

Учебник по libmdbx

От первых шагов до экспертных тем и биндингов

Проект libmdbx

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

Содержание

1. Учебник по libmdbx	4
1.1 Для кого этот учебник	4
1.2 Структура и как читать	4
1.3 Примеры кода	4
1.4 Версии и совместимость	5
1.5 Об учебнике	5
2. Учебник по libmdbx — Том 0. Каркас (оглавление)	6
2.1 Общая логика прогрессии	6
2.2 Том I. Основы (для новичков)	6
2.3 Том II. Практическое использование (для разработчиков)	7
2.4 Том III. Внутренние механизмы (для понимающих)	7
2.5 Том IV. Производительность и оптимизация (для опытных)	8
2.6 Том V. Экспертные темы и edge cases (для продвинутых)	8
2.7 Том VI. Привязки (bindings)	9
3. Том I. Основы	10
3.1 Глава 1. Что такое libmdbx	10
3.2 Глава 2. Установка и первый запуск	12
3.3 Глава 3. Базовая модель данных	15
3.4 Глава 4. Базовые операции CRUD	18
3.5 Глава 5. Транзакции — первые шаги	21
3.6 Итог тома	24
4. Том II. Практическое использование	25
4.1 Глава 6. Курсоры	25
4.2 Глава 7. Мультизначения и DUPSORT	28
4.3 Глава 8. Вторичные индексы	31
4.4 Глава 9. Конфигурация окружения	33
4.5 Глава 10. Режимы долговечности	37
4.6 Глава 11. Многопоточность	39
4.7 Глава 12. Обработка ошибок	41
4.8 Итог тома	43
5. Том III. Внутренние механизмы	44
5.1 Глава 13. Архитектура хранения: B+tree и mmap	44
5.2 Глава 14. MVCC и снапшоты	46
5.3 Глава 15. Copy-on-Write и конвейер коммита	48
5.4 Глава 16. GC — сборщик мусора внутри БД	50

5.5	Глава 17. Рост, сжатие и дефрагментация БД	52
5.6	Глава 18. Вложенные транзакции	54
5.7	Глава 19. Файл блокировок и межпроцессная синхронизация	55
5.8	Глава 20. Долговечность и восстановление	57
5.9	Итог тома	59
6.	Том IV. Производительность и оптимизация	60
6.1	Глава 21. WAF — write amplification factor	60
6.2	Глава 22. Выбор конфигурации под сценарий	62
6.3	Глава 23. Микрооптимизации	63
6.4	Глава 24. Кэш-поиск (get-cached)	65
6.5	Глава 25. Массовые операции	66
6.6	Глава 26. Бенчмарки и измерения	68
6.7	Итог тома	70
7.	Том V. Экспертные темы и edge cases	71
7.1	Глава 27. Правила и советы из базы знаний	71
7.2	Глава 28. Платформенные нюансы	73
7.3	Глава 29. Handle-Slow-Readers (HSR)	74
7.4	Глава 30. Диагностика и отладка	76
7.5	Глава 31. Миграция с LMDB	78
7.6	Глава 32. Паттерны проектирования на libmdbx	80
7.7	Глава 33. Roadmap и будущее: MithrilDB	83
7.8	Итог тома	84
8.	Том VI. Привязки (bindings)	85
8.1	Глава 34. Обзор экосистемы биндингов	85
8.2	Глава 35. Rust (mdbx-sys / mdbx-rs)	86
8.3	Глава 36. Go (mdbx-go)	87
8.4	Глава 37. Python (python-lmdbx / mdbx-py)	88
8.5	Глава 38. Node.js (node-mdbx)	88
8.6	Глава 39. .NET / C# (libmdbx-dotnet)	89
8.7	Глава 40. C++ (mdbx.h++)	89
8.8	Глава 41. Dart (mdbx-dart / Isar)	90
8.9	Глава 42. Nim	91
8.10	Глава 43. Java	91
8.11	Глава 44. Haskell	91
8.12	Глава 45. Ruby	92
8.13	Глава 46. Scala	92
8.14	Итог тома	92
8.15	Заключение учебника (тома I–VI)	93

1. Учебник по libmdbx

Полное практическое руководство по встраиваемой транзакционной базе данных **libmdbx** — от первых шагов до экспертных тем и биндингов. Все примеры кода — компилируемые C++17/C11 программы в каталоге [examples/](#); фрагменты в тексте помечены «фрагмент» и дополнены ссылками на полный код. Для офлайн-чтения доступен [PDF-вариант учебника](#).

1.1 Для кого этот учебник

Учебник предназначен для разработчиков, работающих с C/C++ (и, в Томе VI, с другими языками через биндинги). Минимальный уровень — уверенное владение C и базовое представление о том, что такое базы данных. Опытные инженеры могут начинать со второго-третьего тома: внутренние механизмы, производительность и экспертные темы самодостаточны при условии знакомства с API.

1.2 Структура и как читать

Учебник состоит из шести томов, связанных единой логикой прогрессии:

Том	Тема	Уровень
Том I. Основы	Что такое libmdbx, установка, модель данных, CRUD, транзакции	новичок
Том II. Практика	Курсоры, DUPSORT, индексы, конфигурация, долговечность, многопоточность, ошибки	разработчик
Том III. Механизмы	B+tree, MVCC, CoW, GC, геометрия, блокировки, восстановление	понимающий
Том IV. Производительность	WAF, конфигурации, микрооптимизации, get-cached, массовые операции, бенчмарки	опытный
Том V. Экспертные темы	Правила, платформы, HSR, диагностика, миграция, паттерны, roadmap	продвинутый
Том VI. Биндинги	Rust, Go, Python, Node.js, .NET, C++, и др.	интегратор

- **Начинающим** рекомендуем последовательное чтение Томов I–II: там же ведётся сквозной проект-конфигуратор, упражнения которого закрепляют материал.
- **Опытным** — Тома III–V по необходимости; каждый том самодостаточен.
- **Интеграторам** — Том VI: карточки биндингов со спецификой каждого языка.

Каждая глава построена по схеме «концепция → механизм → практика → нюанс», завершается резюме, чек-листом и (в Томе I–II) упражнениями.

1.3 Примеры кода

Все примеры живут в [examples/](#) и собираются через CMake проекта (или pkg-config):

```
cmake -S . -B build -DMDBX_ENABLE_TESTS=ON
cmake --build build
ctest --test-dir build      # прогоняет и примеры
```

Ссылки вида [examples/c++/01-hello.c++](#) в тексте указывают на полный компилируемый пример (каталог [examples/](#) проекта, доступен и из этого каталога через символическую ссылку); краткие иллюстрации помечены «фрагмент» и не обязаны компилироваться сами по себе.

1.4 Версии и совместимость

Материал соответствует актуальной ветке `master` (на момент публикации — серия v0.15.x). Приведённые API проверены по `mdbx.h` / `mdbx.h++`; некоторые тонкие параметры (дефолты опций, сигнатуры сервисных функций) уточняйте по официальной документации и заголовочным файлам вашей версии.

1.5 Об учебнике

Учебник создан ИИ-агентом **skynet** на основе исходного кода `libmdbx` (ветка `master`), официальных заголовочных файлов `mdbx.h` / `mdbx.h++` и проектной документации. Материал прошёл сверку с кодом и перекрёстную проверку фактов. Цель — дать структурированный, точный и практичный путеводитель по `libmdbx`.

Обратная связь, замечания и предложения — в [issues](#) проекта или в [Telegram-группе](#).

libmdbx v0.15.0-150-g6ca39a94 — собрано 2026-10-19 · [онлайн-версия](#) · [PDF-версия](#)

2. Учебник по libmdbx — Том 0. Каркас (оглавление)

Это оглавление и структура учебника из шести томов: от базовых концепций до биндингов. Каждый том — отдельный файл в этом каталоге; все тома связаны перекрёстными ссылками и единым сквозным проектом.

2.1 Общая логика прогрессии

Базовые концепции (Том I) → Практика использования (Том II) → Внутренние механизмы (Том III)
→ Производительность и оптимизация (Том IV) → Экспертные темы и edge cases (Том V) → Привязки (Том VI)

Каждая глава — по схеме «концепция → механизм → практика → нюанс». Термин определяется при первом употреблении. Сквозной проект (конфигуратор приложений) ведётся с Тома I по Том II; упражнения — шаги этого проекта.

2.2 Том I. Основы (для новичков)

Цель: читатель без опыта работы с embedded key-value БД понимает libmdbx, умеет собрать и выполнить первые операции. Объём ~50–100 страниц.

• Гл. 1. Что такое libmdbx

- Встраиваемые vs клиент-серверные БД; key-value модель; почему mmap; место в ландшафте (LMDB/BerkeleyDB/LevelDB/RocksDB/SQLite на уровне архитектурных свойств); лицензия, экосистема, известные пользователи (Erigon, Reth, Isar, Monica Pass).

• Гл. 2. Установка и первый запуск

- Сборка (амальгама vs полный исходник), подключение (CMake/pkg-config/статика/динамика), ключевые опции сборки; минимальный пример open→put→get; запуск тестов.

• Гл. 3. Базовая модель данных

- MDBX_val; главная таблица и dbi; флаги MDBX_CREATE; ограничения (ключ/значение/число таблиц); integer-ключи; нулевые ключи/значения (отличие от LMDB).

• Гл. 4. Базовые операции CRUD

- put/get/del/replace; флаги (NOOVERWRITE/NODUPDATA/CURRENT/ALLDUPS); чтение get vs курсор; полные C-примеры; обработка кодов возврата.

• Гл. 5. Транзакции — первые шаги

- Зачем транзакции; begin/commit/abort; read-only vs read-write; MVCC интуитивно; правило «одна транзакция — один поток»; примеры.

2.3 Том II. Практическое использование (для разработчиков)

Цель: читатель строит приложения: курсоры, мультизначения, индексы, конфигурация, долговечность, многопоточность, обработка ошибок. Объём ~100–200 страниц.

- **Гл. 6. Курсоры** — позиционирование (FIRST/LAST/NEXT/PREV/SET/SET_RANGE/LOWER/UPPER_BOUND), пакетные операции, UB после удаления, clone/bind, паттерн safe-delete в DUPSORT, примеры.
- **Гл. 7. Мультизначения и DUPSORT** — DUPFIXED/INTEGERDUP/REVERSEDUP, вложенное поддерево значений, инвертированные индексы «ключ→список ID», GET_BOTH/GET_BOTH_RANGE, вторичный индекс.
- **Гл. 8. Вторичные индексы** — основная таблица + индексные; DUPSORT-индекс vs отдельная таблица; поддержание консистентности; составные ключи; удаление по индексу; мини-ORM.
- **Гл. 9. Конфигурация окружения** — геометрия (lower/now/upper/growth_step/shrink_threshold), maxreaders/maxdbs/pagesize, flags, runtime options MDBX_opt_*, выбор страницы 4/8/64 КБ, env_info/env_stat.
- **Гл. 10. Режимы долговечности** — SYNC_DURABLE/NOMETASYNC/SAFE_NOSYNC/UTTERLY_NOSYNC/WRITEMAP; когда какой; synchbytes/syncperiod; mdbx_env_sync_ex; платформенные нюансы (F_FULLFSYNC, boot_id, LockFileEx).
- **Гл. 11. Многопоточность** — sticky threads; NOSTICKYTHREADS (пулы/корутины); регистрация потоков; TLS-деструкторы; fork+resurrect; txn_clone; парковка/вытеснение; несколько сред; MDBX_EXCLUSIVE.
- **Гл. 12. Обработка ошибок** — положительные/отрицательные коды; RESULT_TRUE/FALSE; strerror_r vs strerror; MAP_FULL/TXN_FULL; ожидаемые коды (KEYEXIST/NOTFOUND); BAD_TXN/BAD_DBI/EBADSIGN; WANNA_RECOVERY/MVCC_RETARDED; THREAD_MISMATCH/TXN_OVERLAPPING/BUSY; retry-стратегии.

2.4 Том III. Внутренние механизмы (для понимающих)

Цель: читатель объясняет поведение БД через архитектуру — мост между «как пользоваться» и «как настраивать/отлаживать». Объём ~100–200 страниц.

- **Гл. 13. Архитектура хранения: B+tree и mmap** — ветви/листья/overflow; почему B+tree; страница; mmap без буферного кэша; CoW страницы; состояния страницы; инвариант достижимости.
- **Гл. 14. MVCC и снапшоты** — версионность страниц; тройка мета-страниц (Troika), двухфазный коммит, конечный автомат 216 состояний; RLT; wait-free чтение; safe64; писатель (мьютекс, front txnid, DPL); сравнение с PostgreSQL/Oracle.
- **Гл. 15. Copy-on-Write и конвейер коммита** — touch→копия→DPL→спилл→коммит; дедупликация DPL; спилл (LRU, приоритет overflow, защита курсоров, знаменатели); loose; refund; стадии конвейера (GC→refund→спилл→аудит→мета→sync→разблокировка) и влияние на задержку.
- **Гл. 16. GC** — почему не free-list; формат записей (ключ=txnid, значение=PNL); детент и его вычисление/кэширование; FIFO vs LIFO; BigFoot; рекурсивность обновления; rp_augment_limit/ gc_time_limit; ранняя очистка GC 2025; не-отложенная очистка (devel 0.14.x).
- **Гл. 17. Рост, сжатие и дефрагментация** — почему растёт (читатели/SAFE_NOSYNC/фрагментация); геометрия; MAP_FULL и разрешающие меры (steady-point/HSR/выселение); авто-компактификация refund и её пределы; mdbx_env_defrag; гистерезис; оценка ёмкости.
- **Гл. 18. Вложенные транзакции** — назначение; CoW-тень родителя, спаренные курсоры; join vs undo; судьба таблиц; анти-паттерн «вложенная в каждой функции»; workaround нехватки места; совместимость с WRITEMAP.
- **Гл. 19. Файл блокировок и межпроцессная синхронизация** — зачем LCK; реализации (POSIX-2008/2001, SysV, WIN32FILES), почему LockFileEx, а не именованные мьютексы; версионирование LCK; восстановление после сбоя; reader_check; режим без LCK.
- **Гл. 20. Долговечность и восстановление** — two-phase мета (weak/steady); восстановление без WAL; open for recovery (монотонные номера, эксклюзив); boot_id и LXC; некогерентность page cache (#269, FORCE_CHECK_MMAP_COHERENCY); mdbx_chk; что переживёт сбой питания по режимам.

2.5 Том IV. Производительность и оптимизация (для опытных)

Цель: читатель настраивает libmdbx под сценарий, находит и устраняет узкие места. Объём ~80–150 страниц.

- **Гл. 21. WAF** — что и почему; источники амплификации; батчинг как главный рычаг; высота дерева; спилл-компромисс; `prefer_waf_insteadof_balance/merge_threshold`; расчёт WAF с числами.
- **Гл. 22. Выбор конфигурации под сценарий** — 6 сценариев (запись-heavy, чтение-heavy, мобильное, аналитика, многопроцессное чтение, низкая задержка): режимы/флаги/геометрия/страница/батчинг/числа.
- **Гл. 23. Микрооптимизации** — SIMD-поиск, branchless bsearch, radix-sort, атомики для слабых моделей, минимизация `syscall`, `prefault+mincore`, `auto-appending split`, `merge` с грязным соседом; когда важны.
- **Гл. 24. Кэш-поиск (get-cached)** — понятие, статусы HIT/CONFIRMED/REFRESHED/DIRTY/BEHIND/UNABLE/RACE; когда ускоряет; ABA; многопоточный вариант + `NOSTICKYTHREADS`; `SingleThreaded` vs `multithreaded`; история (Gabriel RABNI, Ghost Body Object).
- **Гл. 25. Массовые операции** — `bunch_delete`, `get_batch`, `GET/PUT_MULTIPLE`, `estimate_range/distance/move`, `txn_clone`, батчинг 100–10000 операций; когда «максимальная транзакция» — анти-паттерн.
- **Гл. 26. Бенчмарки и измерения** — `ioarena`; что измерять; `commit_latency` по стадиям; `PROFGC`; типичные ошибки бенчмаркинга; «6–7× медленнее LMDB» — что это значит; ориентиры (200 TPS, 1–3M `get/s`, 20K–10M вставок/с); масштабирование чтения.

2.6 Том V. Экспертные темы и edge cases (для продвинутых)

Цель: читатель знает тонкости и грабли, умеет отлаживать и принимать архитектурные решения. Объём ~100–200 страниц.

- **Гл. 27. Правила и советы** — проверенные практики и «грабли», контекст критичности, контрпримеры.
- **Гл. 28. Платформенные нюансы** — 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_PREALLOCATE`); Android/bionic; контейнеры (`LXC/Docker`); Wine; Linux <4.x; `tmpfs`.
- **Гл. 29. Handle-Slow-Readers (HSR)** — зачем; `mdbx_env_set_hsr`; параметры и возвраты (ждать/убить/рост/`MAP_FULL`); типовые реализации; `HSR + SAFE_NOSYNC`; пример на C.
- **Гл. 30. Диагностика и отладка** — инструменты (`mdbx_chk` опции и `exit`-коды, `PROFGC` и семантика `gc_prof`, `commit_latency` по стадиям, `txn_info`, `reader_check/list`, логирование, `ASAN/UBSAN/valgrind`); пошаговые сценарии (рост БД, тормоз коммита, `MAP_FULL`, `deadlock/ BUSY`, долгий читатель, повреждение, утечка слотов, производительность, фрагментация, `fork`, Windows-тормоза, контейнеры); чек-листы перед `production`.
- **Гл. 31. Миграция с LMDB** — преимущества; совместимость форматов (0.11.x ↔ 0.12.x); breaking changes (`NOLOCK` убран, `NOTLS` → `NOSTICKYTHREADS`); таблица соответствия `mdb_` → `mdbx_`; поведенческие отличия (тройка мет, двухфазный коммит, контрольные суммы); чек-лист миграции.
- **Гл. 32. Паттерны проектирования** — 8 паттернов (`auto-ID` через `sequence`, вторичный индекс `DUPSORT`, составной ключ+компаратор, очередь задач `table-as-queue`, кольцевой буфер, полносканирующий итератор, репликация через `per-table txnid`, `read-your-writes` через `clone/ embark_read`): задача → решение с кодом → компромиссы → альтернативы.
- **Гл. 33. Roadmap и будущее: MithrilDB** — текущее состояние; общий API нескольких форматов; амальгамация; репликация (нелинейная GC); подписка на изменения; шифрование/сжатие; потоковый режим длинных значений («Streaming BLOBs»); SWIG.

2.7 Том VI. Привязки (bindings)

Цель: читатель использует libmdbx из своего языка с пониманием специфики и ограничений биндинга. Объём ~50–150 страниц.

- **Гл. 34. Обзор экосистемы** — полный список официально отслеживаемых биндингов; статусы (зрелый/экспериментальный/устаревший); карта «язык → биндинг».
- **Гл. 35–46. Карточки биндингов** (по единому шаблону: обзор/установка/базовый пример/ threading/память/нюансы/типовые ошибки/производительность/пробелы/ссылки): 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 (официально отслеживаемый) — краткая справка в гл. 34 (§34.2), без отдельной карточки.
- **Обязательные нюансы:** Go+NOSTICKYTHREADS (deadlock-сценарии, CGO); Rust+async (Send/Sync, use-after-free после commit, блокирующие вызовы); .NET+GC (SafeHandle, pinning, время жизни); Python+GIL (долгие чтения, fork после import).
- **Честность:** при недостатке данных — «требуется анализа репозитория биндинга».

libmdbx v0.15.0-150-g6ca39a94 — собрано 2026-10-19 · [онлайн-версия](#) · [PDF-версия](#)

3. Том I. Основы

Уровень: для новичков — тех, кто никогда не работал с embedded key-value базами данных. **Цель тома:** вы понимаете, что такое libmdbx, умеете собрать её, открыть базу и выполнить первые операции чтения и записи в транзакциях. **Сквозной проект:** на протяжении томов I–II мы строим одно приложение — **конфигуратор приложений** (простое key-value хранилище параметров, которое постепенно обрастает индексами, многопоточностью и оптимизациями).

Прогрессия тома: концепция → практика → механизм → нюанс. Код примеров — полный и компилируемый; вспомогательные функции (`env_open` в C++ / `ex_env_open` в C, а также `check_rc`, `die`) едины для всего учебника (см. `examples/common/`).

3.1 Глава 1. Что такое libmdbx

3.1.1 1.1. Встраиваемая база данных

Базы данных бывают двух принципиально разных видов: **клиент-серверные** и **встраиваемые**.

Клиент-серверная база (PostgreSQL, MySQL) — это отдельный процесс-сервер. Ваше приложение общается с ним по сети или сокету: отправляет SQL, получает ответы. Сервер владеет файлами данных, управляет кэшем, конкурентным доступом и журналом. Вы платите за это сложностью развёртывания, отдельным процессом и сетевыми накладными.

Встраиваемая база (SQLite, libmdbx) — это **библиотека**, которую вы линкуете прямо в свой процесс. Нет сервера, нет сети: ваше приложение вызывает функции, а те напрямую читают и пишут файл. Преимущества — простота (одна библиотека), скорость (нет сетевого слоя) и лёгкость распространения (приложение само носит свою базу). Плата — вы сами отвечаете за конкурентный доступ и резервное копирование.

libmdbx — встраиваемая транзакционная **key-value** база данных, глубоко переработанный потомок известной библиотеки LMDB. «Транзакционная» означает, что группа операций выполняется атомарно: либо все изменения применяются, либо ни одно.

