future, not current, capabilities.
-

7.8 Volume summary

You know the rules and tips, platform-specific notes, HSR, diagnostics via proven algorithms, migration from LMDB and design patterns.

What's next: Volume VI — bindings: how to use libmdbx from Rust, Go, Python, Node.js, .NET, C++, Dart, Nim, Java, Haskell, Ruby, Scala.

8. Volume VI. Bindings

Level: for developers working in languages other than C/C++. **Volume goal:** you know how to use libmdbx from your language and understand the specifics and limitations of each binding. **Honesty:** facts about specific wrappers (repository, version, bugs) require analysis of their repositories and issues. Architectural consequences (NoStickyThreads, Send/Sync, GC/pinning, GIL — the Global Interpreter Lock of Python) are given here; positions that need to be checked against the repository are marked explicitly.

8.1 Chapter 34. Overview of the bindings ecosystem

Officially tracked bindings: **Rust, Go, Node.js, Zig, Python, .NET (C#), C++, Dart, Nim, Java, Haskell, Ruby, Scala**. Plus unofficial ones (openresty/lua and others).

Binding	Language	Status	Details
mdbx.h++	C++	Mature (official C++ API)	Ch. 40
mdbx-rs / mdbx-sys	Rust	Mature/active	Ch. 35
mdbx-go	Go	Active	Ch. 36
python-lmdbx / mdbx-py	Python	Active	Ch. 37
node-mdbx	Node.js	Active	Ch. 38
mdbx-zig (lmdbx-zig)	Zig	Officially tracked	\$34.2
libmdbx-dotnet	.NET / C#	Active	Ch. 39
mdbx-dart / Isar	Dart (Flutter)	Active (Isar uses libmdbx)	Ch. 41
Nim	Nim	—	Ch. 42
Java	Java	—	Ch. 43
Haskell	Haskell	—	Ch. 44
Ruby	Ruby	—	Ch. 45
Scala	Scala	—	Ch. 46

Note: the cards for Ch. 42–46 (Nim, Java, Haskell, Ruby, Scala) are brief; repository details require analysis (see the notes in each card).

8.1.1 34.1. What each binding covers

A typical picture: a binding wraps the **C API** (`mdbx_env_*`, `mdbx_txn_*`, `mdbx_dbi_*`, `mdbx_cursor_*`, `MDBX_val`) into the idiom of the language. Coverage completeness differs: the core (`env/txn/get/put/del/cursor`) is covered everywhere; advanced APIs (`cache_get`, `defrag`, `clone`, `embark_read`, `estimate_*`) — not always. Check the documentation of the specific binding.

8.1.2 34.2. Zig (brief reference)

`mdbx-zig` (repository `lmdbx-zig`) is an officially tracked binding for Zig. Known specifics: Zig is actively used for cross-compiling C/C++; `MDBX_HAVE_BUILTIN_CPU_SUPPORTS=0` may be needed to work around toolchain bugs. No separate card is dedicated to it — repository and version details require analysis.

8.1.3 34.3. Summary of chapter 34

- Officially tracked bindings: Rust, Go, Node.js, Zig, Python, .NET (C#), C++, Dart, Nim, Java, Haskell, Ruby, Scala; plus unofficial ones (openresty/luas and others).
- Each binding wraps the C API (`mdbx_env_*`, `mdbx_txn_*`, `mdbx_dbi_*`, `mdbx_cursor_*`) into the idiom of the language.
- The core (`env/txn/get/put/del/cursor`) is covered everywhere; advanced APIs (`cache_get`, `defrag`, `clone`, `embark_read`, `estimate_*`) — not always.
- Mature/active: `mdbx.h++` (C++), `mdbx-rs/mdbx-sys` (Rust), `mdbx-go` (Go), `python-lmdbx/mdbx-py`, `node-mdbx`, `libmdbx-dotnet`, `mdbx-dart/Isar`.
- `mdbx-zig` (`lmdbx-zig`) — officially tracked; `MDBX_HAVE_BUILTIN_CPU_SUPPORTS=0` may be needed to work around toolchain bugs.
- The Nim/Java/Haskell/Ruby/Scala cards are brief; repository details require analysis.