Аналогия. SQLite — это «встраиваемый PostgreSQL». libmdbx — это «встраиваемый высокопроизводительный кэш-сервер»: она не знает SQL и таблиц в реляционном смысле, но зато умеет отдавать данные со скоростью обращения к памяти.

3.1.2 1.2. Key-value модель данных

Key-value база хранит простые пары: **ключ** → **значение**. Обе части — просто последовательности байтов, о смысле которых знает только ваше приложение.

```
"user:1001" → {"name": "Анна", "role": "admin"}
"user:1002" → {"name": "Борис", "role": "user"}
"theme"    → "dark"
```

Отличие libmdbx от «обычного» словаря (`std::map`, `HashMap`):

- **Ключи всегда упорядочены.** Данные хранятся в сбалансированном дереве (B+tree), поэтому ключи можно обходить по порядку, искать диапазоны, находить «ближайший больший ключ». Это как массив с отсортированными ключами, а не как хэш-таблица.
- **Значения могут быть большими** — вплоть до ~2 ГБ (уходят на отдельные «overflow»-страницы).
- **Данные лежат в файле, отображённом в память.** Чтение ключа — это фактически разыменование указателя, без копирования и без сериализации.

3.1.3 1.3. Почему mmap: память как интерфейс к данным

Большинство баз данных держат собственный буферный кэш: при чтении — копируют страницу с диска в память библиотеки; при записи — накапливают изменения в этом кэше и периодически сбрасывают на диск.

libmdbx использует другой подход: **memory-mapped файл** (mmap). Операционная система отображает файл базы прямо в адресное пространство процесса. Данные выглядят как обычный массив байтов. Чтение — это обращение по адресу; страницу при необходимости подтянет ядро (page cache). Отдельного буферного кэша в библиотеке нет — его роль выполняет страничный кэш ОС.

Из этого следуют три важных свойства:

- 1. **Чтение не копирует данные.** Вы получаете указатель прямо в mmap-области — никакой десериализации. Это источник рекордной скорости чтения (обычно миллионы операций в секунду).
- 2. **Запись идёт через Copy-on-Write (CoW).** Меняя страницу, библиотека не трогает старую версию (её могут читать другие транзакции), а создаёт новую копию. Подробности — в Томе III.
- 3. **Операционная система сама решает, какие страницы держать в памяти.** Библиотека не выбирает стратегию вытеснения — это делает ядро.

3.1.4 1.4. Место libmdbx в ландшафте

База	Модель	Архитектурные свойства
LMDB	KV, mmap, MVCC	Родоначальник подхода; проще, меньше возможностей
libmdbx	KV, mmap, MVCC	Переработанный LMDB: больше режимов долговечности, GC вместо free-list, жёстче гарантии
BerkeleyDB	KV, классическая	Старшее поколение; блокировки уровня страниц, нет mmap
LevelDB/ RocksDB	KV, LSM	Оптимизированы на запись, фоновое сжатие, плохая предсказуемость чтения
SQLite	Реляционная, встраиваемая	SQL, транзакции; медленнее на KV-сценариях

Здесь **MVCC** (multi-version concurrency control, управление многоверсионностью) — способ одновременного чтения и записи без блокировок читателей: каждая транзакция видит неизменный снимок данных (подробно — глава 5 и том III). **LSM** (log-structured merge-tree) — альтернативная архитектура, оптимизированная под запись ценой предсказуемости чтения.

Сравнение на уровне свойств (не бенчмарков): если вам нужен упорядоченный key-value с транзакциями, молниеносным чтением и предсказуемой записью — выбирайте libmdbx. Если нужен SQL — SQLite. Если запись доминирует над чтением и можно терпеть фоновое сжатие — RocksDB.

3.1.5 1.5. Лицензия, экосистема, кто использует

libmdbx распространяется по разрешительной лицензии: исторически — OpenLDAP Public License (OLPL), на master — **Apache-2.0 целиком** (ре-лицензирование в 0.13.x). Позволяет коммерческое использование без открытия собственного кода.

Библиотеку используют в высоконагруженных проектах:

- **Ethereum-экосистема:** Erigon, Akula, Silkworm (все быстрые реализации Ethereum работают на libmdbx);
- **Isar** — локальная база Flutter-приложений;
- мессенджеры, аналитика, кэш-слои.

Экосистема биндингов покрывает Rust, Go, Python, Node.js, .NET, C++, Zig, Dart, Nim, Java, Haskell, Ruby, Scala (подробно — Том VI).

3.1.6 1.6. Резюме главы 1

- `libmdbx` — встраиваемая транзакционная key-value БД (библиотека, не сервер).
- Ключи упорядочены (B+tree); значения — произвольные байты до ~2 ГБ.
- Данные отображены в память (`mmap`): чтение без копирования, запись через `Copy-on-Write`.
- Свойства: `wait-free` читатели, один писатель, транзакции, отсутствие WAL (восстановление по мете).
- `Apache-2.0 (master)/OLPL` (исторически); используется в Ethereum-клиентах, Isar и других проектах.

3.1.7 1.7. Упражнения

1. Чем клиент-серверная БД отличается от встраиваемой? Приведите пример, когда встраиваемая предпочтительнее.
2. Что даёт упорядоченность ключей сверх обычного словаря?
3. Почему чтение через `mmap` может быть быстрее, чем через собственный кэш БД?

3.1.8 1.8. Чек-лист главы 1

- [] я могу объяснить разницу между клиент-серверной и встраиваемой БД и сказать, когда какая предпочтительнее;
- [] понимаю, что `libmdbx` — упорядоченный key-value движок (B+tree), а не хэш-таблица и не SQL;
- [] могу описать роль `mmap`: чтение без копирования, запись через `Copy-on-Write`, вытеснение страниц делает ядро;
- [] знаю ключевые свойства: MVCC, `wait-free` читатели, один писатель, отсутствие WAL;
- [] помню лицензию (`Apache-2.0` на `master`) и примеры проектов на `libmdbx` (Erigon, Isar).

3.1.9 1.9. Что дальше

Понимание «что такое `libmdbx`» пора превратить в работающий код. В главе 2 мы соберём библиотеку двумя способами (амальгама и `CMake`), подключим её к проекту и запустим первый полный пример «hello, `libmdbx`». Для сквозного проекта-конфигуратора это обязательный шаг: без собранной библиотеки и проверенного цикла `open` → `put` → `get` двигаться дальше нельзя.

3.2 Глава 2. Установка и первый запуск

3.2.1 2.1. Сборка из исходников

`libmdbx` можно собирать двумя способами:

- **Амальгамированный** (амальгама): один файл `mdbx.c` + заголовок `mdbx.h`, которые можно просто положить в ваш проект. Минимум зависимостей, идеально для встраивания.
- **Полный исходник**: каталог с `CMakeLists.txt`, тестами, примерами и утилитами (`mdbx_stat`, `mdbx_chk`, `mdbx_copy`, `mdbx_dump`, `mdbx_load`, `mdbx_defrag`).

Амальгама: скачайте `mdbx.h` и `mdbx.c` и добавьте `mdbx.c` в сборку вашего проекта. Больше ничего не требуется — это настоящий «single-file» движок.

3.2.2 2.2. Подключение к проекту

CMake (полный исходник):

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

или **pkg-config**:

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

Собираете сами:

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

Статика vs динамика. Статическая линковка проще распространять (нет проблем с версиями .so) и даёт больше шансов оптимизации. Динамическая позволяет обновлять движок без пересборки приложения.

3.2.3 2.3. Ключевые опции сборки

Опция	Смысл
<code>MDBX_LOCKING</code>	Реализация блокировок: <code>POSIX2008</code> (Linux, по умолчанию), <code>POSIX2001</code> , <code>SYSV</code> (macOS по умолчанию), <code>WIN32FILES</code> (Windows)
<code>MDBX_CHECKING</code>	Уровень внутренних проверок/ассертов (-1...3). 2 / 3 сильно замедляют — только для отладки
<code>MDBX_FORCE_ASSERTIONS</code>	Устаревшая опция (эквивалент <code>MDBX_CHECKING=2</code>); лучше использовать <code>MDBX_CHECKING</code>
<code>MDBX_ENABLE_BIGFOOT</code>	Цепочки GC-записей для очень больших освобождений; включена по умолчанию на 64-битных сборках
<code>MDBX_DEBUG</code>	Логирование/ассерты/аудит (0 по умолчанию)
<code>MDBX_VALIDATION</code>	Дополнительные структурные проверки при каждой операции
<code>MDBX_ENABLE_PGOP_STAT</code>	Счётчики операций со страницами (для диагностики)

Совет: для production собирайте с `MDBX_CHECKING=0` (или не указывайте вовсе) и `NDEBUG` . Отладочная сборка с ассертами может быть в разы медленнее и маскировать реальные тайминги.

3.2.4 2.4. Минимальный пример

Первый полный листинг учебника. Он открывает базу, записывает ключ и читает его.

```
#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. Создаём окружение */
    rc = mdbx_env_create(&env);
    if (rc != MDBX_SUCCESS) die("env_create", rc);

    /* 2. Открываем (или создаём) файл базы; геометрия по умолчанию */
    rc = mdbx_env_open(env, "./demo.mdbx",
                       MDBX_NOSUBDIR, 0664);
    if (rc != MDBX_SUCCESS) die("env_open", rc);

    /* 3. Транзакция чтения-записи */
    rc = mdbx_txn_begin(env, NULL, MDBX_TXN_READWRITE, &txn);
    if (rc != MDBX_SUCCESS) die("txn_begin", rc);

    /* 4. Открываем главную таблицу (имя NULL = главная) */
    rc = mdbx_dbi_open(txn, NULL, 0, &dbi);
    if (rc != MDBX_SUCCESS) die("dbi_open", rc);

    /* 5. Записываем "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. Коммитим */
rc = mdbx_txn_commit(txn);
if (rc != MDBX_SUCCESS) die("txn_commit", rc);

/* 7. Читаем в новой read-only транзакции */
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("Значение: %s\n", (int)data.iov_len, (char *)data.iov_base);
mdbx_txn_abort(txn);

mdbx_env_close(env);
return 0;
}

```

Разберём по шагам.

- `mdbx_env_create` создаёт пустой объект окружения — «соединение» с будущей базой.
- `mdbx_env_open` открывает файл базы; создание управляется параметром `mode`: ненулевой `0664` означает «создать, если базы ещё нет». Флаг `MDBX_CREATE` сюда не передавайте — он для `mdbx_dbi_open()`, а его бит совпадает с `MDBX_NOMETASYNC`. `MDBX_NOSUBDIR` означает, что имя файла дано напрямую, а не как префикс для пары `данные+блокировки`.

Важно: бит `MDBX_CREATE = 0x40000` — это тот же бит, что и `MDBX_NOMETASYNC`. Если передать `MDBX_CREATE` в `flags` у `mdbx_env_open()`, БД создастся (из-за ненулевого `mode`), но приложение **молча** получит ослабленную долговечность (`NOMETASYNC`). Создание включается только через `mode != 0`.

- Все изменения происходят **внутри транзакции** — до коммита они не видны другим читателям.
- `mdbx_dbi_open(txn, NULL, 0, &dbi)` открывает **главную таблицу** (подробно — глава 3).
- `mdbx_put` кладёт пару; `mdbx_get` читает её.
- `MDBX_val` — структура `{ iov_base, iov_len }`: указатель и длина байтового окна.

Фрагмент из `examples/c++/01-hello.c++` — тот же цикл `open` → `put` → `get` → `close` на C++ API: RAII-объекты (`env_managed`, транзакции) сами закрывают окружение, а ключи и значения передаются как `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); // стереть остатки предыдущего запуска

        // Создание: конструктор с create_parameters создаёт БД, если её нет.
        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); // главная таблица
            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";
        }
    }
}

```

Полный код: [01-hello.c++](#) · [C-версия](#)

3.2.5 2.5. Сборка и запуск тестов

В полном исходнике:

```

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

```

Полезен стохастический стресс-тест `mdbx_test` с фиксированным `--prng-seed` — он воспроизводим и помогает отлаживать редкие ошибки (подробно в Томе V, глава 30).

3.2.6 2.6. Нюанс: файлы данных и блокировок

Одна «база» физически состоит из **двух файлов**: данных (`demo.mdbx`) и блокировок (`demo.mdbx-lck`). Файл блокировок содержит мьютекс писателя и таблицу читателей — это межпроцессная инфраструктура. Не удаляйте `.lck` при открытой базе; при копировании базы помните про пару файлов (подробно — Том III, глава 19).

Примеры к главе: [examples/c++/01-hello.c++](#) · C-версия; сквозной проект: [config-store-02.c++](#).

3.2.7 2.7. Резюме главы 2

- Амальгама (`mdbx.c` + `mdbx.h`) или полный исходник с CMake.
- Ключевые опции сборки: `MDBX_LOCKING`, `MDBX_CHECKING`, `MDBX_ENABLE_BIGFOOT`.
- Минимальный цикл: `create` → `open` → `txn_begin` → `dbi_open` → `put/get` → `commit` → `close`.
- База — пара файлов: данные + блокировки.
- Отладочные проверки включайте осознанно — они замедляют.

3.2.8 2.8. Упражнения

1. Соберите минимальный пример и запустите дважды — убедитесь, что значение переживает перезапуск.
2. Добавьте второй ключ и прочитайте оба в одной транзакции.
3. Что произойдёт, если при первом запуске (базы ещё нет) передать `mode = 0` ?

3.2.9 2.9. Чек-лист главы 2

- [] я умею собирать библиотеку двумя способами: амальгамой (`mdbx.c` + `mdbx.h`) и полным исходником с CMake;
- [] знаю ключевые опции сборки (`MDBX_LOCKING`, `MDBX_CHECKING`, `MDBX_ENABLE_BIGFOOT`) и почему для production нужны `MDBX_CHECKING=0` и `NDEBUG`;
- [] могу собрать и запустить минимальный пример, убедившись, что значение переживает перезапуск;
- [] знаю минимальный цикл работы: `create` → `open` → `txn_begin` → `dbi_open` → `put/get` → `commit` → `close`;
- [] помню, что база — это пара файлов (данные + `.lck`), и не удаляю файл блокировок при открытой базе.

3.2.10 2.10. Что дальше

Следующая глава — о том, что именно вы записываете и читаете: базовая модель данных libmdbx. Мы разберём `MDBX_val`, главную и именованные таблицы (`MDBX_dbi`), лимиты, integer-ключи и пустые ключи/значения. Эти понятия определяют схему сквозного проекта-конфигуратора: какие таблицы создавать и что в них хранить.

3.3 Глава 3. Базовая модель данных

3.3.1 3.1. MDBX_val: ключ и значение как байтовые строки

Все данные в libmdbx — последовательности байтов. Ни тип, ни кодировка, ни структура не интересуют движок. Окно на такие данные — `MDBX_val`:

```
typedef struct MDBX_val {
    size_t iov_len; /* длина в байтах */
    void *iovs_base; /* указатель на данные */
} MDBX_val;
```

Важно: `iov_base` обычно указывает **прямо в mmap-область** базы. Поэтому:

- читать значение можно без копирования;
- но после завершения транзакции указатель может стать недействительным — скопируйте данные, если собираетесь их хранить дольше транзакции.

Нюанс (C++): `mdbx::slice` и `mdbx::buffer`. В C++ API роль `MDBX_val` играют два типа. `mdbx::slice` — невладельческое представление (аналог `std::string_view` над `MDBX_val`): он лишь ссылается на чужие байты и ничего не копирует. Конструктор из `std::string` сделан `explicit` неслучайно: `mdbx::slice(std::to_string(k))` прямо в аргументе вызова — безопасно, потому что временная строка живёт до конца полного выражения (то есть пока не завершится весь вызов `insert()` / `get()`, который копирует байты внутрь базы), а вот сохранить такой срез в переменную и использовать позже — висячая ссылка. `mdbx::buffer` — наоборот, владеющий контейнер: по умолчанию он **копирует** содержимое (перегрузки с `make_reference=true` возвращаются к поведению `slice`), поэтому `mdbx::buffer(std::to_string(k))` не накладывает требований на время жизни источника.

3.3.2 3.2. Главная таблица и именованные таблицы (dbi)

Внутри одного файла базы живут **несколько деревьев**. Каждое дерево — «таблица» (в терминах API — `MDBX_dbi`, `map` / `sub-DB`). Есть:

- **главная таблица** — дескрипторы всех остальных таблиц (имя → корень дерева, флаги, счётчики);
- **именованные таблицы** — ваши данные;
- служебные (например, GC-дерево для свободных страниц — подробно в Томе III).

Открытие таблицы в транзакции:

```
/* главная таблица */
mdbx_dbi_open(txn, NULL, 0, &dbi_main);
/* именованная таблица (создаётся при первом открытии с MDBX_CREATE) */
mdbx_dbi_open(txn, "users", MDBX_CREATE, &dbi_users);
```

`MDBX_dbi` — это не глобальный объект, а **привязка к таблице внутри конкретной транзакции**: хендл валиден там, где его открыли, и наследуется вложенными транзакциями. Использование хендла из другой транзакции/потока — ошибка (`MDBX_BAD_DBI`).

Нюанс: повторное открытие таблицы. Для уже существующей таблицы `mdbx_dbi_open()` требует передачи её постоянных флагов (`MDBX_INTEGERKEY`, `MDBX_DUPSORT`, `MDBX_DUPFIXED`, ...); расхождение даёт `MDBX_INCOMPATIBLE`. Если флаги заранее неизвестны (таблица могла быть создана другим кодом), откройте её с флагом `MDBX_DB_ACCEDE`: таблица откроется с фактическими флагами, а узнать их можно через `mdbx_dbi_flags()`. Аналогичный флаг `MDBX_ACCEDE` есть у `mdbx_env_open()` — он открывает базу, уже используемую другим процессом в неизвестном режиме, без ошибки `MDBX_INCOMPATIBLE` (Том II, глава 9).

3.3.3 3.3. Ограничения

Параметр	Значение
Ключ	до ~половины страницы (зависит от размера страницы и режима таблицы); integer-ключи — 8 байт
Значение	до <code>MDBX_MAXDATASIZE</code> ≈ 2 ГБ
Таблиц	<code>MDBX_MAX_DBI</code> = 32765
Читателей	настраивается (<code>mdbx_env_set_maxreaders</code>), до <code>MDBX_READERS_LIMIT</code> = 32767

Размер страницы выбирается при создании базы: 256...65536 байт, по умолчанию 4096, и далее **не изменяется**. Выбор влияет на производительность (Том IV).

Актуальные пределы движок отдаёт через семейство `mdbx_limits_*` — они зависят от размера страницы и сборки, поэтому вместо «защитых» чисел используйте функции: `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()`. Удобные алиасы для отдельных значений — `mdbx_env_get_maxkeysize_ex` и `mdbx_env_get_maxvalsize_ex`.

3.3.4 3.4. Integer-ключи и нативный порядок байт

Для числовых ключей используйте специальные типы: `MDBX_INTEGERKEY` (таблица с целочисленными ключами `uint32_t` / `uint64_t` **нативного порядка**) и `MDBX_INTEGERDUP` (целочисленные мультизначения). Нативный порядок байт означает: ключ `42` в памяти как `uint64_t` — без перестановок байт. Это критично для сравнений и курсоров.

Нестандартный порядок сортировки задаётся **компаратором** — колбэком типа `MDBX_cmp_func`. Кастомные компараторы ключей/значений передаются при открытии таблицы в расширенных вариантах `mdbx_dbi_open_ex()` / `mdbx_dbi_open_ex2()` (аргументы `keycmp` / `datacmp`). Стандартные компараторы бинарных ключей/значений — `mdbx_cmp()` / `mdbx_dcmp()`; выбрать нужный под конкретный набор флагов таблицы можно через `mdbx_get_keycmp(flags)` / `mdbx_get_datacmp(flags)`.

3.3.5 3.5. Нулевые ключи и значения (отличие от LMDB)

В отличие от LMDB, `libmdbx` позволяет **пустые ключи** (длина 0) и **пустые значения** (длина 0). Это удобно для маркеров и флагов-наличия. Пустое значение — это `iov_len == 0` (указатель может быть NULL). Различайте «ключа нет» (`MDBX_NOTFOUND`) и «ключ есть с пустым значением» (`MDBX_SUCCESS` при `iov_len == 0`).

Фрагмент из `examples/c++/02-data-model.c++` — integer-ключи (`key_mode::ordinal`) и нулевое значение (`mdbx::slice()`):

```
// Главная таблица (name == nullptr): обычные ключи и значения.
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";

// Именованная таблица: отдельное пространство ключей внутри той же БД.
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-ключи: таблица с key_mode::ordinal (MDBX_INTEGERKEY),
// ключи — uint64_t в нативном порядке байт.
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";

// Пустой ключ (нулевой длины) и пустое значение.
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";
```

Полный код: [02-data-model.c++](#) · C-версия

Примеры к главе: [examples/c++/02-data-model.c++](#) · C-версия.

3.3.6 3.6. Резюме главы 3

- `MDBX_val` — байтовое окно `{len, ptr}`; указатель может жить в `mmap`.
- Один файл — много таблиц (деревьев); главная таблица хранит дескрипторы.
- `MDBX_dbi` — привязка к таблице в конкретной транзакции.
- Лимиты: ключ $\sim \frac{1}{2}$ страницы, значение ~ 2 ГБ, таблиц 32765.
- Integer-ключи — нативный порядок; пустые ключи и значения разрешены.

3.3.7 3.7. Упражнения

1. Создайте две именованные таблицы («settings», «counters») и положите по ключу в каждую.
2. Попробуйте прочитать ключ из хендла, открытого в другой транзакции — что вернёт API?
3. Проверьте поведение с пустым значением: положите `iov_len=0`, прочитайте — как отличить «пустое значение» от «ключа нет»?

3.3.8 3.8. Чек-лист главы 3

- [] я понимаю, что `MDBX_val` — это байтовое окно `{iov_len, iov_base}` без типа и кодировки;

- [] умею открывать главную и именованные таблицы через `mdbx_dbi_open()` и знаю роль флага `MDBX_CREATE`;
- [] понимаю, что `MDBX_dbi` валиден только в своей транзакции, и помню про `MDBX_BAD_DBI` и `MDBX_DB_ACCEDE`;
- [] проверяю лимиты через функции `mdbx_limits_*`, а не через зашитые числа;
- [] знаю, зачем `MDBX_INTEGERKEY` (нативный порядок байт) и как отличить пустое значение (`iov_len == 0`) от отсутствия ключа (`MDBX_NOTFOUND`).

3.3.9 3.9. Что дальше

Теперь, когда модель данных ясна, можно перейти к операциям над ней. В главе 4 мы освоим базовый набор CRUD — `mdbx_put`, `mdbx_get`, `mdbx_del`, `mdbx_replace` — с флагами записи, обработкой ожидаемых кодов и полным примером. Именно эти операции лягут в основу `cfg_set/cfg_get` сквозного конфигулятора.

3.4 Глава 4. Базовые операции CRUD

3.4.1 4.1. mdbx_put — вставка и обновление

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

Флаги:

Флаг	Смысл
0 / <code>MDBX_UPSERT</code>	Вставить или заменить значение
<code>MDBX_NOOVERWRITE</code>	Вставить только если ключа нет; иначе <code>MDBX_KEYEXIST</code>
<code>MDBX_NODUPDATA</code>	(DUPSORT) не добавлять дубликат значения
<code>MDBX_CURRENT</code>	Обновить текущее значение курсора

3.4.2 4.2. mdbx_get — чтение по ключу

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

Возвращает `MDBX_SUCCESS` (нашли) или `MDBX_NOTFOUND` (ключа нет). Для DUPSORT-таблиц нужен `MDBX_GET_BOTH` (искать конкретное значение) — подробно в Томе II, глава 7.

Расширенный вариант `mdbx_get_ex(txn, dbi, &key, &data, &values_count)` дополнительно возвращает **число значений ключа** (для обычной таблицы — 1; для DUPSORT — сколько значений связано с ключом; удобно без отдельного курсора и `mdbx_cursor_count()`).

3.4.3 4.3. mdbx_del — удаление

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

Если `data == NULL` — удаляется весь ключ. Если задан и таблица DUPSORT — удаляется конкретное значение.

Фрагмент из `examples/c++/03-crud.c++` — флаги записи на C++ API: `insert` соответствует `MDBX_NOOVERWRITE` (при повторе — исключение `mdbx::key_exists`), `upsert` — `MDBX_UPSERT`, `update` — `MDBX_CURRENT`. Флаги `MDBX_NODUPDATA` / `MDBX_ALLDUPS` для мультизначений — в Томе II, главы 6–7:

```
// NOOVERWRITE: вставка уникального ключа.
txn.insert(table, mdbx::slice("key"), mdbx::slice("value1"));

// Повторная вставка того же ключа -> 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 не бросает исключение, а сообщает результат флагом done.
auto ins = txn.try_insert(table, mdbx::slice("key"), mdbx::slice("value2"));
std::cout << "try_insert duplicate -> done=" << (ins.done ? 1 : 0) << "\n";

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

// CURRENT (update): обновить только существующий ключ.
txn.update(table, mdbx::slice("key"), mdbx::slice("value3"));
std::cout << "update existing: " << txn.get(table, mdbx::slice("key")).as_string() << "\n";

```

Полный код: [03-crud.c++ · C-версия](#)

3.4.4 4.4. mdbx_replace — условная замена

`mdbx_replace` объединяет `put+get` в одну операцию: заменить значение, узнать старое, вставить только если совпадает текущее. Используется для атомарных «сравни и замени» без лишних поисков.

```

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

```

3.4.5 4.5. Чтение: mdbx_get vs курсор

`mdbx_get` — точечный поиск по точному ключу. Если нужно обойти данные, найти диапазон, «ближайший больший ключ» — используйте курсор (Том II, глава 6). Для точного чтения `mdbx_get` достаточно.

3.4.6 4.6. Полный пример: 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: прочитайте старое значение и записать новое */
    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("было: %s\n", (int)old.iiov_len, (char *)old.iiov_base);

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

/* проверим удаление */
rc = mdbx_get(txn, dbi, &key, &out);
printf("get после удаления: %s (%d)\n",
      rc == MDBX_NOTFOUND ? "NOTFOUND (ожидаемо)" : "неожиданно", rc);

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

```

Внимание: `old_value` из `mdbx_replace` и `out` из `mdbx_get` указывают в `mmap`. Не храните эти указатели после `commit/abort` — данные могут быть переиспользованы.

3.4.7 4.7. Обработка ошибок

Коды возврата: `MDBX_SUCCESS` (0) — успех; `MDBX_RESULT_TRUE` / `MDBX_RESULT_FALSE` — специальные результаты некоторых операций; отрицательные `MDBX_*` — ошибки.

Частые «ожидаемые» коды (это не сбой, а штатные ответы):

- `MDBX_KEYEXIST` — ключ уже есть (при `MDBX_NOOVERWRITE`);
- `MDBX_NOTFOUND` — ключ (или значение) не найден;
- `MDBX_BAD_DBI` — невалидный/закрытый хендл таблицы.

Строковое описание: `mdbx_strerror(rc)`. Есть и потокобезопасный вариант `mdbx_strerror_r()` — используйте его в многопоточном коде (Том II, глава 12).

Примеры к главе: [examples/c++/03-crud.c++](#) · [C-версия](#); сквозной проект: [config-store-04.c++](#).

3.4.8 4.8. Резюме главы 4

- `put/get/del/replace` — базовые операции; флаги управляют поведением при конфликтах.
- `mdbx_get` — точечное чтение; для обхода/диапазонов — курсоры.
- `MDBX_KEYEXIST` / `MDBX_NOTFOUND` — ожидаемые состояния, а не ошибки.
- `mdbx_replace` — атомарная замена «чтение старого + запись нового» одним вызовом.
- Указатели из `mmap` валидны только внутри транзакции.

3.4.9 4.9. Упражнения

1. Добавьте обработку «expected error»: вставьте ключ повторно с `NOOVERWRITE` и обработайте `MDBX_KEYEXIST` без выхода из программы.
2. Напишите цикл, который вставляет 10 000 ключей в **одной** транзакции и измеряет время.
3. Что делает `mdbx_del` с `data`-аргументом в обычной (не DUPSORT) таблице?

3.4.10 4.10. Чек-лист главы 4

- [] я умею выполнять все базовые операции: `mdbx_put`, `mdbx_get`, `mdbx_del` и `mdbx_replace`;
- [] понимаю различие флагов записи `MDBX_NOOVERWRITE`, `MDBX_UPSERT`, `MDBX_CURRENT` и выбираю их осознанно;
- [] обрабатываю `MDBX_KEYEXIST` и `MDBX_NOTFOUND` как штатные состояния, а не сбой;
- [] понимаю, что `mdbx_replace` — атомарное «сравнить и заменить» одним вызовом;
- [] помню, что указатели из `mmap` (`out`, `old_value`) нельзя использовать после `commit/abort`.

3.4.11 4.11. Что дальше

Одиночные операции мы уже умеем, но настоящая сила libmdbx — в транзакциях. Глава 5 вводит begin/commit/abort, различие read-only и read-write транзакций и интуицию MVCC — почему читатели не блокируют писателя. Для конфигуратора это фундамент: атомарная запись группы настроек и безопасная параллельная работа с базой.

3.5 Глава 5. Транзакции — первые шаги

3.5.1 5.1. Что такое транзакция и зачем она нужна

Транзакция — это группа операций, которая выполняется **атомарно** и **изолированно**:

- либо все изменения применяются, либо ни одно (атомарность);
- пока транзакция не закоммичена, её изменения не видны другим (изоляция).

Пример: перевод денег — списать со счёта А и зачислить на счёт В. Если это сделать двумя отдельными операциями, между ними можно потерять данные при сбое. В транзакции — безопасно.

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)` — начать транзакцию. `flags == 0` (то же, что `MDBX_TXN_READWRITE`) — читающая-пишущая; для транзакции только на чтение передавайте `MDBX_TXN_RDONLY`.
- `mdbx_txn_commit(txn)` — применить изменения (для write) или просто закрыть (для read).
- `mdbx_txn_abort(txn)` — отбросить изменения и закрыть.

Совет: всегда закрывайте транзакцию — либо `commit`, либо `abort`. Незакрытая read-транзакция может «заморозить» переработку страниц (Том III, глава 16).

3.5.3 5.3. Read-only vs read-write

Read-only транзакции не блокируют друг друга и не требуют мьютекса писателя. Их можно иметь много одновременно, в любых потоках.

Read-write транзакция в любой момент времени в базе **одна** (глобальный мьютекс писателя). Если два потока попытаются начать пишущую транзакцию одновременно — один дожждётся другого.

3.5.4 5.4. Почему читатели не блокируют друг друга (MVCC интуитивно)

libmdbx использует **MVCC** (multiversion concurrency control). Упрощённо:

1. Каждая транзакция видит «снимок» базы — состояние на момент её начала.
2. Пишущая транзакция не изменяет существующие страницы, а создаёт **новые версии** (Copy-on-Write).
3. Читатель, начавшийся раньше, продолжает видеть старые версии, пока жив его снимок.

Поэтому читатели никогда не конфликтуют ни друг с другом, ни с писателем. Расплата — старые версии страниц занимают место, пока живы старые снимоты (подробно — Том III, главы 14 и 16).

Фрагмент из `examples/c++/04-transactions.c++` — read-only наблюдатель в отдельном потоке видит стабильный MVCC-снимок, пока пишущая транзакция перезаписывает ключ и затем делает `abort` (откат):

```
{
    auto txn = env.start_write();
    auto table = txn.open_map(nullptr);
```

```

    txn.insert(table, mdbx::slice("key"), mdbx::slice("v1"));
    txn.commit();
}

// Read-only наблюдатель в отдельном потоке: фиксированный снимок базы.
std::thread observer(observer_run, std::ref(env));
while (stage.load() != 1)
    std::this_thread::yield();

// Пишущая транзакция перезаписывает ключ, но ещё не коммитит.
{
    auto wtxn = env.start_write();
    auto table = wtxn.open_map(nullptr);
    wtxn.upsert(table, mdbx::slice("key"), mdbx::slice("v2"));
    stage = 2; // наблюдатель проверяет снимок при незакоммиченной записи
    while (stage.load() != 3)
        std::this_thread::yield();
    wtxn.abort(); // откат: v2 не сохраняется
}

```

Полный код: [04-transactions.c++](#) · C-версия

3.5.5 5.5. Базовое правило: одна транзакция — один поток

Транзакция привязана к потоку, который её создал («sticky thread»). Использовать объект транзакции из другого потока нельзя — это нарушение дисциплины (`MDBX_THREAD_MISMATCH`). Впрочем, есть режим `MDBX_NOSTICKYTHREADS` для пулов потоков и корутин (Том II, глава 11).

3.5.6 5.6. Полный пример с транзакциями

```

#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-транзакция: два ключа атомарно ---- */
    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 транзакция ---- */
    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: просто закрываем */

    /* ---- Abort: изменения отбрасываются ---- */
    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 не появится в базе */

    /* проверяем, что "c" нет */

```

```

rc = mdbx_txn_begin(env, NULL, MDBX_TXN_RDONLY, &txn);
if (rc) die("txn_r2", rc);
rc = mdbx_get(txn, dbi, &key, &out);
printf("после abort: %s\n", rc == MDBX_NOTFOUND ? "ключа нет (верно)" : "неожиданно");
mdbx_txn_abort(txn);

mdbx_env_close(env);
return 0;
}

```

Продолжить чтение после коммита — `mdbx_txn_commit_embark_read()`. Коммит в libmdbx «посаживает» транзакцию на свежий снимок (детент — удержание читателя на снимке состояния, механизм раскрыт в томе III); специальная функция `mdbx_txn_commit_embark_read()` делает то же и сразу возвращает **новую read-only транзакцию на уже закоммиченном состоянии** — без разрыва «коммит → новый begin». Полезно для паттернов read-your-writes и обработки очередей (Том V, глава 32). Идентификатор транзакции (порядковый номер в базе) можно узнать через `mdbx_txn_id()`.

3.5.7 5.7. Сквозной проект: начало

Наш конфигурактор приложений. Идея: хранить настройки в виде таблиц «секция → ключ → значение».

```

/* config_store.h — каркас сквозного проекта (главы 2-5) */
#ifndef CONFIG_STORE_H
#define CONFIG_STORE_H
#include <mdbx.h>

typedef struct {
    MDBX_env *env;
    MDBX_dbi dbi;
} 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

```

Реализация — следующими главами; к концу Тома II это будет полноценное приложение с индексами и многопоточностью.

Примеры к главе: [examples/c++/04-transactions.c++](#) · C-версия; сквозной проект: [config-store-05.c++](#).

3.5.8 5.8. Резюме главы 5

- Транзакция = атомарная, изолированная группа операций.
- begin (0/ MDBX_TXN_READWRITE — пишущая; MDBX_TXN_RDONLY — только чтение), commit, abort.
- MVCC: читатели видят снимок и не блокируют писателя.
- Одна транзакция — один поток (по умолчанию).
- Незакрытые read-транзакции «морозят» переработку страниц.

3.5.9 5.9. Упражнения

1. Напишите функцию `cfg_set`, которая использует `MDBX_NOOVERWRITE` для создания и `0` для обновления — как их объединить в одну транзакцию?
2. Измерьте: сколько write-транзакций в секунду выдерживает ваш диск (без батчинга)?
3. Почему начинать read-write транзакцию «на всякий случай» — плохая идея (подсказка: мьютекс писателя)?

3.5.10 5.10. Чек-лист главы 5

- [] я понимаю свойства транзакции: атомарность (всё или ничего) и изоляцию;
- [] умею начинать и корректно закрывать транзакции всех видов: `MDBX_TXN_READWRITE`, `MDBX_TXN_RDONLY`, `mdbx_txn_abort()`;
- [] понимаю, почему пишущая транзакция всегда одна, и как MVCC позволяет читателям не блокировать писателя;
- [] знаю правило «одна транзакция — один поток» и код `MDBX_THREAD_MISMATCH`;
- [] умею показать на примере, что abort отбрасывает изменения, а незакрытая read-транзакция мешает переработке страниц.

3.5.11 5.11. Что дальше

На этом Том I завершён: у вас есть работающий каркас configurator. Том II доводит его до полноценного приложения: курсоры для обхода и диапазонов (глава 6), DUPSORT и вторичные индексы (главы 7–8), конфигурация окружения (глава 9), режимы долговечности (глава 10), многопоточность (глава 11) и обработка ошибок (глава 12). Начинаем с курсоров — без них не сделать ни обхода таблицы, ни поиска по диапазону.

3.6 Итог тома

Вы познакомились с базовой моделью libmdbx: что это, как собрать, как открыть базу, выполнить CRUD в транзакциях. У вас работает сквозной проект-«configurator».

Что дальше: Том II — курсоры, мультимножества и DUPSORT, вторичные индексы, конфигурация окружения, режимы долговечности, многопоточность и обработка ошибок.

4. Том II. Практическое использование

Уровень: для разработчиков, уже освоивших Том I. **Цель тома:** вы умеете строить приложения на libmdbx — курсоры, мультизначения и DUPSORT, вторичные индексы, конфигурацию окружения, режимы долговечности, многопоточность и устойчивую обработку ошибок. **Сквозной проект:** конфигуратор приложений доводим до полноценного приложения с индексами и многопоточностью.

Схема глав: концепция → механизм → практика → нюанс.

4.1 Глава 6. Курсоры

4.1.1 6.1. Что такое курсор и зачем он нужен

`mdbx_get` умеет только точечный поиск по точному ключу. Для всего остального — обхода таблицы, поиска диапазона, «ближайшего большего ключа», итерации с удалением — нужен **курсor**.

Курсор — это «указатель» на позицию внутри дерева (точнее, стек позиций от корня к листу). Он позволяет двигаться по упорядоченным ключам в обе стороны и выполнять операции относительно текущей позиции.

4.1.2 6.2. Открытие и закрытие

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

Курсор принадлежит транзакции: создаётся внутри неё и живёт не дольше её. После `commit` / `abort` курсор закрывать нельзя — используйте его только до завершения транзакции.

4.1.3 6.3. Позиционирование

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

Ключевые операции:

Операция	Смысл
<code>MDBX_FIRST</code> / <code>MDBX_LAST</code>	Перейти к первому / последнему ключу
<code>MDBX_NEXT</code> / <code>MDBX_PREV</code>	Следующий / предыдущий ключ
<code>MDBX_SET</code>	Найти точный ключ (позиция на него)
<code>MDBX_SET_RANGE</code>	Найти первый ключ $\geq$ заданного («ближайший больший или равный»)
<code>MDBX_SET_LOWERBOUND</code>	Как <code>SET_RANGE</code> , но требует допустимый data-аргумент для DUPSORT
<code>MDBX_SET_UPPERBOUND</code>	Найти первый ключ $>$ заданного
<code>MDBX_GET_BOTH</code> / <code>GET_BOTH_RANGE</code>	(DUPSORT) найти конкретное значение / первое $\geq$ значения

После `MDBX_LAST`, если данные закончились, операция возвращает `MDBX_NOTFOUND`, а курсор оказывается «на конце» — состояние `eof`. Это штатная ситуация для завершения обхода.

Для частых сценариев есть отдельные удобные функции вместо пары «`cursor_get(op)` + проверка кода»: `mdbx_cursor_on_first()`, `mdbx_cursor_on_last()` и их dup-варианты `mdbx_cursor_on_first_dup()` / `mdbx_cursor_on_last_dup()`; проверить «конец обхода» — логический

`mdbx_cursor_eof()`. Дистанцию между двумя позициями курсора можно узнать через `mdbx_cursor_distance()` — полезно для оценки объёма диапазона перед обработкой.

Фрагмент из `examples/c++/05-cursors.c++` — прямая итерация (FIRST/NEXT) и `SET_RANGE` («ближайший больший или равный»):

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

// Прямая итерация: FIRST затем NEXT до конца.
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: первый ключ, не меньший заданного.
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";
```

Полный код: [05-cursors.c++](#) · [C-версия](#)

4.1.4 6.4. Пример: обход таблицы и поиск диапазона

```
#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);

    /* напомним таблицу */
    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);
    }

    /* полный обход */
    rc = mdbx_cursor_open(txn, dbi, &cur); if (rc) die("cursor", rc);
    printf("Все ключи:\n");
    while ((rc = mdbx_cursor_get(cur, &k, &d, MDBX_NEXT)) == MDBX_SUCCESS)
        printf(" %.*s\n", (int)k.iov_len, (char *)k.iov_base);
    /* на конце — MDBX_NOTFOUND, это нормально */
    if (rc != MDBX_NOTFOUND) die("next", rc);

    /* диапазон: первый ключ >= "c" и все следующие */
    printf("Диапазон >= 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. Курсор после удаления — UB (критическое предупреждение)

Внимание: после `mdbx_cursor_del()` позиция курсора не определена. Использование курсора без повторного позиционирования — неопределённое поведение. Всегда после удаления делайте `NEXT` (или `PREV`) и проверяйте `MDBX_NOTFOUND`.

4.1.6 6.6. Клонирование курсоров

`mdbx_cursor_clone()` размножает позицию курсора (для параллельной обработки на одном снапшоте), `mdbx_cursor_bind()` — привязывает существующий курсор к другой транзакции/таблице.

4.1.7 6.7. Паттерн safe-delete в DUPSORT

Удаление значений в DUPSORT-таблице одним курсором во время итерации — опасно (известны баги в 0.12.x, подробно в Томе V). Безопасный паттерн — **два курсора**: один позиционируется, второй удаляет.

Фрагмент (C, иллюстрация): полная компилируемая версия — ниже и в [examples/c++/06-dupsort-delete.c++](#).

```
/* удалить все значения ключа "target" в DUPSORT-таблице */
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); /* первый dup */
while (rc == MDBX_SUCCESS) {
    MDBX_val dk = k, dv = d;
    /* второй курсор на ту же позицию */
    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);
```

Для массового удаления целых диапазонов лучше использовать `mdbx_cursor_bunch_delete()` («удаление гроздьями») — оно вырезает целые страницы и ветви, а не перебирает элементы.

Фрагмент из [examples/c++/06-dupsort-delete.c++](#) — паттерн safe-delete двумя курсорами: `it` итерирует по дубликатам (`NEXT_DUP`), `del` позиционируется через `GET_BOTH` и удаляет текущее значение:

```
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"));

// Паттерн «два курсора»: `it` итерирует по дубликатам ключа,
// `del` удаляет текущую пару (позиция `it` при этом остаётся валидной).
const mdbx::slice target("target");
auto it = txn.open_cursor(multi);
auto del = txn.open_cursor(multi);
size_t deleted = 0;
// Первый dup ключа сохраняем, остальные удаляем.
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: удалить только текущее значение
        ++deleted;
    }
}
```

Полный код: [06-dupsort-delete.c++](#) · C-версия

Примеры к главе: [examples/c++/05-cursors.c++](#) · C-версия; [examples/c++/06-dupsort-delete.c++](#) · C-версия; сквозной проект: [config-store-06.c++](#).