8.1.4 34.4. Chapter 34 checklist

- [] Choose a binding per the chapter table and verify its status against the repository.
- [] Check which advanced APIs (`cache_get`, `defrag`, `clone`, `embark_read`, `estimate_*`) the binding supports.
- [] For Zig, account for `MDBX_HAVE_BUILTIN_CPU_SUPPORTS=0` in case of toolchain problems.
- [] For the brief cards (Nim/Java/Haskell/Ruby/Scala), analyze the binding's repository before production.

8.2 Chapter 35. Rust (`mdbx-sys` / `mdbx-rs`)

Overview. `mdbx-sys` — low-level FFI bindings to the C API; `mdbx-rs` — a safe wrapper (idiomatic Rust). Status — active/mature. (*Repository details require verification.*)

Installation. Via cargo: `mdbx-sys = "..."`, `mdbx-rs = "..."`.

Basic example (simplified, generic for the kind):

```
use mdbx::{Env, Environment};

fn main() -> mdbx::Result<()> {
    let env = unsafe { Env::builder().open("db.mdbx")? };
    let txn = env.begin_rw_txn()?;
    txn.put(b"greeting", b"hello, libmdbx");
    txn.commit()?;
    Ok(())
}
```

(The exact syntax depends on the binding version — check its README.)

Critical nuances (from the libmdbx architecture):

- **Send/Sync.** A transaction and a cursor are bound to a thread (sticky). A safe wrapper must reflect this in its types: a transaction is `!Sync` (use from two threads = `THREAD_MISMATCH`). For async runtimes (tokio and others), `MDBX_NOSTICKYTHREADS` is mandatory, since a future may continue on another OS thread.
- **Use-after-free after commit/abort.** `MDBX_val` points into mmap; after the transaction ends the pointers are invalid. The wrapper must restrict the lifetime of values to the transaction's lifetime rather than handing `&[u8]` out.
- **Blocking calls in an async context.** Commit and writes block the thread; in an async runtime — only via `spawn_blocking`, otherwise the whole worker is blocked.

Performance. The binding's overhead is minimal (FFI + ownership checks); reads remain close to C. (*Check the binding's benchmarks.*)

Gaps. Version-dependent: `cache_get`, `defrag`, `clone` and others may be missing.

Card summary. A two-layer pair: `mdbx-sys` — low-level FFI bindings to the C API, `mdbx-rs` — a safe idiomatic wrapper (installed via cargo). The main architectural consequences are sticky transactions (`!Sync`, `MDBX_NOSTICKYTHREADS` is mandatory for async runtimes) and invalid `MDBX_val` pointers after commit/abort; blocking calls in async — only via `spawn_blocking`. The overhead is minimal (FFI + ownership checks); advanced APIs (`cache_get`, `defrag`, `clone`) are version-dependent.

Card checklist.

- [] For an async runtime (tokio and others) set `MDBX_NOSTICKYTHREADS`.
- [] Ensure the wrapper restricts value lifetimes to the transaction (does not hand out `&[u8]` after commit/abort).
- [] Run blocking commits/writes via `spawn_blocking`.
- [] Check `cache_get/defrag/clone` availability against the binding version.

8.3 Chapter 36. Go (mdbx-go)

Overview. `mdbx-go` — Go binding via CGO. Status — active.

Installation. `go get github.com/.../mdbx-go`.

Basic example:

```
import "github.com/.../mdbx-go"

env, _ := mdbx.EnvCreate()
env.Open("db.mdbx", mdbx.CREATE|mdbx.NOSUBDIR, 0664)
txn, _ := env.BeginTxn(nil, 0)
dbi, _ := txn.DBIOpen(nil, 0)
txn.Put(dbi, []byte("k"), []byte("v"), 0)
txn.Commit()
```

(Verify the signatures against the repository.)