4.1.8 6.8. Резюме главы 6

- Курсор — позиция в упорядоченном дереве; `mdbx_cursor_get(op)` управляет перемещением.

- `MDBX_SET_RANGE` — основа диапазонных запросов.
- После `cursor_del` позиция недействительна — перепозиционируйтесь.
- Для удаления в DUPSORT — два курсора или `bunch_delete`.

4.1.9 6.9. Упражнения

1. Напишите функцию обхода всех ключей от конца к началу (`MDBX_LAST + MDBX_PREV`).
2. Найдите «ключ, следующий за X» двумя способами: `SET_RANGE` и `SET_UPPERBOUND`.
3. Удалите каждый второй ключ в обходе и объясните, почему позиционирование обязано обновляться.

4.1.10 6.10. Чек-лист главы 6

- [] я умею открывать и закрывать курсор и понимаю, что он живёт не дольше своей транзакции;
- [] свободно пользуюсь `MDBX_FIRST` / `MDBX_LAST` / `MDBX_NEXT` / `MDBX_PREV` и `MDBX_SET_RANGE` для диапазонных запросов;
- [] помню, что после `mdbx_cursor_del()` позиция не определена, и всегда перепозиционируюсь;
- [] умею применять паттерн «два курсора» для безопасного удаления в DUPSORT;
- [] могу пройти таблицу «от конца к началу» (`MDBX_LAST + MDBX_PREV`) и отличить конец обхода (`MDBX_NOTFOUND`, `mdbx_cursor_eof()`) от ошибки.

4.1.11 6.11. Что дальше

Следующая глава расширяет модель данных: флаг `MDBX_DUPSORT` превращает таблицу в мультикарту «ключ → упорядоченное множество значений». Мы разберём формы хранения, флаги `MDBX_DUPFIXED`, `MDBX_INTEGERDUP`, `MDBX_REVERSEDUP`, навигацию по дубликатам и главный приём — инвертированный индекс «поле → список ID». Именно он станет основой вторичных индексов конфигулятора в главе 8.

4.2 Глава 7. Мультизначения и DUPSORT

4.2.1 7.1. Что такое DUPSORT

Обычная таблица хранит один ключ → одно значение. Флаг `MDBX_DUPSORT` превращает таблицу в **мультикарту**: один ключ → **упорядоченное множество значений**. Значение играет роль «второго ключа» со своим порядком сортировки.

Пример: теги пользователя `user:1001` → `{"admin", "staff", "vip"}`.

4.2.2 7.2. Формы хранения

Внутри (подробно в Томе III) движение значений выбирается по количеству:

- немного значений на ключ — плотная **суб-страница** (вложенная страница в листе);
- много значений — отдельное **вложенное B+tree**;
- фиксированный размер значений — специализированные **dupfix-страницы** плотной упаковки.

Важное следствие: значение-«мультизначение» не уходит на overflow-страницы, поэтому ограничено рамками ключа (порядка половины страницы).

4.2.3 7.3. Флаги DUPSORT

Флаг	Смысл
<code>MDBX_DUPSORT</code>	Таблица — мультикарта
<code>MDBX_DUPFIXED</code>	Все значения фиксированной длины (требует одинаковой длины! иначе <code>MDBX_BAD_VALSIZE</code>)
<code>MDBX_INTEGERDUP</code>	Значения — <code>uint32_t</code> / <code>uint64_t</code> нативного порядка (требует <code>DUPFIXED + DUPSORT</code>)
<code>MDBX_REVERSEDUP</code>	Обратный порядок сортировки значений

4.2.4 7.4. Операции с мультизначениями

```
/* добавить значение к ключу (MDBX_NODUPDATA защищает от дублей) */
mdbx_put(txn, dbi, &key, &data, MDBX_NODUPDATA);

/* найти конкретное значение */
mdbx_cursor_get(cur, &key, &data, MDBX_GET_BOTH); /* точно */
mdbx_cursor_get(cur, &key, &data, MDBX_GET_BOTH_RANGE); /* первое >= data */

/* навигация по значениям ключа */
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);

/* число значений ключа */
size_t count;
mdbx_cursor_count(cur, &count);
```

4.2.5 7.5. Инвертированные индексы: паттерн «ключ → список ID»

Классическое применение DUPSORT — **инвертированный индекс**: значение поля → список идентификаторов записей. Это «дешёвый вторичный индекс» без отдельной таблицы-связки.

Фрагмент (C, иллюстрация): полная компилируемая версия — ниже и в [examples/c++/08-inverted-index.c++](#).

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

/* при создании пользователя: 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);
```

Получение списка админов — это курсор по `idx_role` с ключом `"admin"` и `NEXT_DUP`.

Фрагмент из [examples/c++/08-inverted-index.c++](#) — построение инвертированного индекса «слово → список ID документов» поверх DUPSORT-таблицы (`value_mode::multi`):

```
{
    auto txn = env.start_write();
    auto docs = txn.create_map("docs", mdbx::key_mode::usual, mdbx::value_mode::single);
    // Индекс: ключ — слово, значения — ID документов (мультизначения).
    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 добавляет ID к слову
    }
    txn.commit();
}
```

Полный код: [08-inverted-index.c++](#) · C-версия

4.2.6 7.6. Полный пример: вторичный индекс на 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, "Анна", "admin"}, {1002, "Борис", "user"},
        {1003, "Виктор", "admin"}, {1004, "Галина", "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);
    }

    /* вывести всех админов через индекс */
    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("Админы:\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;
}
```

Примеры к главе: [examples/c++/07-dupsort.c++](#) · C-версия; [examples/c++/08-inverted-index.c++](#) · C-версия.

4.2.7 7.7. Резюме главы 7

- `MDBX_DUPSORT` = ключ → упорядоченное множество значений.
- `DUPFIXED` требует одинаковых длин; `INTEGERDUP` — целочисленные значения нативного порядка.
- Навигация по значениям: `FIRST_DUP/LAST_DUP/NEXT_DUP/PREV_DUP/GET_BOTH`.
- Инвертированный индекс на DUPSORT — дешёвый вторичный индекс.

4.2.8 7.8. Упражнения

1. Постройте индекс «по возрасту» и выведите всех пользователей старше 30.
2. Объясните, зачем `MDBX_NODUPDATA` при добавлении в DUPSORT-индекс.
3. Что произойдёт при попытке `DUPFIXED`-вставки значения другой длины?

4.2.9 7.9. Чек-лист главы 7

- [] я понимаю, что `MDBX_DUPSORT` — это «ключ → упорядоченное множество значений», и значение играет роль второго ключа;
- [] знаю, когда требуются `MDBX_DUPFIXED` (одинаковая длина значений) и `MDBX_INTEGERDUP`;
- [] умею находить и перебирать значения ключа: `MDBX_GET_BOTH`, `MDBX_GET_BOTH_RANGE`, `MDBX_FIRST_DUP / NEXT_DUP`, `mbx_cursor_count()`;
- [] добавляю значения с `MDBX_NODUPDATA`, чтобы не плодить дубликаты;
- [] умею построить инвертированный индекс «слово → список ID» поверх DUPSORT-таблицы.

4.2.10 7.10. Что дальше

Мультизначения — полуфабрикат; в главе 8 из него собирается законченный приём: вторичный индекс. Мы построим схему «основная таблица + индексные», научимся поддерживать их консистентность в одной транзакции, сравним DUPSORT-индекс с отдельной таблицей и познакомимся с составными ключами и удалением по индексу. Для сквозного проекта это шаг к «мини-ORM» с сущностью «пользователи».

4.3 Глава 8. Вторичные индексы

4.3.1 8.1. Зачем нужны вторичные индексы в KV-базе

В key-value базе нет SQL-индексов: данные ищутся только по первичному ключу. Чтобы искать «по другому полю», мы строим индекс **сами** — отдельную таблицу, где ключом становится искомое поле, а значением — первичный ключ исходной записи.

4.3.2 8.2. Паттерн: основная таблица + индексные

users:	user_id → {name, email, ...}	(первичная)
idx_email:	email → user_id	(индекс)
idx_by_role:	role → user_id (DUPSORT)	(индекс)

Поддержание консистентности — **на вашей стороне**: при каждом изменении `users` обновите все индексы в **той же транзакции** (иначе будет рассинхрон).

Фрагмент из `examples/c++/09-secondary-index.c++` — основная таблица и индекс обновляются в одной транзакции: при вставке пользователей — и при удалении (запись + её индексные ссылки):

```
{
    auto txn = env.start_write();
    auto users = txn.create_map("users", mdbx::key_mode::usual, mdbx::value_mode::single);
    // Индекс: ключ — роль, значения — ID пользователей (мультизначения).
    auto by_role = txn.create_map("by_role", mdbx::key_mode::usual, mdbx::value_mode::multi);

    // Вставка пользователей и поддержание индекса — в одной транзакции.
    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();
}

// Удаление пользователя вместе с его записью в индексе.
{
    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")); // значение конкретного ключа
    txn.commit();
}
```

Полный код: [09-secondary-index.c++](#)

Нюанс: открытие таблицы с неизвестными флагами. Если таблица могла быть создана другим кодом (и её постоянные флаги — `MDBX_DUPSORT`, `MDBX_INTEGERKEY` и т.п. — заранее неизвестны), передайте при открытии `MDBX_DB_ACCEDE`: вместо `MDBX_INCOMPATIBLE` таблица откроется со своими фактическими флагами. Узнать их можно через `mdbx_dbi_flags()` (точная версия — `mdbx_dbi_flags_ex()` — дополнительно возвращает state-биты таблицы: `MDBX_DBI_CREAT` / `MDBX_DBI_DIRTY` / `MDBX_DBI_FRESH` / `MDBX_DBI_STALE`). Это же касается повторного открытия таблиц в новой транзакции: хендл `MDBX_dbi` из старой транзакции не переносится в новую, а `mdbx_dbi_open()` в новой транзакции без тех же флагов вернёт `MDBX_INCOMPATIBLE`.

4.3.3 8.3. DUPSORT-индекс vs отдельная таблица

- **DUPSORT-индекс** («поле → список ID») идеален для «один ко многим»: список адресатов — это значения одного ключа. Быстрое добавление/удаление одного ID.
- **Отдельная таблица** («составной ключ → маркер») — когда нужно больше данных об отношении или составные условия.

4.3.4 8.4. Составные индексы

Составной ключ — конкатенация полей в одном ключе. Порядок байт критичен: чтобы числовое поле в составном ключе сравнивалось корректно, его нужно приводить к **big-endian** порядку (или использовать целочисленные типы нативного порядка с отдельным компаратором).

Фрагмент из [examples/c++/10-composite-key.c++](#) — упаковка двух полей в составной ключ с big-endian порядком байт: лексикографический порядок ключа совпадает с числовым порядком пары:

```
// Упаковка (x, y) в uint64_t так, чтобы побайтовый (лексикографический) порядок
// ключа совпадал с числовым порядком пары. Для этого используется big-endian
// представление беззнаковых полей фиксированной ширины.
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
}
```

Полный код: [10-composite-key.c++](#)

4.3.5 8.5. Удаление по индексу

Удалять через индекс нельзя напрямую — найдите первичный ключ по индексу, затем удалите запись и обновите индексы. Используйте безопасный паттерн итерации (глава 6).

4.3.6 8.6. Мини-ORM поверх libmdbx (сквозной проект)

Расширяем конфигуратор: добавляем сущность «пользователи» с индексом по имени.

```
/* 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; }

    /* индекс: 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;
}
```

Нюанс: атомарность гарантирует именно «всё или ничего»: если падение случится между операциями, транзакция откатится целиком. Это и есть цена правильных индексов — обновляйте их только в одной транзакции с основной записью.

Примеры к главе: [examples/c++/09-secondary-index.cpp](#); [examples/c++/10-composite-key.cpp](#); сквозной проект: [config-store-08.cpp](#).

4.3.7 8.7. Резюме главы 8

- Вторичный индекс — таблица «поле → первичный ключ», которую вы ведёте сами.
- Все изменения (данные + индексы) — в одной транзакции.
- DUPSORT-индекс для «один ко многим»; составные ключи для нескольких полей.
- Удаление записи — вместе с её индексными ссылками; иначе индекс «протухнет».
- Инвертированный индекс поверх DUPSORT — дешёвая альтернатива отдельной таблице-связке.

4.3.8 8.8. Упражнения

1. Добавьте индекс «по городу» и напишите функцию поиска.
2. Напишите `user_rename`, которая атомарно меняет имя и перестраивает `idx_name`.
3. Почему составные ключи требуют аккуратности с порядком байт?

4.3.9 8.9. Чек-лист главы 8

- [] я понимаю, что вторичный индекс — это таблица «поле → первичный ключ», которую веду сам;
- [] обновляю данные и все индексы в одной транзакции, помня о цене нарушения этого правила;
- [] осознанно выбираю между DUPSORT-индексом («поле → список ID») и отдельной таблицей;
- [] знаю, зачем в составных ключах числовые поля приводятся к `big-endian`;
- [] удаляю запись вместе с её индексными ссылками; иначе индекс «протухнет»;
- [] помню про `MDBX_DB_ACCEDE` на случай, когда флаги таблицы заранее неизвестны.

4.3.10 8.10. Что дальше

Индексы и данные упираются в следующий вопрос: как настроить само окружение. Глава 9 — о геометрии файла (`mdbx_env_set_geometry()`), лимитах `maxreaders / maxdbs`, флагах окружения, runtime-опциях `MDBX_opt_*` и статистике. Конфигуратору, который растёт от тестовой базы до рабочей, нужны управляемый размер файла и предсказуемые лимиты — иначе рано или поздно придёт `MDBX_MAP_FULL`.

4.4 Глава 9. Конфигурация окружения

4.4.1 9.1. Геометрия файла

Геометрия определяет, как растёт и сжимается файл базы:

```
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);
```

Параметры (значение `-1` = «по умолчанию/не менять»):

Параметр	Смысл
<code>size_lower</code>	Нижняя граница размера файла (не сжиматься ниже)
<code>size_now</code>	Начальный размер при создании
<code>size_upper</code>	Жёсткий предел ; при достижении и нехватке места — <code>MDBX_MAP_FULL</code>
<code>growth_step</code>	Шаг роста файла
<code>shrink_threshold</code>	Порог, при котором файл может усесться
<code>pagesize</code>	Размер страницы (256...65536; задаётся до первого <code>open</code>)

Внимание: геометрию задавайте **один раз, до `env_open`** (или при создании). Менять `upper` «на ходу» надёжно нельзя. Значение `upper` для новой БД движок выбирает сам (~золотое сечение ОЗУ, ограничено mmap-лимитом ≈ 140 ТБ на 64-бит); `TOO_LARGE` / `ENOMEM` возможны при явно чрезмерном `upper` (например, под ASAN/Valgrind) — всегда задавайте адекватный `upper` явно.

Историческая справка. `mdbx_env_set_mapsize(env, size)` — устаревшая обёртка над `mdbx_env_set_geometry()`, эквивалентная вызову `mdbx_env_set_geometry(env, size, size, size, -1, -1, -1)`. Как и геометрия, она меняет размер БД «на лету» (в том числе после `mdbx_env_open()`), но приравнивает нижнюю/текущую/верхнюю границы к одному значению — база теряет возможность авто-роста выше `size`. Предпочитайте `mdbx_env_set_geometry()` с отдельными параметрами. Аналогично `mdbx_env_sync()` — устаревший синоним `mdbx_env_sync_ex(env, force=true, nonblock=false)`, т.е. блокирующая полная синхронизация (глава 10).

4.4.2 9.2. maxreaders / maxdbs / pagesize

```
mdbx_env_set_maxreaders(env, readers); /* слоты таблицы читателей, до open */
mdbx_env_set_maxdbs(env, dbs);      /* лимит именованных таблиц */
/* размер страницы задаётся аргументом pagesize в mdbx_env_set_geometry() — отдельной функции нет */
```

Текущие значения читаются парами геттеров: `mdbx_env_get_maxreaders()` / `mdbx_env_get_maxdbs()`.

`maxreaders` — сколько потоков одновременно могут держать read-транзакции. Не занижайте его в приложениях с пулами потоков.

4.4.3 9.3. Флаги окружения

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

Ключевые флаги: `MDBX_RDONLY`, `MDBX_WRITEMAP`, `MDBX_NOMETASYNC`, `MDBX_SAFE_NOSYNC`, `MDBX_UTCERLY_NOSYNC`, `MDBX_NOSTICKYTHREADS`, `MDBX_EXCLUSIVE`.

Ещё два важных флага передаются **при `mdbx_env_open()`**, а не через `set_flags`:

- `MDBX_ACCEDE` — открыть базу, **уже используемую другим процессом в неизвестном режиме**: вместо ошибки `MDBX_INCOMPATIBLE` окружение откроется в совместимом с текущим использованием режиме (применяется к флагам долговечности, `MDBX_LIFORECLAIM` и `MDBX_NORDAHEAD`). Не влияет, если текущий процесс — единственный или все открытия read-only.
- `MDBX_EXCLUSIVE` — монопольное открытие: успех только если база не открыта никем другим.

Пара полезных функций окружения: `mdbx_env_warmup()` — «прогреть» файл БД в оперативной памяти (кэш страниц) до начала работы; `mdbx_env_get_fd()` — файловый дескриптор данных (для своих `fsync`/резервного копирования).

4.4.4 9.4. Runtime options (MDBX_opt_*)

Тонкая настройка через `mdbx_env_set_option` / `mdbx_env_get_option` :

Опция	Смысл
<code>MDBX_opt_rp_augment_limit</code>	Предел накопления списков при поиске последовательностей в GC
<code>MDBX_opt_gc_time_limit</code>	Тайм-лимит на поиск в GC в пишущей транзакции (1/65536 с)
<code>MDBX_opt_txn_dp_limit</code>	Лимит грязных страниц транзакции (по умолчанию $\approx 1/42$ ОЗУ)
<code>MDBX_opt_loose_limit</code>	Кэш loose-страниц (по умолчанию 64)
<code>MDBX_opt_writethrough_threshold</code>	Порог выбора <code>0_DSYNC</code> vs <code>fdatsync</code> (дефолт 2 грязных страницы; на Windows игнорируется)
<code>MDBX_opt_prefault_write_enable</code>	Упреждающая запись для WRITEMAP
<code>MDBX_opt_sync_bytes</code> / <code>MDBX_opt_sync_period</code>	Авто-синхронизация (дефолты: ~ 1.05 ГБ / 42.42 с; явный 0 = отключено)
<code>MDBX_opt_merge_threshold</code>	Порог слияния страниц (16.16 %; дефолт 33%, диапазон [12.5%..50%])
<code>MDBX_opt_prefer_waf_insteadof_balance</code>	Предпочтение грязного соседа при слиянии (дефолт true с 2026-01-04)

Фрагмент из `examples/c++/11-geometry-options.c++` — задание геометрии до `open` (через `create_parameters().set_geometry(geo)`) и runtime-опций после:

```
// Геометрия: нижняя/текущая/верхняя границы размера БД, шаг роста,
// порог сжатия. Здесь — компактная БД для примера.
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-опции: устанавливаем и читаем обратно.
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);
```

Полный код: [11-geometry-options.c++](#) · C-версия

4.4.5 9.5. Выбор размера страницы

Страница	Когда
4 КБ	По умолчанию; универсально
8 КБ	Страховка от большой фрагментированной GC
64 КБ	Большие значения (меньше overflow-прогонов); дороже мелкие записи

Учтите: значение, не помещающееся в одну страницу, хранится одним неразрывным куском на последовательных страницах и требует прогона **подряд идущих** свободных страниц — его выделение дороже и при фрагментации может привести к `MDBX_MAP_FULL` . Поэтому выбирая страницу, ориентируйтесь на реальный (типичный и максимальный) размер значений: чем больше доля значений, не помещающихся в одну страницу, тем выгоднее крупная страница. Если длинное значение нельзя уменьшить — разбейте его на чанки, помещающиеся в одну страницу (`mdbx_limits_valsize4page_max()`); вручную такое разбиение требуется только до появления планируемого **потокowego API для длинных значений** («Streaming BLOBs»). Детали — [Ограничения и подводные камни](#).