Critical nuances (from the architecture):

- **NoStickyThreads is always set.** A Go goroutine can migrate between OS threads, so the binding opens the environment with `MDBX_NOSTICKYTHREADS`.
- **One transaction object cannot be used from two goroutines simultaneously** — that is a race.
- **Deadlock risk:** functions requiring the environment write lock (`Env.SetOption`, `Env.Sync`, `Env.Stat`, `Env.Defrag`, `Env.Close`) can deadlock while an active write transaction runs in another goroutine. Pattern: one transaction — one goroutine, or explicit serialization via a channel/mutex.
- `runtime.LockOSThread` is needed if you want to pin a goroutine to an OS thread (reduces risks when working with the reader TLS slot).
- **CGO overhead** — CGO boundary on every operation; for hot paths group operations into batches.

Gaps. Completeness of the advanced API — check the repository.

Card summary. `mdbx-go` — an active CGO binding (installed via `go get`). Goroutines migrate between OS threads, so the binding always opens the environment with `MDBX_NOSTICKYTHREADS`; one transaction object cannot be used from two goroutines simultaneously. The main risks are deadlock on the environment's write functions and CGO overhead on every operation, mitigated by batching.

Card checklist.

- [] One transaction — one goroutine (or explicit serialization via a channel/mutex).
- [] Do not call `Env.SetOption` / `Env.Sync` / `Env.Stat` / `Env.Defrag` / `Env.Close` from another goroutine while a write transaction is active.
- [] Pin the goroutine via `runtime.LockOSThread` if needed.
- [] For hot paths group operations into batches (the CGO boundary).

8.4 Chapter 37. Python (python-lmdbx / mdbx-py)

Overview. Python binding over the C API. Status — active (*verification*).

Critical nuances (from the architecture):

- **GIL and blocking calls.** Long read transactions and bulk writes hold the GIL → other Python threads are blocked. Release the GIL in the binding during long operations.
- **Pointers into mmap are valid as long as the transaction is alive.** Python GC does not manage this — an explicit context manager (`with env.begin(...)`) is needed to guarantee commit/abort in time.
- **fork() after import.** With `multiprocessing`, call `resurrect_after_fork` in the child process.
- **ctypes vs CFFI vs pybind.** The choice affects overhead; for batches minimize the number of boundary calls.

Gaps. Verify against the repository (Python 3.x versions, wheels, advanced API).

Card summary. A Python binding over the C API, status active (verification). The key point is the GIL: long read transactions and bulk writes block other Python threads; the lifetime of mmap pointers must be guaranteed explicitly — via a context manager (`with env.begin(...)`). With `multiprocessing`, `resurrect_after_fork` is mandatory in the child process; the choice of ctypes/CFFI/pybind affects overhead.

Card checklist.

- [] Guarantee commit/abort via a context manager (`with env.begin(...)`).
- [] Release the GIL in the binding during long operations.
- [] Call `resurrect_after_fork` in the child process after `fork()`.
- [] For batches minimize the number of boundary calls (ctypes/CFFI/pybind).

8.5 Chapter 38. Node.js (node-mdbx)

Overview. Node.js binding (N-API). Status — active (verification).

Critical nuances (from the architecture):

- **Event loop and blocking calls.** The synchronous C API blocks the event loop. The wrapper should offer async variants (worker threads / libuv) for large operations.
- **Buffer ↔ MDBX_val.** A Node.js Buffer is an owning wrapper; when passing it to libmdbx watch the lifetime (do not pass a buffer that may be freed).
- **NoStickyThreads** — same as Go, if worker threads are used.

Gaps. Verify against the repository.

Card summary. `node-mdbx` — an active N-API binding (verification). The synchronous C API blocks the event loop — the wrapper should offer async variants (worker threads/libuv) for large operations; with worker threads `MDBX_NOSTICKYTHREADS` is needed (as in Go). A Node.js Buffer is an owning wrapper: watch its lifetime when passing it to libmdbx.

Card checklist.

- [] Run large operations via async variants (worker threads/libuv) without blocking the event loop.
- [] Do not pass buffers whose lifetime is not guaranteed.
- [] With worker threads set `MDBX_NOSTICKYTHREADS`.
- [] Verify the binding's capabilities against the repository.

8.6 Chapter 39. .NET / C# (libmdbx-dotnet)

Overview. .NET binding over the C API. An alternative in the ecosystem is LightningDB (not libmdbx! — check exactly what you are using). Status — active (verification).

Critical nuances (from the architecture):

- **Pinning.** `MDBX_val` points into mmap; if the value is a managed buffer, the .NET GC may move it. Solution: `fixed` / `pinned`, or a marshal copy, or `Span<T>`.
- **SafeHandle for env/txn/cursor.** Correct release via `SafeHandle`/finalizer.
- **GC finalizer and ordering.** Close in the correct order: `env` → `txn` → `cursor`; dangling handles give `BAD_DBI` / `BAD_TXN`.
- **Unpredictable lifetime** due to GC — do not hold long transactions between collections.

Gaps. Verify against the repository (versions, platforms, advanced API).

Card summary. `libmdbx-dotnet` — an active .NET binding over the C API (verification); do not confuse it with LightningDB (that is not `libmdbx`). The main point is pinning: `MDBX_val` points into mmap, while the .NET GC may move a managed buffer (`fixed` / `pinned`, a marshal copy or `Span<T>`). Release via `SafeHandle`/finalizer in the order `env` → `txn` → `cursor`; dangling handles give `BAD_DBI` / `BAD_TXN`.

Card checklist.

- [] Pin managed buffers (`fixed` / marshal copy / `Span<T>`), do not rely on the GC.
- [] Release `env/txn/cursor` via `SafeHandle`/finalizer in the correct order.
- [] Do not hold long transactions across GC collections.
- [] Verify that you use a `libmdbx` binding, not LightningDB.

8.7 Chapter 40. C++ (mdbx.h++)

Overview. The official C++ API (`mdbx.h++`), canon v0.15.0-263. Mature.

Class hierarchy:

- `env` / `env_managed` — environment; `txn` / `txn_managed` — transactions; `cursor` / `cursor_managed` — cursors; `map_handle` — table; `slice` / `buffer` — keys/values.
- The managed variants (`*_managed`) are RAII: automatic closing on scope exit.

Nuance: slice is non-owning, buffer is owning. `mdbx::slice` is merely a view over `MDBX_val` (a `std::string_view`-like reference): a link to bytes owned elsewhere, no copying. Therefore `mdbx::slice(std::to_string(k))` inline in a call argument is safe (the temporary string lives until the end of the full-expression, and `insert()` copies the bytes into the database synchronously), but keeping such a slice beyond the statement is a dangling reference. `mdbx::buffer` stores its own copy by default; the "reference" mode is enabled explicitly (`make_reference=true`) — then the `slice` lifetime rules apply.

Typed operations. Map-over-keys/values via type translation; parameter structs with fluent setters (geometry, mode, durability, reclaiming).

Exceptions. Hierarchy `mdbx::error` → `mdbx::exception` (from `std::runtime_error`) → `mdbx::fatal`; 35 typed exceptions (`bad_map_id`, `db_corrupted`, `db_full`, `key_exists`, `not_found`, `transaction_ousted` and others). Fatal errors lead to termination.