4.4.6 9.6. Статистика

```
mdbx_env_info_ex(env, &info, sizeof(info)); /* геометрия, меты, счётчики */
mdbx_env_stat_ex(env, txn, &stat, sizeof(stat)); /* размеры, страницы */
mdbx_dbi_stat_ex(txn, dbi, &dst, sizeof(dst), 0); /* статистика таблицы */
```

Фрагмент из [examples/c++/12-env-stat.c++](#) — чтение статистики окружения (аналоги `mdbx_env_stat_ex()` / `mdbx_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(); // аналог 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(); // аналог 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";
```

Полный код: [12-env-stat.c++](#) · [C-версия](#)

Примеры к главе: [examples/c++/11-geometry-options.c++](#) · [C-версия](#); [examples/c++/12-env-stat.c++](#) · [C-версия](#); сквозной проект: [config-store-09.c++](#).

4.4.7 9.7. Резюме главы 9

- Геометрия (`lower/now/upper/growth/shrink/pagesize`) задаётся до `open`.
- `upper` — жёсткий предел; заниженный → `MDBX_MAP_FULL`, явно завышенный (например, под ASAN/Valgrind) → `TOO_LARGE` / `ENOMEM`. Дефолт для новой БД движок выбирает сам (≈ золотое сечение ОЗУ).
- `maxreaders` / `maxdbs` / `pagesize` — до первого `open`.
- Опции `MDBX_opt_*` — тонкая настройка GC/спилла/синхронизации.

4.4.8 9.8. Упражнения

1. Задайте геометрию «от 1 МБ до 4 ГБ, шаг 64 МБ» до `open` и проверьте `env_info`.
2. Что вернёт `env_open` при явно завышенном `upper` (например, 140 ТБ на 64-бит)? Зафиксируйте код ошибки.
3. Создайте базу с страницей 64 КБ и сравните с 4 КБ на чтении больших значений.

4.4.9 9.9. Чек-лист главы 9

- [] я задаю геометрию (`size_lower` / `size_now` / `size_upper` / `growth_step` / `shrink_threshold`) до `env_open`;
- [] понимаю, что `size_upper` — жёсткий предел: при исчерпании приходит `MDBX_MAP_FULL`, а чрезмерный `upper` даёт `TOO_LARGE` / `ENOMEM`;
- [] знаю, где задаются `maxreaders`, `maxdbs` и размер страницы;
- [] различаю ключевые флаги окружения (`MDBX_WRITEMAP`, `MDBX_SAFE_NOSYNC`, `MDBX_EXCLUSIVE`, `MDBX_ACCEDE`) и основные `MDBX_opt_*`-опции;
- [] умею снимать статистику через `mdbx_env_info_ex()` / `mdbx_env_stat_ex()` и использовать её для диагностики.

4.4.10 9.10. Что дальше

Настроенное окружение — ещё не всё: нужно решить, что происходит при коммите. Глава 10 разбирает режимы долговечности: `weak/steady` мета, чем рискуют `MDBX_NOMETASYNC` и `MDBX_SAFE_NOSYNC`, зачем нужна авто-синхронизация и как режимы различаются на macOS, Linux и Windows. Выбор режима напрямую определяет скорость и надёжность конфигуратора.

4.5 Глава 10. Режимы долговечности

4.5.1 10.1. Что значит «закоммитить»

Строгий коммит должен: (1) записать на диск данные (новые версии страниц) и (2) записать мету — сделать снимок «видимым» для будущих открытий. Между этими событиями различают:

- **weak (слабая) мета** — данные в файле, но не гарантированно на постоянном носителе;
- **steady (устойчивая) мета** — данные сброшены на диск; снимок переживает сбой системы.

4.5.2 10.2. Режимы

Режим	Поведение	Риск
<code>MDBX_SYNC_DURABLE</code> (по умолчанию)	Данные → flush → мета → flush	Нет: полный ACID
<code>MDBX_NOMETASYNC</code>	Данные флашатся, мета — отложено	Потеря последних коммитов при сбое
<code>MDBX_SAFE_NOSYNC</code>	Ничего не флашится сразу, сохраняется предыдущий steady	Откат к последнему steady; рост файла (эффект квази-долгого читателя)
<code>MDBX_UTTERLY_NOSYNC</code>	Никаких флашей, никаких steady-гарантий	База может не пережить сбой
<code>MDBX_WRITEMAP</code>	Запись через mmap (+msync)	Сочетается с режимами выше

Фрагмент из `examples/c++/13-sync-modes.c++` — смена sync-режима между открытиями одной и той же БД: полная долговечность (`robust_synchronous`) → `NOMETASYNC` (`half_synchronous_weak_last`):

```
// Прогон 1: полная долговечность (sync перед ответом коммита).
{
    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); // явный сброс на диск
    std::cout << "durable: ok\n";
}

// Прогон 2: NOMETASYNC (метаданные не синхронизируются при каждом коммите).
{
    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();
}
```

Полный код: [13-sync-modes.c++ · C-версия](#)

4.5.3 10.3. Когда какой режим

- **По умолчанию** — максимум надёжности (ACID). Для баз, где потеря коммита неприемлема.
- `MDBX_NOMETASYNC` — высоконагруженная запись, где можно потерять «хвост» последних коммитов.
- `MDBX_SAFE_NOSYNC` — запись может ускориться в разы (до 10×), но: файл растёт (страницы новее steady не переиспользуются до нового steady-point), при сбое — откат к последнему steady.
- `MDBX_UTTERLY_NOSYNC` — только для некритичных данных/кэшей.
- `MDBX_WRITEMAP` — для больших транзакций и тонкой настройки записи; сочетается с флагами выше (например, `SAFE_NOSYNC` | `WRITEMAP`).

4.5.4 10.4. Авто-синхронизация

`MDBX_SAFE_NOSYNC` без управления ростом — источник бесконтрольного роста файла. Управляйте им:

```
mdbx_env_set_syncbytes(env, 512ULL * 1024 * 1024); /* flush после 512 МБ записи */
mdbx_env_set_syncperiod(env, 3 << 16); /* 3 секунды в формате 16.16: 3<<16 = 3·65536 единиц (сырое значение 3 ≈ 46 мкс) */
```

И вручную из отдельного потока: `mdbx_env_sync_ex(env, force, nonblock)` — при `nonblock=true` вызов возвращается сразу (асинхронно), завершение проверяется дешёвым `mdbx_env_sync_poll(env)`.

4.5.5 10.5. Платформенные нюансы

- **macOS:** по умолчанию `fcntl(F_FULLFSYNC)` — максимум долговечности, но медленно; `MDBX_APPLE_SPEED_INSTEADOF_DURABILITY` меняет приоритет на скорость.
- **Linux:** `boot_id` используется для контроля отката слабых мет (важно в LXC).
- **Windows:** файловые блокировки `LockFileEx` медленнее именованных мьютексов (влияет на мелкие транзакции); `MDBX_WRITEMAP` помогает. Сквозная запись используется всегда, `MDBX_NOMETASYNC` переключает на «ленивая + flush» (попуг `writethrough` игнорируется).

4.5.6 10.6. Пример: переключение режима

Фрагмент (C, иллюстрация):

```
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); /* включаем */
mdbx_env_set_syncbytes(env, 256ULL * 1024 * 1024);
mdbx_env_open(env, "./safe.mdbx", MDBX_NOSUBDIR, 0664);
```

Примеры к главе: [examples/c++/13-sync-modes.c++](#) · [C-версия](#); сквозной проект: [config-store-10.c++](#).

4.5.7 10.7. Резюме главы 10

- Режимы долговечности различаются тем, что флашится: данные, мета, ничего.
- `SAFE_NOSYNC` — скорость ценой роста файла и отката к steady при сбое.
- Авто-синхронизация (`syncbytes / syncperiod`) обязательна для `SAFE_NOSYNC`.
- Платформенные различия (`F_FULLFSYNC`, `boot_id`, `LockFileEx`) влияют на выбор.

4.5.8 10.8. Упражнения

1. Сравните скорость коммитов в `DURABLE` vs `SAFE_NOSYNC` на 10 000 мелких транзакциях.
2. Наблюдайте рост файла в `SAFE_NOSYNC` без авто-sync — и после настройки `syncbytes`.
3. Объясните, почему `UTTERLY_NOSYNC` — «максимальный риск»: что именно может не пережить сбой.

4.5.9 10.9. Чек-лист главы 10

- [] я понимаю, что строгий коммит — это «данные на диск + мета на диск», и различаю weak/steady мета;
- [] знаю риски каждого режима: `MDBX_SYNC_DURABLE`, `MDBX_NOMETASYNC`, `MDBX_SAFE_NOSYNC`, `MDBX_UTTERLY_NOSYNC` и сочетания с `MDBX_WRITEMAP`;
- [] для `MDBX_SAFE_NOSYNC` настраиваю авто-синхронизацию (`syncbytes / syncperiod`), чтобы контролировать рост файла;
- [] умею переключать режимы и вызывать `mdbx_env_sync_ex()` вручную;
- [] помню платформенные нюансы: `F_FULLFSYNC` на macOS, `boot_id` на Linux, `LockFileEx` на Windows.

4.5.10 10.10. Что дальше

Режимы долговечности задали правила записи; следующая глава — о работе с базой из нескольких потоков. Глава 11 разбирает sticky threads, флаг `MDBX_NOSTICKYTHREADS` и его ловушки, TLS-слоты читателей, `fork()`, клонирование транзакций и парковку долгоживущих читателей. Конфигуратору с пулом потоков или корутинами без этих знаний не обойтись.

4.6 Глава 11. Многопоточность

4.6.1 11.1. Модель потоков: sticky threads по умолчанию

По умолчанию транзакция «прилипает» к потоку, который её создал. Другие потоки не могут использовать этот объект — получите `MDBX_THREAD_MISMATCH`. Читатели (разные транзакции в разных потоках) работают параллельно и без блокировок.

Фрагмент из `examples/c++/14-multithreading.c++` — писатель коммитит новые записи, а читатели на клонах одного снимка (`base.clone()`) продолжают видеть стабильную картину:

```
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;
});

// Два читателя: каждый работает со своим клоном базового снимка.
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(); // клон снимка для этого потока
        auto table = clone.open_map(nullptr);
        snapshot_count[r] = clone.get_map_stat(table).ms_entries;
        clone.abort();
    });
}
```

Полный код: [14-multithreading.c++](#)

4.6.2 11.2. `MDBX_NOSTICKYTHREADS`

Флаг `MDBX_NOSTICKYTHREADS` позволяет передавать транзакцию между потоками — нужно для пулов потоков, корутин, `async-runtime` (`tokio` и т.п.), где операция может продолжиться в другом потоке.

Внимание: с `NOSTICKYTHREADS` функции, требующие write-блокировку среды (`mdbx_env_set_option`, `set_flags`, `set_geometry`, `sync`, `stat`, `defrag`, `close`), могут deadlock'нуться, если пишущая транзакция «переехала» в другой поток, который ждёт эту блокировку. Сериализуйте такие вызовы (`channel/mutex`), либо держите их в потоке-владельце транзакции.

4.6.3 11.3. Регистрация потоков и TLS

Слот читателя привязывается к потоку через TLS. При корректном завершении потока деструктор очищает слот. Если поток завершается «нештатно» (или TLS-деструктор не сработал — известные баги `glibc #21031/#21032`), слот может «утечь». Для потоков без TLS — явная регистрация:

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

Диагностика утечек — `mdbx_reader_check(env, &dead)` (Том V, глава 30).

4.6.4 11.4. fork()

После `fork()` наследник **не наследует** mmap- и record-блокировки. Использовать окружение в дочернем процессе можно только после:

```
mdbx_env_resurrect_after_fork(env);
```

Вызывается один раз в наследнике, не в родителе.

4.6.5 11.5. Клонирование транзакций

`mdbx_txn_clone()` размножает read-only транзакцию — несколько обработчиков на одном снапшоте без повторного сканирования RLT (RLT — таблица зарегистрированных читателей, внутренняя структура libmdbx; подробно в томе III, гл. 14) — что полезно для параллельной обработки.

4.6.6 11.6. Парковка и вытеснение

Долгоживущий читатель «пинит» **детент** (старейший активный снапшот; подробно — Том III, глава 14) и мешает переработке страниц. Решения:

```
mdbx_txn_park(txn); /* освободить слот читателя, сохранив хендл */
mdbx_txn_unpark(txn); /* возобновить (при вытеснении — с restart_if_ousted=true) */
```

Если писателю не хватает пространства, припаркованные читатели **выселяются** (ousted): слот переводится из `PARKED` в `OUSTED`; при следующем обращении читатель либо перезапустится на свежем снапшоте, либо получит `MDBX_OUSTED`.

Внимание: после парковки снапшот не удерживается — разыменовывать указатели, полученные до парковки, запрещено (страницы могут быть переиспользованы).

4.6.7 11.7. Несколько сред; MDBX_EXCLUSIVE

- Нельзя открывать одну и ту же БД дважды в одном процессе (защита от гонок; legacy-режим — `MDBX_DBG_LEGACY_MULTIOOPEN`).
- Несколько разных баз (разные файлы) в одном процессе — можно.
- `MDBX_EXCLUSIVE` — монопольное открытие (только этот процесс).

Примеры к главе: `examples/c++/14-multithreading.c++`; `examples/c++/15-fork-resurrect.c++`; `examples/c++/16-parking.c++`; сквозной проект: `config-store-11.c++`.

4.6.8 11.8. Резюме главы 11

- Sticky threads по умолчанию; `NOSTICKYTHREADS` для пулов/корутин, но с риском deadlock в write-функциях.
- TLS-деструкторы чистят слоты читателей; при нештатных завершениях — `reader_check` /регистрация.
- `resurrect_after_fork` обязателен в наследнике.
- Парковка/вытеснение — решение для долгих чтений.

4.6.9 11.9. Упражнения

1. Напишите пример с двумя потоками: один пишет, второй читает. Проверьте отсутствие блокировок читателя.
2. Воспроизведите сценарий: write-транзакция начата в потоке А, поток В вызывает `env_sync` с `NOSTICKYTHREADS` — зафиксируйте deadlock.
3. Придумайте, когда `txn_clone` экономит ресурсы по сравнению с несколькими `txn_begin`.

4.6.10 11.10. Чек-лист главы 11

- [] я понимаю, что транзакция по умолчанию привязана к потоку, и узнаю `MDBX_THREAD_MISMATCH`;
- [] знаю, когда оправдан `MDBX_NOSTICKYTHREADS` и какие write-функции при нём могут deadlock'нуться;

- [] понимаю, как очищаются TLS-слоты читателей и чем помогают `mdbx_thread_register()` / `mdbx_thread_unregister()`;
- [] после `fork()` вызываю `mdbx_env_resurrect_after_fork()` в наследнике, а не в родителе;
- [] умею клонировать read-only транзакции (`mdbx_txn_clone()`) для параллельной обработки одного снимка;
- [] использую `mdbx_txn_park()` / `mdbx_txn_unpark()` для долгоживущих читателей и помню, что припаркованный снимок не удерживается.

4.6.11 11.11. Что дальше

Остался последний практический навык — реакция на ошибки. Глава 12 систематизирует коды возврата: ожидаемые состояния (`MDBX_KEYEXIST`, `MDBX_NOTFOUND`), проблемы пространства (`MDBX_MAP_FULL` vs `MDBX_TXN_FULL`), ошибки повреждения и нарушения дисциплины, а также retry-циклы для конкурентных сценариев. После неё конфигурактор будет устойчив и к сбоям, и к конкуренции.

4.7 Глава 12. Обработка ошибок

4.7.1 12.1. Коды возврата

- `MDBX_SUCCESS` (0) — успех.
- Отрицательные `MDBX_*` — ошибки (`MDBX_NOTFOUND`, `MDBX_KEYEXIST`, `MDBX_BAD_DBI`, ...).
- Специальные результаты: `MDBX_RESULT_TRUE` (операция выполнена), `MDBX_RESULT_FALSE` (нет данных/не требуется).
- Системные коды (errno-подобные) могут приходить из ОС — они отрицательные и выводятся как есть.

4.7.2 12.2. `mdbx_strerror_r` vs `mdbx_strerror`

`mdbx_strerror(rc)` использует общий буфер — **не потокобезопасен**. В многопоточном коде используйте:

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

4.7.3 12.3. `MDBX_MAP_FULL`: что делать

`MDBX_MAP_FULL` — база упёрлась в `upper`, а переиспользовать нечего (GC пуст/заморожен). Порядок:

1. **Abort текущей write-транзакции** (продолжение после изменения геометрии → `MDBX_BAD_TXN`).
2. Проверить долгих читателей (`mdbx_stat -r retained`; `mdbx_env_info_ex`).
3. Задать/поднять `upper` **до создания/при открытии**.
4. Установить HSR-колбэк (HSR — Handle-Slow-Readers, механизм вытеснения застрявших читателей; Том V, глава 29).

4.7.4 12.4. `MAP_FULL` vs `TXN_FULL`

- `MDBX_MAP_FULL` — исчерпано пространство файла (геометрия).
- `MDBX_TXN_FULL` — исчерпан внутренний лимит грязных/retired-страниц транзакции. Лечение — коммитить раньше, дробить транзакции, проверить `mdbx_txn_info()`.

4.7.5 12.5. Ожидаемые коды

- `MDBX_KEYEXIST` — ключ уже есть (при `NOOVERWRITE`).
- `MDBX_NOTFOUND` — ключа/значения нет. Это штатные состояния, а не ошибки: обрабатывайте их ветвлением, а не паникой.

4.7.6 12.6. Ошибки повреждения и восстановления

- `MDBX_BAD_TXN`, `MDBX_BAD_DBI` — невалидный хендл (нарушение дисциплины/закрытая транзакция).
- `MDBX_EBADSIGN` — повреждена сигнатура.
- `MDBX_WANNA_RECOVERY` — БД требует восстановления при read-only открытии; откройте read-write или `mdbx_env_open_for_recovery()`.
- `MDBX_MVCC_RETARDED` — читатель старше актуального снапшота.
- `MDBX_LAGGARD_READER` — читатель отстал (см. HSR).

4.7.7 12.7. Нарушение дисциплины

- `MDBX_THREAD_MISMATCH` — транзакцию используют из чужого потока.
- `MDBX_TXN_OVERLAPPING` — пересечение read/write транзакций в одном потоке.
- `MDBX_BUSY` — конфликт владения.
- `MDBX_BAD_RSLT` — неверный слот читателя. Это сигналы архитектурной ошибки, а не «случайных глюков».

4.7.8 12.8. Стратегии повторных попыток (retry loops)

Для конкурентных сценариев («сравни-и-замени»):

Фрагмент (C, иллюстрация):

```
for (;;) {
    mdbx_txn_begin(env, NULL, MDBX_TXN_READWRITE, &txn);
    /* читаем условие */
    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; }
    /* пишем новое значение */
    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) { /* увеличить upper/подождать читателей */ continue; }
    return rc;
}
```

Фрагмент из [examples/c++/17-error-handling.c++](#) — retry-цикл на «занятость» окружения (`MDBX_BUSY` при конкуренции процессов / `EAGAIN` внутри процесса): открытие в исключительном режиме, пока БД уже открыта, повторяется ограниченное число раз:

```
// Retry-стратегия: попытка открыть окружение в исключительном режиме, пока
// оно уже открыто текущим процессом, даёт «занятость» (MDBX_BUSY при
// конкуренции процессов или системную ошибку EAGAIN внутри процесса).
// Такой отказ перехватывается и обрабатывается повтором с ограничением.
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";
```

Полный код: [17-error-handling.c++](#) · C-версия

Примеры к главе: [examples/c++/17-error-handling.c++](#) · C-версия; сквозной проект: [config-store-12.c++](#).

4.7.9 12.9. Резюме главы 12

- Успех = 0; ошибки отрицательные; `RESULT_TRUE/FALSE` — специальные результаты.
- `mdbx_strerror_r` — потокобезопасный вариант.
- `MAP_FULL` лечится геометрией+HSR; `TXN_FULL` — ранними коммитами.
- `KEYEXIST` / `NOTFOUND` — ожидаемые состояния.

- Коды дисциплины (`THREAD_MISMATCH` и др.) указывают на архитектурные ошибки.

4.7.10 12.10. Упражнения

1. Напишите обработчик `MDBX_KEYEXIST` в сквозном проекте: при конфликте — читать и решать.
2. Опишите, чем `MAP_FULL` отличается от `TXN_FULL`, на примерах.
3. Сделайте getry-цикл вокруг `cfg_set` со стратегией «читать-проверить-записать».

4.7.11 12.11. Чек-лист главы 12

- [] я правильно читаю коды возврата: 0 — успех, отрицательные — ошибки, `MDBX_RESULT_TRUE` / `MDBX_RESULT_FALSE` — специальные результаты;
- [] в многопоточном коде использую `mdbx_strerror_r()`, а не `mdbx_strerror()`;
- [] знаю порядок действий при `MDBX_MAP_FULL` (abort, проверка читателей, поднять `upper` до открытия, HSR) и отличаю его от `MDBX_TXN_FULL`;
- [] обрабатываю `MDBX_KEYEXIST` и `MDBX_NOTFOUND` ветвлением, а не паникой;
- [] воспринимаю коды дисциплины (`MDBX_THREAD_MISMATCH`, `MDBX_TXN_OVERLAPPING`) как сигнал архитектурной ошибки;
- [] умею писать getry-цикл «читать-проверить-записать» для конкурентных сценариев.

4.7.12 12.12. Что дальше

Том II дал полный практический инструментарий, но многие его правила («закрывайте read-транзакции», «геометрия — до open», «`SAFE_NOSYNC` растит файл») выглядели как требования без объяснений. Том III открывает внутренние механизмы libmdbx: B+tree и mmap, MVCC, конвейер коммита, GC свободных страниц, геометрию и восстановление после сбоев. С ними правила превращаются в понимание «почему».

4.8 Итог тома

Том II дал вам практический инструментарий: курсоры, DUPSORT и индексы, конфигурацию окружения, режимы долговечности, многопоточность и обработку ошибок. Сквозной проект-конфигуратор теперь — полноценное приложение с индексами.

Что дальше: Том III — внутренние механизмы: B+tree и mmap, MVCC, конвейер коммита, GC, геометрия, вложенные транзакции, файл блокировок, долговечность и восстановление.

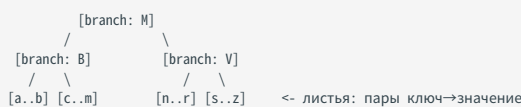
5. Том III. Внутренние механизмы

Уровень: для понимающих архитектуру; мост между «как пользоваться» и «как настраивать и отлаживать». **Цель тома:** вы объясняете поведение libmdbx через её устройство: B+tree и mmap, MVCC, конвейер коммита, GC, геометрию, вложенные транзакции, файл блокировок и восстановление.

5.1 Глава 13. Архитектура хранения: B+tree и mmap

5.1.1 13.1. B+tree: ветви, листья, overflow

Данные libmdbx хранит в **B+tree** — сбалансированном дереве, где все значения лежат в листьях, а внутренние узлы (branch) содержат только разделители-ключи и указатели на дочерние страницы.



Свойства:

- поиск — спуск от корня к листу, $O(\log N)$;
- все листья одного уровня связаны — обход диапазона идёт по соседям;
- при переполнении листа — **split** (деление пополам), рекурсивно вверх, включая новый корень;
- при опустошении ниже порога — **rebalance/merge** с соседом (порог `merge_threshold`).

Значения, не помещающиеся в лист (больше примерно половины страницы), уходят на **overflow-страницы**: контигуальный прогон `P_LARGE`-страниц; в листе остаётся указатель `{pgno, npages}`.

Когда значение не помещается даже в одну overflow-страницу (`npages > 1`), прогон должен быть последовательностью **подряд идущих** страниц нужной длины. Это главный практический момент: выделение такого прогона дороже выделения любой отдельной свободной страницы (движок может сканировать и сливать записи GC или дописывать в конец файла), а перезапись значения переписывает весь прогон целиком (CoW всех его страниц, фрагментация, churn в GC). Смягчить проблему можно, разбив длинное значение на чанки, помещающиеся в одну страницу (`mdbx_limits_vsize4page_max()`); вручную такое разбиение требуется только до появления планируемого **потокowego API для длинных значений** («Streaming BLOBs»). Подробности и способы смягчения — [Ограничения и подводные камни](#).

5.1.2 13.2. Почему B+tree, а не B-tree или LSM

- **B-tree** — данные и в листьях, и в ветвях: поиск может остановиться на любом уровне, но обход требует перехода вверх/вниз. B+tree с данными только в листьях даёт предсказуемый спуск и линейный обход.
- **LSM** (LevelDB/RocksDB) оптимизирован на запись (append-only + фоновое сжатие), но чтение страдает от многоуровневых просмотров. B+tree даёт честный $O(\log N)$ на чтение и запись без фоновых процессов.

5.1.3 13.3. Страница — единица хранения и I/O

Размер страницы — 256...65536 байт (по умолчанию 4096), выбирается при создании базы и далее неизменен. Типы:

Тип	Назначение
branch	Внутренний узел: разделители и указатели
leaf	Лист: пары ключ→значение
large / overflow	Контигуальный прогон больших значений
meta	Одна из трёх мета-страниц (снэпшот состояния базы)
dupfix / subpage	Плотная упаковка мультизначений

Каждая страница несёт **метку транзакции** (`txnid`), создавшей её текущую версию — это основа CoW и MVCC.

С базой можно связать пользовательский **канарейник** (`MDBX_canary`, 32 байта — четыре `uint64_t`: `x`, `y`, `z`, `v`; поле `v` всегда равно номеру транзакции): `mdbx_canary_put(txn, &canary)` записывает его в мету, `mdbx_canary_get(txn, &canary)` читает. Обновлённые значения видны другим процессам только после коммита. Это не данные таблиц, а произвольная метка приложения (например, версия схемы), переживающая открытия базы и доступная даже при монтировании только для чтения.

5.1.4 13.4. mmap: виртуальная память как окно в файл

Файл отображается в адресное пространство целиком. Чтение — разыменование указателя; страницу подтянет ядро (page cache). Отдельного буферного кэша в библиотеке нет.

Следствия:

- чтение без копирования (вы получаете адрес прямо в `mmap`);
- страничный кэш ОС управляет резидентностью;
- **объём базы ограничен адресным пространством** (на 32-бит — реально ~1–3 ГБ);
- при БД ≫ ОЗУ растут накладные на PTE в ядре.

Для области `mmap` применяется `madvise(MADV_NOHUGEPAGE)` (отказ от THP) — огромные страницы мешают точной оценке резидентности.

Фрагмент из `examples/c++/18-tree-height.c++` — оценка дерева через `mdbx_env_stat_ex()`: высота, страницы leaf/branch/overflow и число записей; значение больше размера страницы создаёт overflow-страницы:

```
{
    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"));
    // Значение больше размера страницы (4096) уходит на overflow-страницы.
    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";
```

Полный код: [18-tree-height.c++](#) · C-версия.

5.1.5 13.5. Инвариант достижимости

Каждая не-мета страница до указателя `first_unallocated` достижима **ровно один раз**: либо принадлежит какому-то дереву (обычной, главной или GC-таблице), либо перечислена в записи GC. На этом инварианте строится аудит целостности и инструмент проверки базы (`mdbx_chk`).

Примеры к главе: [examples/c++/18-tree-height.c++](#) · C-версия.

5.1.6 13.6. Резюме главы 13

- B+tree: данные в листьях, ветви — разделители; split/merge поддерживают баланс.
- Большие значения — overflow-прогоны страниц.
- mmap вместо буферного кэша: чтение без копирования, плата — адресное пространство/PTE.
- Инвариант достижимости — фундамент целостности.

5.1.7 13.7. Упражнения

1. Объясните, почему `mdbx_get` на 64-бит практически не зависит от объёма базы.
2. Что меняется для читателя, если дерево выросло с высоты 3 до высоты 5?

5.1.8 13.8. Чек-лист главы 13

- [] Могу объяснить, почему все значения лежат в листьях B+tree, а в ветвях — только разделители, и что это даёт для поиска и обхода диапазонов.
- [] Знаю, куда уходит значение, не помещающееся в лист (overflow-прогон), и что остаётся в листе вместо него.
- [] Понимаю, почему значение на несколько overflow-страниц требует прогона подряд идущих страниц, и чего это стоит при выделении и перезаписи.
- [] Понимаю, почему libmdbx не заводит собственный буферный кэш, и какие следствия это имеет (чтение без копирования, резидентность управляется ядром, рост PTE при БД $\gg$ ОЗУ).
- [] Могу назвать ограничение объёма базы адресным пространством и его следствие на 32-битных платформах.
- [] Знаю назначение канарейки (`MDBX_canary`) и что поле `v` всегда равно номеру транзакции.
- [] Умею сформулировать инвариант достижимости и объяснить, что на нём построено (аудит целостности, `mdbx_chk`).

5.2 Глава 14. MVCC и снапшоты

5.2.1 14.1. Версионность страниц, снапшот читателя

Каждая страница помечена номером транзакции, создавшей её текущую версию. Читатель при старте фиксирует снапшот — состояние базы на момент начала транзакции. Всё, что он видит после, — непротиворечивая картина этого момента, независимо от последующих коммитов.

5.2.2 14.2. Тройка мета-страниц и двухфазный коммит

В базе всегда **три мета-страницы** (`NUM_METAS = 3`). Это позволяет держать два валидных снапшота («свежий» и «steady» — с гарантированно сброшенными на диск данными) и один хвостовой слот для перезаписи.

Обновление меты — **двухфазное**: сначала инвалидируется половина (запись номера транзакции с обнулением второй), затем заполняются поля и валидируется вторая половина. Читатель видит либо старую цельную мету, либо новую — никогда «полуобновлённую».

5.2.3 14.3. Конечный автомат тройки

Состояние каждой меты кодируется независимо; полная таблица переходов покрывает **216 (6³)** комбинаций и исчерпывающе проверяется внутренней валидацией. Это защита от «невозможных» состояний после сбоев.

5.2.4 14.4. Таблица читателей (RLT)

Читатель регистрирует номер снимка в **таблице читателей** (в файле блокировок). Слот содержит: номер снимка, pid/tid владельца, и поля, описывающие, сколько страниц читатель «пинит».

Старейший активный снимок — **детент** — вычисляется сканированием RLT (lock-free, с кэшированием). Детент — водораздел переиспользования страниц (глава 16).

5.2.5 14.5. Wait-free чтение и safe64

На операциях чтения читатель **не берёт блокировок** — он просто ходит по mmap. Единственное «записывающее» действие — регистрация слота при старте транзакции.

64-битные поля (номера транзакций в RLT и мете) читаются через специальный атомарный протокол **safe64**: значения пишутся «сначала младшее слово, затем старшее», чтение повторяется при обнаружении «обрывка». Это защита от разорванных 64-битных чтений на слабых моделях памяти.

5.2.6 14.6. Писатель: глобальный мьютекс, front txnid, грязный список

В любой момент времени **одна write-транзакция** (глобальный мьютекс писателя в файле блокировок). При старте писатель получает **предварительный номер** (`front txnid = txnid + 1`): новые и CoW-копии страниц метятся им и не видны читателям до коммита. Изменённые страницы скапливаются в **грязном списке** (DPL).

Фрагмент из `examples/c++/19-readers-lag.c++` — читатель удерживает снимок; коммиты писателя увеличивают его «отставание» (lag), а `mdbx_reader_list` показывает слоты читателей с txnid и lag:

```
// Коммиты после создания снимка увеличивают lag читателя.
for (int i = 3; i < 8; ++i) {
    auto txn = env.start_write();
    auto wtable = txn.open_map(nullptr);
    txn.insert(wtable, mdbx::slice("k" + std::to_string(i)), mdbx::slice("v"));
    txn.commit();
}

// Перечислить читателей: слот читателя имеет lag > 0.
struct visitor {
    int operator()(const mdbx::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 mdbx::continue_loop;
    }
} v;
env.enumerate_readers(v);
```

Полный код: [19-readers-lag.c++](#).

Примеры к главе: [examples/c++/19-readers-lag.c++](#).

5.2.7 14.7. Резюме главы 14

- MVCC: страница помечена txnid; читатель видит свой снимок.
- Тройка мет с двухфазным обновлением и конечным автоматом 216 состояний.
- RLT и детент — основа безопасности переиспользования.
- Wait-free чтение; safe64 против разорванных чтений.
- Один писатель; front txnid скрывает незакоммиченное.

5.2.8 14.8. Упражнения

1. Зачем нужны именно три мета-страницы, а не две?
2. Что произойдёт, если писатель упадёт в середине двухфазного обновления меты?

5.2.9 14.9. Чек-лист главы 14

- [] Объясняю, почему читатель видит непротиворечивую картину независимо от последующих коммитов (снимок + метка `txnid` на страницах).
- [] Знаю, зачем нужны именно три мета-страницы и какие два валидных снимка они держат.
- [] Могу описать двухфазное обновление меты и объяснить, почему читатель никогда не видит «полуобновлённую» мету.
- [] Понимаю, как детент вычисляется сканированием RLT и почему он — водораздел перенесения страниц.
- [] Знаю, что чтение `wait-free` и не берёт блокировок, и зачем нужен протокол `safe64`.
- [] Могу объяснить, почему незакоммиченные страницы писателя не видны читателям (`front txnid` , грязный список).

5.3 Глава 15. Copy-on-Write и конвейер коммита

5.3.1 15.1. Жизненный цикл страницы при записи

Состояние страницы определяется сравнением её метки `mp->txnid` с номером транзакции:

Состояние	Условие	Смысл
frozen	<code>txnid < txn->txnid</code>	Старая версия, видима читателям; менять нельзя — только копировать
spilled	<code>txnid == txn->txnid</code>	Уже записана на диск текущим писателем
shadowed	<code>txnid > txn->txnid</code>	Есть более новая версия; эта — устаревшая
modifiable	<code>txnid == front txnid</code>	Новая версия текущего писателя; можно менять на месте

Механизм touch (CoW): если страница «заморожена» — выделяем новую (из GC или хвоста файла), копируем содержимое, метим `front txnid` , кладем в грязный список; старую освобождаем. Родитель, указывавший на старую, тоже становится «изменяемым» — CoW распространяется вверх к корню. Курсоры, ссылавшиеся на старую версию, переводятся на новую.

5.3.2 15.2. Грязный список (DPL)

В пределах одной транзакции страница попадает в грязный список **один раз**, сколько бы раз её ни меняли — это ключевой фактор сдерживания WAF (write amplification factor — коэффициент усиления записи; подробно в томе IV, глава 21).

5.3.3 15.3. Спилл

Если объём грязных страниц превышает лимит (`dp_limit` , по умолчанию $\approx 1/42$ ОЗУ), включается **спилл**: часть страниц выгружается на диск заранее. Страницы, на которые ссылаются курсоры, спиллу не подлежат; порядок выбора управляется опциями-знаменателями (минимум/максимум доли).

Нюанс: если спиллнутая страница затем снова изменяется — она будет записана повторно (дополнительная амплификация). Это компромисс «память vs WAF».

5.3.4 15.4. Loose-страницы и refund

- **Loose-страницы** — небольшой кэш освобождённых в текущей транзакции грязных страниц (лимит `loose_limit` , по умолчанию 64) для быстрого переиспользования без обращения к GC.

- **Refund** — возврат освобождённых хвостовых страниц в неразмеченное пространство (а не в GC); снижает WAF и даёт онлайн-сжатие.

5.3.5 15.5. Конвейер коммита

1. Закрыть/проверить курсоры.
2. **GC-обработка**: освобождённые страницы складываются в записи GC-дерева.
3. **Refund**: хвостовые освобождения возвращаются в неразмеченное пространство.
4. **Спилл**: при переполнении грязного списка — выгрузка заранее.
5. **Аудит** (в режимах повышенной проверки): сверка сумм страниц.
6. **Двухфазное обновление меты** + синхронизация по режиму долговечности.
7. Снять блокировку писателя, обновить кэш детента, вычистить мёртвых читателей.

Каждая стадия измерима через `mdbx_txn_commit_ex()` → `MDBX_commit_latency` (`preparation/gc_wallclock/audit/write/sync/ending/whole`).

Фрагмент из `examples/c++/20-commit-latency.c++` — сбор `MDBX_commit_latency` по стадиям через `commit_get_latency()` для серии коммитов и усреднение:

```
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(); // возвращает 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";
```

Полный код: `20-commit-latency.c++`.

Примеры к главе: `examples/c++/20-commit-latency.c++`.

5.3.6 15.6. Резюме главы 15

- CoW: frozen → копия → modifiable; путь к корню переписывается целиком.
- DPL дедуплицирует страницы внутри транзакции — главный рычаг WAF.
- Спилл — компромисс память/WAF; loose и refund снижают запись.
- Конвейер коммита: GC → refund → спилл → аудит → мета → sync → разблокировка.

5.3.7 15.7. Упражнения

1. Почему изменение одного байта в листе переписывает весь путь до корня?
2. Как `mdbx_txn_commit_ex` поможет увидеть, что «застряло» — спилл или sync?

5.3.8 15.8. Чек-лист главы 15

- [] Умею по метке `mp->txnid` определить состояние страницы (frozen/spilled/shadowed/modifiable) и что с ней можно делать.
- [] Объясняю, почему CoW переписывает путь до корня и как курсоры переводятся на новые версии страниц.
- [] Знаю, что страница попадает в грязный список один раз за транзакцию, и почему это сдерживает WAF.
- [] Могу описать, когда включается спилл (`dp_limit` ≈ 1/42 ОЗУ), какие страницы он не трогает и чем плох повторный спилл.

- [] Различаю loose-страницы (быстрое переиспользование, `loose_limit = 64`) и refund (возврат хвостовых страниц в неразмеченное пространство).
- [] Воспроизвожу порядок конвейера коммита и знаю, какие его стадии видны в `MDBX_commit_latency`.

5.4 Глава 16. GC — сборщик мусора внутри БД

5.4.1 16.1. Почему нет free-list

У libmdbx нет классического списка свободных страниц. Освобождённые страницы учитываются **персистентно, внутри файла данных**, в специальном **GC-дереве** (отдельная таблица, корень — в мете).

5.4.2 16.2. Формат записи GC

- **Ключ записи** — номер транзакции, освободившей страницы.
- **Значение** — список номеров страниц (сжато, диапазонами).
- Одна запись физически ограничена (~1000 номеров при 4 КБ странице).

Формат GC заморожен и не менялся с v11.3 — важная гарантия совместимости.

5.4.3 16.3. Детент: безопасность переиспользования

Страницы из записи GC с ключом `T` можно переиспользовать только когда $T \leq \text{детент}$ — старейший активный снапшот. Выше детента записи «заморожены» до завершения соответствующего читателя.

Детент вычисляется сканированием RLT (без блокировок, с кэшированием результата).

5.4.4 16.4. FIFO vs LIFO

Политика	Механизм	Эффект
FIFO (по умолчанию)	Берутся страницы из самой старой подходящей записи	Список живёт дольше, остывает на диске
LIFO (<code>MDBX_LIFORECLAIM</code>)	Самые свежие освобождения первыми	Минимально короткий круг циркуляции; страницы ещё «тёплые»; на системах с write-back кэшем — рост производительности записи в разы

Нюанс: `MDBX_LIFORECLAIM` почти не даёт эффекта с `SAFE_NOSYNC` / `UTTERLY_NOSYNC` — там длину цикла определяет частота `env_sync()`. Для поиска плотных последовательностей используются векторные (SIMD) ядра: SSE2/AVX2/AVX512/NEON.

5.4.5 16.5. BigFoot

Одна запись GC вмещает ~1000 номеров страниц (~4 МБ при 4 КБ). Если транзакция освобождает больше (например, заменяя огромное значение), записи оформляются **цепочкой на последовательных txnid** — режим BigFoot. Читатели обязаны быть новее всей цепочки. Цена — больше записей, выше GC-дерево.

BigFoot включён по умолчанию на 64-битных сборках (`MDBX_ENABLE_BIGFOOT`); полностью совместим с форматом базы.

5.4.6 16.6. Рекурсивность обновления GC

GC — тоже CoW-дерево: изменение GC требует выделения страниц, что влияет на список страниц, попадающий в GC. Поэтому перед обновлением формируется оперативный запас свободных страниц: нехватка → нерациональный рост БД, избыток → оверхед.

5.4.7 16.7. rp_augment_limit и gc_time_limit

Поиск плотных последовательностей (для больших значений) может быть дорогим при фрагментации. Два ограничителя стоимости:

- `rp_augment_limit` — предел накопления списков при поиске; превышение → дешевле дописать новые страницы в хвост файла;
- `gc_time_limit` — тайм-лимит (в 1/65536 секунды) на поиск в течение пишущей транзакции.

Правило: уменьшайте `rp_augment_limit` только вместе с `gc_time_limit`; слишком малый `rp_augment` не лечит рост GC, а усугубляет его ради скорости вставки длинных записей.

Фрагмент из `examples/c++/22-gc-limits.c++` — установка и чтение лимитов GC: `rp_augment_limit` (запас страниц для рекламации) и `gc_time_limit` (1/65536 секунды на поиск последовательностей):

```
env.set_extra_option(opt::rp_augment_limit, 128 * 1024);
// gc_time_limit задаётся в 1/65536 долях секунды (16.16 fixed point):
// 2500 ≈ 38 мс на поиск последовательностей страниц в GC внутри транзакции.
env.set_extra_option(opt::gc_time_limit, 2500);

// Нагрузка с длинными значениями: заставляет GC работать активнее.
{
    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";
```

Полный код: [22-gc-limits.c++](#).

5.4.8 16.8. Ранняя очистка GC (2025) и не-отложенная очистка (devel)

- **Ранняя очистка (2025):** переработанные записи GC начинают обрабатываться раньше.
- **Не-отложенная очистка (devel, 0.14.x):** переработанные записи GC удаляются сразу после чтения, а не при фиксации; накладные расходы становятся пропорциональны объёму операций. Это же открыло путь к явной дефрагментации без копирования (0.14.2+).

Примеры к главе: [examples/c++/21-gc-observe.c++](#); [examples/c++/22-gc-limits.c++](#).

5.4.9 16.9. Резюме главы 16

- GC — дерево записей «txnid → список страниц» внутри файла (не free-list).
- Переиспользование только новее детента.
- FIFO по умолчанию; `MDBX_LIFORECLAIM` — LIFO.
- BigFoot — цепочки для больших освобождений.
- `rp_augment_limit` / `gc_time_limit` — контроль стоимости поиска.

5.4.10 16.10. Упражнения

1. Объясните, почему долгий читатель «замораживает» переработку, используя понятие детента.
2. Когда LIFO даст выигрыш, а когда нет?

5.4.11 16.11. Чек-лист главы 16

- [] Могу объяснить, почему в libmdbx нет free-list и где персистентно учитываются освобождённые страницы.
- [] Описываю формат записи GC: ключ — txnid освободителя, значение — список страниц диапазонами; формат заморожен с v11.3.
- [] Объясняю условие переиспользования через детент и что происходит с записями GC выше детента.
- [] Знаю разницу FIFO и LIFO и когда MDBX_LIFORECLAIM даёт выигрыш, а когда нет.
- [] Понимаю механизм BigFoot: цепочки записей на последовательных txnid и требование к читателям быть новее всей цепочки.
- [] Могу объяснить, почему обновление GC рекурсивно (GC — тоже CoW-дерево) и зачем нужен оперативный запас страниц.
- [] Знаю назначение rp_augment_limit и gc_time_limit и правило их совместного уменьшения.

5.5 Глава 17. Рост, сжатие и дефрагментация БД

5.5.1 17.1. Почему БД растёт

- **Долгие читатели** замораживают детент → GC не переиспользует страницы → писатель берёт новые из хвоста файла. Даже цикл put→del «распухает».
- **MDBX_SAFE_NOSYNC** — постоянный квази-долгий читатель (последний steady-коммит).
- **Фрагментация** и нехватка контигуальных последовательностей для больших значений.
- **Забывшие читатели** (крэш потока, неправильный TLS).

5.5.2 17.2. Геометрия: growth_step, upper, shrink_threshold

Файл растёт автоматически, кратными `growth_step`, до `upper`. Усечение возможно только до последней используемой страницы; одна занятая страница у конца может блокировать сжатие неопределённо долго. Сжатие требует гистерезиса (шаг сжатия > шага роста), иначе файл «дышит».

5.5.3 17.3. MDBX_MAP_FULL и разрешающие механизмы

Когда GC пуст/заморожен и файл упёрся в `upper`, срабатывают, по возможности: новый steady-point (сдвиг детента) → HSR-колбэк (HSR — Handle-Slow-Readers, механизм вытеснения застрявших читателей; подробно в томе V, гл. 29) → выселение припаркованных читателей → и лишь затем `MDBX_MAP_FULL`.

Фрагмент из `examples/c++/23-map-full.c++` — воспроизведение `MDBX_MAP_FULL` (в примере жёсткая геометрия ограничивает файл 512 КБ) и разрешение через увеличение `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";
}

// Увеличиваем верхнюю границу — и продолжаем с того же места.
mdbx::env::geometry bigger;
bigger.size_upper = 8 * 1024 * 1024;
try {
    env.set_geometry(bigger);
} catch (const mdbx::db_unable_extend &) {
    // На 32-битной Windows адресный диапазон сразу за отображением может
    // оказаться занят: переемываем ту же БД с увеличенной геометрией.
    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());
}

```

Увеличение `upper` на живой БД требует переотображения: на большинстве платформ оно выполняется на месте, но на 32-битной Windows рядом с текущим отображением может не оказаться свободного диапазона адресов — тогда возникает `MDBX_UNABLE_EXTEND_MAPSIZE`, а переносимый путь — закрыть и переоткрыть БД с увеличенной геометрией (новое отображение создаётся сразу с нужным `upper`).