Fragment from `examples/c++/47-cpp-api.c++` — typed map operations (`insert` / `upsert` via `map_handle`) and typed exceptions (`mdbx::key_exists` and others):

```
{
    auto txn = env.start_write(); // RAII: will close/roll back itself on exception
    auto ordinal = txn.create_map("ordinal", mdbx::key_mode::ordinal, mdbx::value_mode::single);
    auto multi = txn.create_map("multi", mdbx::key_mode::usual, mdbx::value_mode::multi);

    txn.insert(ordinal, buffer::key_from_u64(42), "answer");
    txn.upsert(multi, mdbx::slice("tag"), mdbx::slice("x"));
    txn.upsert(multi, mdbx::slice("tag"), mdbx::slice("y"));

    // Typed exceptions by error code.
    try {
        txn.insert(ordinal, buffer::key_from_u64(42), "again");
        std::cerr << "FAIL: duplicate insert did not throw\n";
        return EXIT_FAILURE;
    } catch (const mdbx::key_exists &) {
        std::cout << "duplicate -> mdbx::key_exists as expected\n";
    }
}
```

```
txn.commit();
}
```

Full code: [47-cpp-api.c++](#).

C++20. Concepts, allocators (`buffer<>` with ownership policies), `inplace_storage_size_rounding`.

Example:

Fragment (C++, illustration): the full compilable version is in [examples/c++/47-cpp-api.c++](#). The map-style operations (`insert`, `upsert`, `get`, `erase`) are methods of the transaction, not of the table.

```
#include <mdbx.h++>

int main() {
    mdbx::env_managed env("db.mdbx",
                          mdbx::env::operate_parameters{});
    auto txn = env.start_write();
    auto db = txn.create_map("kv", mdbx::key_mode::usual,
                             mdbx::value_mode::single);
    txn.upsert(db, "greeting", "hello, libmdbx");
    txn.commit();
}
```

Check the exact syntax against your version of `mdbx.h++`.

Critical nuances:

- The `upper` value for a new database is chosen by the engine itself (~the golden section of RAM, bounded by the `mmap` limit ≈ 140 TB on 64-bit); `TOO_LARGE` / `ENOMEM` are possible with an explicitly excessive upper (e.g., under ASAN/Valgrind). Always specify the geometry explicitly.
- Exceptions vs return codes: pick a single style; `mdbx::error` is caught by type.

Gaps. `key_mode::msgpack` is declared but not implemented.

Card summary. `mdbx.h++` — the official mature C++ API (canon v0.15.0-263): RAII classes `env_managed` / `txn_managed` / `cursor_managed`, typed operations and 35 typed exceptions via `mdbx::error`. Always specify the geometry explicitly — the engine chooses `upper` for a new database itself (~the golden section of RAM). A known gap: `key_mode::msgpack` is declared but not implemented.

Card checklist.

- [] Use the RAII variants (`*_managed`) for automatic closing.
- [] Always specify the geometry explicitly (the engine chooses the default `upper`).
- [] Pick a single error style: `mdbx::error` exceptions (caught by type) or return codes.
- [] Do not use `key_mode::msgpack` (declared but not implemented).

Examples for this chapter: [examples/c++/47-cpp-api.c++](#).

8.8 Chapter 41. Dart (mdbx-dart / Isar)

Overview. `mdbx-dart` — a binding for Dart/Flutter; **Isar** — a popular local Flutter database using `libmdbx`. Status — active.

Critical nuances (from the architecture):

- **Isolate specifics.** Dart isolates are separate threads; transactions do not cross isolates (sticky thread). Each isolate has its own transactions.
- **Mobile platforms.** An embedded database on Android/iOS: be careful with parking long reads and with low WAF (extending flash lifetime).

Gaps. Verify against the repository.

Card summary. `mdbx-dart` — an active binding for Dart/Flutter; the popular local database Isar is built on libmdbx. Transactions do not cross isolates (sticky thread) — each isolate works with its own transactions. On mobile platforms (Android/iOS) parking long reads and low WAF (extending flash lifetime) matter.

Card checklist.

- [] Do not pass transactions between isolates (sticky thread).
- [] On Android/iOS carefully park long reads.
- [] Watch for low WAF to extend flash lifetime.
- [] Verify the details against the repository.

8.9 Chapter 42. Nim

Overview. Nim binding. *(Repository and version data require analysis of the binding's repository.)*

Expected nuances (from the architecture): memory management (Nim GC does not own mmap pointers — manual lifetime restrictions are needed); thread affinity (sticky).