Полный код: [23-map-full.c++](#).

5.5.4 17.4. Авто-компактификация (implicit shrink)

При каждом коммите освобождённые хвостовые страницы возвращаются в неразмеченное пространство (refund). Когда свободный хвост превышает `shrink_threshold` (+ небольшой запас), коммит дополнительно: сбрасывает «лишний» диапазон через `madvise(MADV_DONTNEED/REMOVE)` и усекает файл.

Ограничения: не работает, пока долгие читатели удерживают хвост или файл используют несколько процессов; при `SAFE_NOSYNC` ограничено частотой steady-point'ов.

5.5.5 17.5. Явная дефрагментация

`mdbx_env_defrag()` / утилита `mdbx_defrag` (0.14.2+) — аккуратная перестановка страниц к началу файла циклами; каждый цикл — отдельная коммитнутая транзакция. Параметры: `defrag_atleast/defrag_enough` (минимальное/достаточное сокращение), `time_atleast/time_limit` (1/65536 c), `acceptable_backlash`, `preferred_batch`. Код результата — `MDBX_defrag_enough_threshold`.

Гарантированное уменьшение файла — также `mdbx_copy -c` (компактифицированная копия) или `dump+load (-a)`.

5.5.6 17.6. Оценка ёмкости

Деление размера файла на размер элемента **переоценивает** реальную ёмкость: страничная гранулярность, путь к корню, GC и гистерезис добавляют накладные. Оценивайте по худшему случаю заморозки переиспользования.

Примеры к главе: [examples/c++/23-map-full.c++](#); [examples/c++/24-defrag.c++](#).

5.5.7 17.7. Резюме главы 17

- Рост = не переиспользуемые страницы (читатели/SAFE_NOSYNC/фрагментация).
- `upper` — жёсткий предел; MAP_FULL разрешается steady/HSR/выселением.
- Implicit shrink — бесплатная компактификация в составе коммитов.
- Явная дефрагментация циклами; гарантированное сжатие — `copy -c / dump+load`.

5.5.8 17.8. Упражнения

1. Почему «в базе 10 ГБ, значит 10 ГБ данных» — ошибка оценки?
2. Опишите сценарий, где авто-компактификация не работает.

5.5.9 17.9. Чек-лист главы 17

- [] Называю причины роста БД: долгие читатели, `MDBX_SAFE_NOSYNC`, фрагментация, забытые читатели.
- [] Понимаю, почему усечение возможно только до последней используемой страницы и зачем нужен гистерезис шага сжатия.
- [] Воспроизвожу цепочку разрешения `MDBX_MAP_FULL`: steady-point → HSR-колбэк → выселение припаркованных читателей → ошибка.
- [] Знаю ограничения авто-компактификации (implicit shrink) и когда она не работает.
- [] Могу описать явную дефрагментацию циклами и гарантированные пути уменьшения файла (`mdbx_copy -c`, `dump+load`).

- [] Объясняя, почему деление размера файла на размер элемента переоценивает реальную ёмкость.

5.6 Глава 18. Вложенные транзакции

5.6.1 18.1. Зачем нужны

Вложенная транзакция — дочерняя write-транзакция внутри родительской, позволяющая группировать изменения с возможностью **частичного отката**.

5.6.2 18.2. Реализация

Дочерняя транзакция работает на том же номере транзакции и front txnid. Она наследует у родителя: пространство страниц и грязный список (с тенью родительского), retired-списки (списки уже освобождённых, но ещё удерживаемых от переиспользования страниц — см. 19.4), буферы GC, дескрипторы таблиц и курсоры (через механизм «спаренных» курсоров).

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

- **Коммит:** грязные страницы дочерней сливаются в родительский список; счётчики GC переносятся. Если дочерняя «чистая» — быстрый путь без слияния.
- **Abort:** все изменения отбрасываются; утилизированные страницы «рас-утилизируются»; хендлы и курсоры восстанавливаются.

Фрагмент из `examples/c++/25-nested-txn.c++` — вложенная транзакция: `start_nested()` + `abort()` откатывает её изменения (undo); симметричный `commit()` вливает их в родителя (join):

```
// Вложенная транзакция с откатом (undo).
{
    auto nested = txn.start_nested();
    auto ntable = nested.open_map(nullptr);
    nested.insert(ntable, mdbx::slice("k2"), mdbx::slice("v2"));
    nested.abort(); // откат вложенной
}
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";
}
```

Полный код: [25-nested-txn.c++](#).

5.6.4 18.4. Судьба таблиц

- Таблица, **созданная** во вложенной: при commit остаётся, при abort исчезает.
- Таблица, **удалённая** во вложенной: при commit удаляется, при abort «оживает».

5.6.5 18.5. Анти-паттерн: вложенная транзакция в каждой функции

Вызывать `mdbx_txn_begin(parent, ...)` в каждой маленькой функции — анти-паттерн: теряется смысл группировки, растут накладные. Вкладывайте осознанно, по границам логических операций.

5.6.6 18.6. Workaround нехватки места

Вложенные транзакции — известный workaround для сценариев нехватки места (можно частично откатиться, не теряя предыдущие изменения).

Нюанс: вложенные транзакции не сочетаются с `MDBX_WRITEMAP`.

Примеры к главе: [examples/c++/25-nested-txn.c++](#).

5.6.7 18.7. Резюме главы 18

- Вложенная транзакция = дочерняя write-транзакция с частичным откатом.
- Commit — слияние в родителя; Abort — полный откат.
- Судьба созданных/удалённых таблиц следует правилам CoW.
- Не вкладывайте «на каждую функцию»; не сочетайте с WRITEMAP.

5.6.8 18.8. Упражнения

1. Напишите пример «группа обновлений с откатом последнего шага» через вложенную транзакцию.
2. Проверьте судьбу таблицы, созданной и удалённой во вложенной транзакции при abort.

5.6.9 18.9. Чек-лист главы 18

- [] Объясняю, что дочерняя транзакция работает на том же номере транзакции, и что она наследует у родителя.
- [] Различаю commit (join): слияние грязных страниц в родителя, и abort (undo): полный откат и «рас-утилизация» страниц.
- [] Знаю судьбу таблиц, созданных и удалённых во вложенной транзакции, при commit и при abort.
- [] Умею показать частичный откат через `start_nested() + abort()`.
- [] Могу обосновать, почему вложенная транзакция «в каждой функции» — анти-паттерн.
- [] Знаю, что вложенные транзакции — workaround нехватки места, и что они не сочетаются с `MDBX_WRITEMAP`.

5.7 Глава 19. Файл блокировок и межпроцессная синхронизация

5.7.1 19.1. Зачем отдельный файл

Рядом с файлом данных живёт файл блокировок (`.lck`): глобальный мьютекс писателя, таблица читателей (RLT), статистика, кэши старейшего снапшота. Он нужен для координации **между процессами**.

Фрагмент из `examples/c++/26-two-processes.c++` — межпроцессный доступ: процесс-писатель коммитит данные, процесс-читатель (потомок через `fork()`) открывает ту же БД и читает их:

```
// Писатель: родительский процесс коммитит данные и закрывает окружение.
{
    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) {
    // Читатель: потомок открывает ту же БД и читает данные.
    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);
}
```

Полный код: [26-two-processes.c++](#).

5.7.2 19.2. Реализации блокировок

Опция сборки `MDBX_LOCKING` :

Flavour	Примитивы	Сигнатура LCK
POSIX2008	Надёжные мьютексы (Linux по умолчанию)	0x8017
POSIX2001	Общие мьютексы	0x8017
SYSV	Семафоры (macOS по умолчанию)	0xF18D
WIN32FILES	Windows (LockFileEx)	0xF10C
POSIX1988	(legacy)	0xFC29

Нюанс: почему у **POSIX2001** и **POSIX2008** одна сигнатура — это не опечатка. Оба флавора хранят в LCK межпроцессный примитив одного и того же типа — `pthread_mutex_t`, поэтому их on-disk формат совпадает байт-в-байт, и сигнатура у них общая. Различие — только в используемых возможностях API: надёжные (robust) мьютексы появились в POSIX.1-2008, режим 2001 работает с обычными process-shared мьютексами. Следствие: базу, открытую в одном из этих двух режимов, можно продолжить в другом — а с SYSV, WIN32FILES и POSIX1988 нельзя (у них другие сигнатуры, и LCK такого формата будет отвергнут).

При этом фактическая проверка совместимости при открытии использует не сырую сигнатуру, а производный хеш `MDBX_LOCK_FORMAT` — он смешивает сигнатуру флавора с `sizeof(reader_slot_t)` и офсетами ключевых полей структуры `lck_t`, так что приведённые константы — лишь «база», уникальная для формата примитива, а полный паспорт учитывает и раскладку остальных полей.

Формат LCK версионирован **отдельно** от формата данных: изменение структуры ломает только совместный доступ к уже открытой базе, не сам файл данных.

5.7.3 19.3. Почему LockFileEx на Windows, а не именованные мьютексы

Файловые блокировки (LockFileEx) выбраны осознанно: работают на сетевых дисках и защищают от некомпетентных действий. Цена — в наивных бенчмарках с множеством мелких транзакций libmdbx может отставать от LMDb (захват/освобождение файловой блокировки — сотни микросекунд).

5.7.4 19.4. Слот читателя (RLT)

txnid (atomic)	номер снимота (INVALID_TXNID = свободен)
tid	владелец; PARKED = UINT64_MAX, OUSTED = UINT64_MAX-1
pid	процесс-владелец
snapshot_pages_used	first_unallocated на момент снимота
snapshot_pages_retired	сколько страниц удерживает от переиспользования

Живость слотов проверяется средствами ОС (защита от повторного использования pid/tid). stale-слоты не сканируются поштучно — таблица целиком реинициализируется, когда LCK открывает единственный процесс.

5.7.5 19.5. Восстановление блокировок после сбоя процесса

После краха процесса ОС освобождает его блокировки; мёртвые слоты читателей очищаются механизмами живости и `mdbx_reader_check()`.

5.7.6 19.6. Режим без файла блокировок

Возможен режим «один процесс, эксклюзивно» без `.lck` (опция сборки) — когда межпроцессный доступ не нужен.

Примеры к главе: [examples/c++/26-two-processes.c++](#).

5.7.7 19.7. Резюме главы 19

- LCK: мьютекс писателя + RLT + статистика; отдельно версионизируется.
- Реализации: POSIX/SysV/WIN32FILES; сигнатуры flavour'ов.
- LockFileEx — выбор ради сетевых дисков; медленнее для мелких транзакций.
- Слот RLT: txnid/tid/pid + пининг-поля; живость через ОС.
- Возможен режим без LCK (один процесс).

5.7.8 19.8. Упражнения

1. Почему удалять `.lck` при открытой базе — ошибка?
2. Чем `mdbx_reader_check` отличается от «автоматической» очистки?

5.7.9 19.9. Чек-лист главы 19

- [] Могу перечислить, что живёт в файле блокировок, и зачем он нужен при межпроцессном доступе.
- [] Знаю, что формат LCK версионизируется отдельно от формата данных, и что это значит для совместимости.
- [] Объясняю, почему на Windows выбран `LockFileEx`, и чем за это платим в бенчмарках с мелкими транзакциями.
- [] Описываю слот читателя RLT и поля, фиксирующие, сколько страниц читатель удерживает от переиспользования.
- [] Понимаю, как проверяется живость слотов средствами ОС и что очищает мёртвые слоты после краха процесса (`mdbx_reader_check()`).
- [] Знаю, когда возможен режим без файла блокировок (один процесс, эксклюзивно).

5.8 Глава 20. Долговечность и восстановление

5.8.1 20.1. Двухфазное обновление меты: weak vs steady

- **weak** — данные в файле, но не гарантированно сброшены на постоянный носитель;
- **steady** — данные сброшены на диск; снимок переживает сбой системы.

5.8.2 20.2. Восстановление без WAL

WAL отсутствует намеренно. После сбоя при открытии выбирается последняя **цельная и валидная** мета (согласованная тройка, совпадение половин txnid, контрольная сумма). «Полузаписанные» транзакции, не успевшие опубликовать мету, просто не существуют.

`mdbx_txn_checkpoint(txn, weakening_durability, &latency)` — **вариант коммита**: фиксирует транзакцию и возвращает латенсы по стадиям (`MDBX_commit_latency`), позволяя ослабить долговечность через `weakening_durability`. В заголовке помечено как «может измениться в будущих релизах» — используйте для диагностики, а не как стабильный API. Принудительное продвижение steady-точки выполняется `mdbx_env_sync_ex(env, force=true, ...)` (Том II, глава 10), а не checkpoint'ом.

5.8.3 20.3. Open for recovery

`mdbx_env_open_for_recovery()` открывает базу с выбором целевой мета-страницы (`target_meta`) под эксклюзивной блокировкой, чтобы разрешить «застывшую» тройку. Recovery-проверки с 0.12.7+ **не изменяют базу**. Код `MDBX_WANNA_RECOVERY` возвращается при read-only открытии базы, требующей восстановления.

Внимание: в заголовке функция помечена как внутренний API утилиты `mdbx_chk`, «subject to change at any time» — в прикладном коде она не рекомендуется; штатный путь — read-write открытие или `mdbx_chk`.

Фрагмент из `examples/c++/27-recovery.c++` — падение писателя (`_exit` без `commit/abort`) откатывается при повторном открытии (`steady`), а `open_for_recovery()` открывает конкретную meta-страницу:

```
// Повторное открытие: незакоммиченные изменения откатились.
{
    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();
}

// Режим восстановления: открыть конкретную meta-страницу (0..2).
{
    auto recovery = mdbx::env_managed::open_for_recovery(path.c_str(), 0 /* target_meta */, true);
    std::cout << "open_for_recovery: ok\n";
}
```

Полный код: [27-recovery.c++](#).

5.8.4 20.4. boot_id: семантика и поведение в LXC

`boot_id` — идентификатор загрузки ОС, учитываемый при откате слабых мет. В LXC `boot_id` может отсутствовать/быть общим для контейнеров — контроль отката слабых мет учитывает это. Это важно при сбоях внутри контейнеров.

5.8.5 20.5. Некогерентность unified page cache

Известная проблема [issue #269](#): некогерентность объединённого page cache между процессами. Защита — `MDBX_FORCE_CHECK_MMAP_COHERENCY` (опция сборки, дефолт 0); внесена в сериях 0.11.5–0.11.6; счётчик срабатываний виден в диагностике.

5.8.6 20.6. mdbx_chk: проверка целостности

`mdbx_chk` — глубокая валидация: меты/тройка, деревья, упорядоченность, инвариант достижимости, GC. Режимы: `-w` (read-write, откат к `steady`), `-d` (постраничный обход — без него нельзя найти lost/double-used страницы), `-i` (игнор ложных ошибок порядка при кастомных компараторах), `-0/-1/-2` (конкретная мета), `-vvvvv` (гистограммы). Exit-код: 0 = чисто.

5.8.7 20.7. Что переживёт сбой питания — по режимам

Режим	После сбоя
<code>MDBX_SYNC_DURABLE</code>	Все закоммиченные данные на месте
<code>MDBX_NOMETASYNC</code>	Потеря последних коммитов (мета отложено)
<code>MDBX_SAFE_NOSYNC</code>	Откат к последнему <code>steady</code>
<code>MDBX_UTTERLY_NOSYNC</code>	Никаких гарантий; возможна порча

Примеры к главе: [examples/c++/27-recovery.c++](#).

5.8.8 20.8. Резюме главы 20

- Двухфазная мета: `weak/steady`; восстановление — выбор последней цельной меты.
- `Open-for-recovery` с `target_meta`; с 0.12.7 проверки не изменяют базу.
- `boot_id` важен в LXC; page-cache несоогласованность — [#269](#) + `FORCE_CHECK`.
- `mdbx_chk` — основной инструмент проверки; режимы см. выше.

5.8.9 20.9. Упражнения

1. Опишите, что произойдёт с базой после `kill -9` посреди коммита, в режиме `DURABLE`.
2. Зачем `mdbx_chk -d` нужен для поиска `lost-unused` страниц?

5.8.10 20.10. Чек-лист главы 20

- [] Различаю `weak` и `steady` снапшоты и знаю, какой из них переживает сбой системы.
 - [] Объясняю, как работает восстановление без WAL: выбор последней цельной и валидной меты; «полузаписанные» транзакции просто не существуют.
 - [] Знаю назначение `mdbx_env_open_for_recovery()` с `target_meta` и почему в прикладном коде она не рекомендуется.
 - [] Понимаю роль `boot_id` при откате слабых мет и его особенности в LXC.
 - [] Знаю о некогерентности `unified page cache` (#269) и защите `MDBX_FORCE_CHECK_MMAP_COHERENCY`.
 - [] Умею выбрать режимы `mdbx_chk` (`-w`, `-d`, `-i`, `-0/-1/-2`) под задачу и понимаю, зачем нужен `-d`.
 - [] Могу для каждого режима долговечности сказать, что переживёт сбой питания.
-

5.9 Итог тома

Вы понимаете внутреннее устройство `libmdbx`: от страницы и B+tree до меты, GC и восстановления. Теперь вы можете объяснять любое наблюдаемое поведение через архитектуру.

Что дальше: Том IV — производительность и оптимизация: WAF, выбор конфигурации под сценарий, микрооптимизации, `get-cached`, массовые операции, бенчмарки.

6. Том IV. Производительность и оптимизация

Уровень: для опытных разработчиков. **Цель тома:** вы умеете настраивать libmdbx под конкретный сценарий, понимаете источники write amplification и узкие места, умеете измерять и интерпретировать метрики.

6.1 Глава 21. WAF — write amplification factor

6.1.1 21.1. Что такое WAF и почему он важен

WAF (Write Amplification Factor) — отношение реально записанных на диск байт к объёму данных, принятых от приложения. WAF = 100 значит: на каждый записанный вами байт диск получает 100 байт. Высокий WAF означает износ SSD, меньшее время жизни диска и худшую пропускную способность.

6.1.2 21.2. Источники амплификации

1. **Страничная гранулярность.** CoW не модифицирует старую страницу, а создаёт новую. Изменение одного байта в листе = запись целой страницы (например, 4 КБ).
2. **Путь к корню.** Новая версия листа меняет указатель у родителя → родитель тоже становится новым → и так до корня. Точечное обновление = $(\text{высота}+1) \times \text{pagesize}$ байт записи.
3. **GC-записи и мета.** Освобождённые страницы описываются в GC-дереве; двухфазная мета — ещё 1–3 страницы на коммит.
4. **Спилл.** Спиллутая страница, затем снова изменённая, записывается повторно.
5. **Merge/rebalance.** Слияния полупустых страниц могут переписывать соседние.

6.1.3 21.3. Батчинг — главный рычаг

В пределах одной транзакции страница попадает в грязный список **один раз**. N операций на M уникальных страницах записывают $M \times \text{pagesize}$ байт (+GC+мета). Поэтому:

- одна операция на транзакцию: $\text{WAF} = \text{высота}+1 \approx 4\text{--}5$ (и выше);
- 10 000 операций в одной транзакции: WAF приближается к «1 страница на множество изменений».

Вывод: батчинг (100–10 000 операций на коммит) — самый мощный рычаг снижения WAF.

6.1.4 21.4. Влияние высоты дерева

Высота $B+\text{tree} \sim \log_B(N)$, где B — среднее число потомков в узле (сотни). Увеличение объёма данных в 2^k раз добавляет $\sim k$ уровней. Для point-update удельный WAF растёт **логарифмически** с объёмом — медленно.