Card summary. A Nim binding; repository and version data require analysis of the binding's repository. Expected nuances: the Nim GC does not own mmap pointers — manual lifetime restrictions are needed; transactions are bound to threads (sticky).

Card checklist.

- [] Manually restrict the lifetime of mmap pointers (the Nim GC does not own them).
- [] Keep transactions in the thread that created them (sticky).
- [] Verify the binding's repository and version.

8.10 Chapter 43. Java

Overview. Java binding (JNI). *(Data requires repository analysis.)*

Expected nuances (from the architecture): pinning critical buffers (JNI `GetByteArrayElements` or direct `ByteBuffer`); releasing native handles (`env/txn/cursor`) in the correct order; multithreading (each thread has its own transactions).

Card summary. A Java binding via JNI; data requires repository analysis. Expected nuances: pinning critical buffers (JNI `GetByteArrayElements` or direct `ByteBuffer`), releasing native handles (`env/txn/cursor`) in the correct order, each thread having its own transactions.

Card checklist.

- [] Pin critical buffers (JNI `GetByteArrayElements` /direct `ByteBuffer`).
- [] Release native handles (`env/txn/cursor`) in the correct order.
- [] Do not use one transaction from several threads.
- [] Verify the data against the binding's repository.

8.11 Chapter 44. Haskell

Overview. Haskell binding (FFI). *(Data requires repository analysis.)*

Expected nuances (from the architecture): purity and effects (STM vs IO); binding transactions to OS threads; the lifetime of pointers into mmap.

Card summary. A Haskell binding via FFI; data requires repository analysis. Expected nuances: reconciling purity and effects (STM vs IO), binding transactions to OS threads, the lifetime of pointers into mmap.

Card checklist.

- [] Reconcile transaction purity/effects (STM vs IO).
 - [] Account for transactions being bound to OS threads.
 - [] Restrict the lifetime of pointers into mmap.
 - [] Verify the data against the binding's repository.
-

8.12 Chapter 45. Ruby

Overview. Ruby binding. (*Data requires repository analysis.*)

Expected nuances (from the architecture): the GVL (Ruby VM's global lock — analogous to the Python GIL) and blocking calls; releasing native handles via finalizer/ensure.

Card summary. A Ruby binding; data requires repository analysis. Expected nuances: the GVL (Ruby VM's global lock — analogous to the Python GIL) and blocking calls; releasing native handles via finalizer/ensure.

Card checklist.

- [] Account for the GVL during long/blocking operations.
 - [] Release native handles via finalizer/ensure.
 - [] Verify the data against the binding's repository.
-

8.13 Chapter 46. Scala

Overview. Scala binding (via JVM). (*Data requires repository analysis.*)

Expected nuances (from the architecture): the same as Java (JNI, pinning, handles), plus Scala idiom specifics (futures — be careful with threads).

Card summary. A Scala binding via the JVM; data requires repository analysis. Expected nuances — the same as Java (JNI, pinning, handle release), plus Scala idiom specifics: futures require care with threads.

Card checklist.

- [] Account for the Java JNI nuances (pinning, handle release order).
 - [] In futures do not rely on a fixed thread.
 - [] Verify the data against the binding's repository.
-

8.14 Volume summary

- The binding core (env/txn/CRUD/cursor) is covered everywhere; the advanced API — not always.
 - Critical architectural consequences repeat themselves: NoStickyThreads (Go/async), Send/Sync and use-after-free (Rust), pinning/SafeHandle (Java/.NET), GIL (Python/Ruby), the lifetime of pointers into mmap (all).
 - The Nim/Java/Haskell/Ruby/Scala cards require repository analysis — marked explicitly.
-

8.15 Textbook conclusion (volumes I–VI)

You have walked the path from "what a key-value database is" to "why the GC cycle degrades with long readers" and "how to use libmdbx from any language". Refresh the concepts as needed, check your configurations against the checklists of Volume V, chapter 30, and measure — numbers without context are meaningless.