6.1.5 21.5. Спилл как компромисс

Когда грязный список переполняет `dp_limit` ($\approx 1/42$ ОЗУ), часть страниц выгружается заранее. Если такая страница снова изменяется — она пишется повторно (лишний WAF). Это компромисс «память vs WAF».

6.1.6 21.6. `prefer_waf_insteadof_balance` и `merge_threshold`

Опция `prefer_waf_insteadof_balance` намеренно жертвует равномерностью заполнения страниц ради меньшего числа записываемых страниц (использовать уже изменённую страницу-соседа); включена по умолчанию. Порог слияния — `MDBX_opt_merge_threshold` (16.16 %; дефолт 33%, допустимый диапазон [12.5%..50%]). Оба — прямые рычаги WAF.

6.1.7 21.7. Расчёт WAF для типичных сценариев

Для point-update в дереве высоты h на странице 4 КБ: запись $\approx (h+1) \times 4096$ байт на операцию. При $h=4$ и значении 32 байта: 20 КБ / 32 Б $\approx$ **WAF $\approx$ 640** (одиночные операции!). При батчинге 10 000 записей на те же 5 000 уникальных страниц: $5\,000 \times 4096 / (10\,000 \times 32) \approx$ **WAF $\approx$ 64**. С дальнейшим ростом батча WAF продолжает падать.

Внимание: числа без контекста бесполезны — всегда указывайте режим синхронизации, размер страницы, высоту дерева и размер транзакции.

Фрагмент из `examples/c++/28-waf-batching.c++` — замер записанных страниц (`newly+cow` из `mi_pgop_stat`, те же счётчики, что у `mdbx_stat` -p) для разных размеров батча записи:

```
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";
```

Полный код: [28-waf-batching.c++](#).

Примеры к главе: [examples/c++/28-waf-batching.c++](#).

6.1.8 21.8. Резюме главы 21

- WAF = записано на диск / принято от приложения.
- Источники: гранулярность страницы, путь к корню, GC/мета, спилл, merge.
- Батчинг — главный рычаг: страница пишется раз за транзакцию.
- Рост с объёмом — логарифмический; большие значения — линейный вклад.

6.1.9 21.9. Упражнения

1. Посчитайте WAF для point-update при высоте 3 и странице 4 КБ.
2. Как повлияет батч из 1 000 обновлений на те же 500 страниц?

6.1.10 21.10. Чек-лист главы 21

- [] Могу посчитать запись для point-update как $(\text{высота}+1) \times \text{pagesize}$ байт и получить из неё WAF.
- [] Называю все источники амплификации: гранулярность страницы, путь к корню, GC/мета, спилл, merge/rebalance.
- [] Объясняю, почему страница попадает в грязный список один раз за транзакцию и почему батчинг — главный рычаг снижения WAF.
- [] Знаю, что удельный WAF растёт с объёмом логарифмически, и могу объяснить почему ($\text{высота} \sim \log_B(N)$).
- [] Понимаю компромисс спилла: `dp_limit` $\approx 1/42$ ОЗУ, повторная запись спиллнутой страницы.
- [] Знаю эффект `prefer_waf_insteadof_balance` и `MDBX_opt_merge_threshold`.
- [] Привожу числа WAF только с контекстом: режим синхронизации, размер страницы, высота дерева, размер транзакции.

6.2 Глава 22. Выбор конфигурации под сценарий

6.2.1 22.1. Сценарий 1: Высокая пропускная способность записи (Ethereum-подобный)

- Режим: `MDBX_SAFE_NOSYNC` + авто-sync (`syncbytes / syncperiod`), или `NOMETASYNC`.
- Геометрия: `upper` с запасом; `growth_step` крупный (реже `fsync`).
- Страница: 4–8 КБ.
- Батчинг: десятки–тысячи операций на коммит.
- Ожидаемые числа: сотни тысяч–миллионы вставок/с (зависит от железа и размера ключей).

6.2.2 22.2. Сценарий 2: Чтение-heavy с редкой записью (кэш, индекс)

- Режим: по умолчанию (`DURABLE`); запись редкая, надёжность важна.
- Страница: по размеру значений; короткие ключи.
- `maxreaders` — по числу потоков-читателей.
- Чтение линейно масштабируется по ядрам (wait-free).

6.2.3 22.3. Сценарий 3: Встраиваемое мобильное приложение (Isar-подобный)

- Маленькая база; страница 4 КБ.
- `MDBX_EXCLUSIVE` при необходимости монопольного доступа.
- Аккуратные read-only транзакции; парковка для долгих чтений.
- Низкий WAF (малый churn) — продлевает жизнь flash.

6.2.4 22.4. Сценарий 4: Аналитика, обход всей БД

- Чтение full-scan; важно: страница 64 КБ для последовательного чтения? (Нет — читается всё; важна локальность и отсутствие долгих писателей.)
- Курсоры + `get_batch`; оценка диапазонов через `estimate_*`.

6.2.5 22.5. Сценарий 5: Многопроцессное чтение (веб-сервер + воркеры)

- Несколько процессов открывают одну базу read-only; один — write.
- `maxreaders` учитывает все процессы; LCK-файл общий.
- Аккуратно с `boot_id` в контейнерах; page-cache когерентность (#269).

6.2.6 22.6. Сценарий 6: Низкая задержка, маленькие транзакции

- Каждая транзакция — одна операция (минимальная задержка).
- Режим: `NOMETASYNC` / `SAFE_NOSYNC`, чтобы не платить `fsync` на каждую.
- На Windows помнить про LockFileEx (мелкие транзакции дороги).
- Ориентир накладных: ~0.5 мкс на транзакцию сверх I/O.

Фрагмент из `examples/c++/29-scenario-configs.c++` — типовые конфигурации как функции, возвращающие `operate_parameters`: флаги и опции под каждый сценарий главы:

```
// Запись-heavy: ленивая синхронизация, достаточно именованных таблиц.
params write_heavy() { return params().lazy_weak_tail().set_max_maps(64); }

// Чтение-heavy: максимальная долговечность, много читательских слотов.
params read_heavy() { return params().robust_synchronous().set_max_readers(1024).set_max_maps(64); }

// Мобильное устройство: малый размер файла, бережное отношение к флеш-памяти.
params mobile() { return params().lazy_weak_tail().set_max_maps(64); }
```

```
// Аналитика: пакетная запись, не мелочиться с балансировкой.
params analytics() { return params().lazy_weak_tail().set_max_maps(64); }

// Многопроцессное чтение: много читательских слотов, стабильная геометрия.
params multi_process_read() { return params().set_max_readers(4096).set_max_maps(64); }

// Низкая задержка: NOMETASYNC (данные на диске, метаданные могут отставать).
params low_latency() { return params().half_synchronous_weak_last().set_max_maps(64); }
```

Полный код: [29-scenario-configs.c++](#).

Примеры к главе: [examples/c++/29-scenario-configs.c++](#).

6.2.7 22.7. Резюме главы 22

- Режим синхронизации выбирается по требованиям к durability.
- Геометрия задаётся с запасом; `upper` не занижать.
- Батчинг масштабируется под сценарий (latency vs throughput).
- Платформенные особенности (Windows LockFileEx, LXC `boot_id`) влияют.

6.2.8 22.8. Упражнения

1. Подберите конфигурацию для сценария «аналитика» и обоснуйте выбор страницы.
2. Что изменится для «низкой задержки» на Windows по сравнению с Linux?

6.2.9 22.9. Чек-лист главы 22

- [] Для каждого из шести сценариев могу обосновать выбор режима синхронизации (от `DURABLE` до `SAFE_NOSYNC` / `NOMETASYNC`).
- [] Знаю, что геометрия задаётся с запасом и почему `upper` не занижают.
- [] Умею подобрать размер батча под баланс latency/throughput конкретного сценария.
- [] Понимаю, когда важен размер `maxreaders` (чтение-heavy, многопроцессное чтение).
- [] Помню платформенные ловушки: LockFileEx на Windows, `boot_id` в контейнерах, когерентность page cache (#269).
- [] Могу объяснить, почему для full-scan переход на большую страницу не панацея.

6.3 Глава 23. Микрооптимизации

6.3.1 23.1. SIMD-поиск свободных страниц

Поиск плотных последовательностей в GC использует векторные ядра: SSE2/AVX2/AVX512/NEON (ускорение $\times 4/\times 8/\times 16$; выбор на этапе выполнения). Это внутренняя оптимизация — вы лишь влияете на неё через `rp_augment_limit` / `gc_time_limit`.

6.3.2 23.2. Branchless бинарный поиск (CMOV)

Бинарный поиск по разделителям branch-страниц использует безусловные переходы (CMOV) — нет ветвлений, предсказуемость. Выигрыш заметен в хот-путях поиска.

6.3.3 23.3. Radix-sort и сортировочные сети

Используются во внутренних процедурах (сортировка списков страниц, слияние PNL).

6.3.4 23.4. C11-атомики для слабых моделей памяти

На ARM/AArch64/PPC/MIPS/RISC-V применяются аккуратные атомики (acquire/release, CAS, safe64). Корректность гарантируется сборкой (проверка атомарности).

6.3.5 23.5. Минимизация системных вызовов

Хот-пути избегают syscall'ов; батчинг I/O через bulk iovec; `mdbx_copy` использует массовую запись.

6.3.6 23.6. Prefault-запись и mincore

`MDBX_opt_prefault_write_enable` — упреждающая запись страниц для устранения page-fault'ов и чтений с диска при первом обращении в `MDBX_WRITEMAP`. Резидентность отслеживается через `mincore` / `madvise`. Выигрыш — когда БД > ОЗУ и страницы часто вытесняются.

6.3.7 23.7. Auto-appending split и слияние с грязным соседом

- Auto-appending split — эффективное добавление ключей в конец (append-сценарии).
- Merge с уже грязным соседом — меньше CoW-копий (связано с `prefer_waf_insteadof_balance`).

Фрагмент из `examples/c++/30-append-prefault.c++` — пакетная загрузка отсортированных ключей: `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());
}
```

Полный код: [30-append-prefault.c++](#).

6.3.8 23.8. Когда эти оптимизации важны, а когда — нет

SIMD/CMOV дают проценты-десятки процентов на конкретных путях. Для большинства приложений решают **батчинг и режим синхронизации**, а не микрооптимизации. Не оптимизируйте вслепую — измеряйте.

Примеры к главе: [examples/c++/30-append-prefault.c++](#).

6.3.9 23.9. Резюме главы 23

- Внутренние оптимизации (SIMD, CMOV, radix, атомики) уже встроены.
- Ваши рычаги: `prefault`, `auto-append`, `merge`-политика.
- Микрооптимизации вторичны к батчингу и режиму синхронизации.

6.3.10 23.10. Упражнения

1. Когда включение `prefault`-записи оправдано? Сформулируйте условие (размер БД vs ОЗУ).
2. Почему «не измерять, а оптимизировать» — анти-паттерн?

6.3.11 23.11. Чек-лист главы 23

- [] Знаю, какие микрооптимизации уже встроены (SIMD-ядра, CMOV, radix-sort, C11-атомики), и что на SIMD-поиск влияю только через `rp_augment_limit` / `gc_time_limit`.
- [] Могу сформулировать условие оправданности `prefault`-записи: БД > ОЗУ, частые вытеснения страниц, `MDBX_WRITEMAP`.
- [] Понимаю выигрыш `auto-appending split` для `append`-сценариев и слияния с уже грязным соседом (связь с `prefer_waf_insteadof_balance`).

- [] Знаю, что хот-пути минимизируют syscall'и (bulk iovec, массовая запись в `mdbx_copy`).
- [] Могу объяснить, почему батчинг и режим синхронизации решают больше, чем микрооптимизации, и почему оптимизировать нужно по измерениям.

6.4 Глава 24. Кэш-поиск (get-cached)

6.4.1 24.1. Что такое get-cached

`mdbx_cache_get()` — сервис чтения «повторяющихся» ключей. Для ключа хранится адрес значения в mmap + версионные метки. При повторном чтении вместо полного спуска по B+tree выполняется **ленивый (ранний) выход**: спускаемся только пока встречаем страницы, изменённые после последнего подтверждения.

6.4.2 24.2. Механизм

Запись кэша: `{trunk_txnid, last_confirmed_txnid, offset, length}` (`offset == 0` = «ключа нет»).

Первый вызов — полный поиск + заполнение записи. Последующие:

1. спуск останавливается на первой странице, не изменённой после `last_confirmed_txnid` ;
2. если ничего не менялось — `MDBX_CACHE_HIT` после нескольких лёгких сравнений;
3. если изменения есть — подтверждение актуальности (`MDBX_CACHE_CONFIRMED`) или полный поиск (`MDBX_CACHE_REFRESHED`).

6.4.3 24.3. Статусы

Статус	Значение
HIT / CONFIRMED / REFRESHED	Результат получен
DIRTY	Значение на грязной (незакоммиченной) странице; валидно только в текущей write-транзакции
BEHIND	Читатель старше версии в кэше; кэш обойти
UNABLE	АВА-ситуация; поиск в обход кэша
RACE	Параллельное обновление записи; поиск в обход
ERROR (MDBX_CACHE_ERROR)	Ошибка, не связанная с отсутствием ключа

Фрагмент из `examples/c++/31-get-cached.c++` — отображение всех статусов кэша `get_cached` : HIT / CONFIRMED / REFRESHED , DIRTY , BEHIND , UNABLE , RACE и 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 "?";
    }
}
```

Полный код: [31-get-cached.c++](#).

6.4.4 24.4. Когда даёт ускорение

Кэш эффективен для **повторяющихся** ключей с малым числом изменений (конфигурация, счётчики, маркеры, lookaside-таблицы). Для уникальных ключей выгоды нет (первый поиск всё равно полный). Отношение полного поиска к HIT-пути легко достигает 10^2 – 10^3 .

6.4.5 24.5. Когда не работает

- Уникальные ключи (нет повторов).
- Активные пишущие транзакции дают `DIRTY` (не кэшируются).
- Многопоточность требует `MDBX_NOSTICKYTHREADS` для многопоточного варианта; однопоточный вариант (`mdbx_cache_get_SingleThreaded`) — дешевле.

6.4.6 24.6. История

Запрос многопоточного lockfree get-cached — Gabriel RABHI («Ghost Body Object»). Многопоточный вариант реализован; статус на текущем каноне — уточнять по мере развития API.

Примеры к главе: [examples/c++/31-get-cached.c++](#).

6.4.7 24.7. Резюме главы 24

- get-cached — ленивый кэш с ранним выходом и версионными метками.
- HIT/.../RACE — статусы результата; `DIRTY` в write-транзакциях.
- Эффективен для повторяющихся «горячих» ключей; до 1000× быстрее поиска.
- Многопоточный вариант требует `NOSTICKYTHREADS`.

6.4.8 24.8. Упражнения

1. Оцените: цикл из 1 000 000 чтений одного ключа — сколько сэкономит кэш?
2. Почему `DIRTY` не кэшируется?

6.4.9 24.9. Чек-лист главы 24

- [] Описываю запись кэша `{trunk_txnid, last_confirmed_txnid, offset, length}` и смысл `offset == 0`.
- [] Объясняю механику ленивого выхода: спуск останавливается на первой странице, не изменённой после `last_confirmed_txnid`.
- [] Различаю статусы `HIT / CONFIRMED / REFRESHED / DIRTY / BEHIND / UNABLE / RACE / ERROR` и знаю, когда поиск идёт в обход кэша.
- [] Понимаю, что `DIRTY` валиден только в текущей write-транзакции и не кэшируется.
- [] Умею оценить, когда кэш даст ускорение (повторяющиеся «горячие» ключи), а когда нет (уникальные ключи).
- [] Знаю требование `MDBX_NOSTICKYTHREADS` для многопоточного варианта и более дешёвый однопоточный вариант (`mdbx_cache_get_SingleThreaded`).

6.5 Глава 25. Массовые операции

6.5.1 25.1. bunch_delete: удаление «гроздьями»

`mdbx_cursor_bunch_delete()` с режимом-действием (`MDBX_bunch_action_t`) удаляет текущее значение / всё мультизначение / всё до или после позиции / всё подряд. Вместо перебора элементов движок **вырезает целые страницы и ветки** — стоимость пропорциональна числу страниц, а не элементов.

6.5.2 25.2. get_batch: пакетное чтение

`mdbx_cursor_get_batch()` — симметричное массовое чтение партиями (bulk iovec).

6.5.3 25.3. GET/PUT_MULTIPLE

Работа с мультимнозначными пакетами: `MDBX_GET_MULTIPLE` / `MDBX_PUT_MULTIPLE` для DUPFIXED-таблиц.

6.5.4 25.4. estimate_range/distance/move

Эвристическая оценка без сканирования: `mdbx_estimate_range()`, `mdbx_estimate_distance()`, `mdbx_estimate_move()`. Строится по общим страницам стеков двух позиций. Точность: в худшем случае расхождение до 4× **на каждом уровне** дерева (кроме первого и последнего); на практике — единицы процентов.

Фрагмент из `examples/c++/32-bulk-ops.c++` — массовые операции: `get_batch` (чтение пачками), `estimate` (оценка числа записей) и `bunch_delete` (удаление «гроздьями» с вырезанием страниц):

```
// Пакетное чтение: до 100 пар за вызов.
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";

// Оценка количества записей (приблизительно).
cur.to_first();
auto est = cur.estimate(mdbx::cursor::move_operation::last);
std::cout << "estimate: ~" << est.approximate_quantity << " keys\n";

// Удаление «гроздьями»: всё от текущей позиции до конца.
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";
```

Полный код: [32-bulk-ops.c++](#).

6.5.5 25.5. txn_clone: параллельная обработка на одном снапшоте

`mdbx_txn_clone()` размножает read-only транзакцию: несколько обработчиков на одном снапшоте без повторного сканирования RLT.

6.5.6 25.6. Батчинг транзакций

100–10 000 операций на коммит — типичный диапазон для записи-heavy. Больше операций → ниже WAF и выше пропускная способность.

6.5.7 25.7. Когда «максимально большая транзакция» — анти-паттерн

- Транзакция в миллионы операций может переполнить лимиты dirty-страниц (`TXN_FULL`), породить спилл и всплеск latency.
- Долгая write-транзакция блокирует других писателей и увеличивает окно риска.
- Дробите на разумные батчи (например, по 10 000 операций) — баланс между WAF и задержкой.

Примеры к главе: [examples/c++/32-bulk-ops.c++](#).

6.5.8 25.8. Резюме главы 25

- `bunch_delete/delete_range` — массовое удаление вырезанием страниц.
- `get_batch`, `GET/PUT_MULTIPLE` — пакетное чтение/запись.
- `estimate_*` — дешёвая оценка диапазонов (до 4×/уровень в худшем случае).

- `txn_clone` — параллельная обработка на одном снапшоте.
- Разумный батч лучше «максимально большой транзакции».

6.5.9 25.9. Упражнения

1. Сравните посточное удаление диапазона и `bunch_delete` на 100 000 ключах.
2. Когда `estimate_range` полезен до выполнения запроса?

6.5.10 25.10. Чек-лист главы 25

- [] Объясняю, почему `bunch_delete` дешевле перебора элементов: вырезаются целые страницы и ветки, стоимость пропорциональна числу страниц.
- [] Знаю симметричные массовые чтения/записи: `get_batch`, `MDBX_GET_MULTIPLE` / `MDBX_PUT_MULTIPLE` (DUPFIXED-таблицы).
- [] Понимаю принцип `estimate_*`: оценка по общим страницам стеков двух позиций, в худшем случае до 4× на уровень дерева.
- [] Умею применить `txn_clone` для параллельной обработки на одном снапшоте.
- [] Могу назвать последствия «максимально большой транзакции»: `TXN_FULL`, спилл, всплеск latency, блокировка других писателей.
- [] Выбираю разумный размер батча (например, ~10 000 операций) как баланс между WAF и задержкой.

6.6 Глава 26. Бенчмарки и измерения

6.6.1 26.1. ioarena

`ioarena` — стандартный бенчмарк-конвейер проекта. Запускается на готовых конфигурациях; показывает TPS записи/чтения для разных движков и режимов.

6.6.2 26.2. Что измерять

- **TPS записи** — коммитов/сек в режимах синхронизации.
- **get/s чтения** — операций/сек.
- **WAF** — через счётчики страниц (`mdbx_stat -p`).
- **Задержку коммита** — `mdbx_txn_commit_ex()` → `MDBX_commit_latency` по стадиям (`preparation/gc_wallclock/audit/write/sync/ending/whole`).

6.6.3 26.3. MDBX_ENABLE_PROFGC

Профилирование GC: собирается в LCK-файле, возвращается в `commit_latency.gc_prof`. Ключевые поля: `work_rtime_monotonic` / `work_xtime_cpu` (время GC), `work_rsteps` / `work_xpages` (фрагментация), `work_majflt` (page-faults), `max_reader_lag` / `max_retained_pages` (долгие читатели), `kicks` (HSR — Handle-Slow-Readers, вытеснение застрявших читателей; см. том V, гл. 29).

Фрагмент из `examples/c++/33-profiler.c++` — чтение PROFGC-полей из `commit_latency.gc_prof` (`wloops`, `coalescences`, `flushes`, `kicks`, `max_reader_lag`, `max_retained_pages`); поля заполняются в сборках с `MDBX_ENABLE_PROFGC`:

```
// Нагрузка, порождающая работу GC: вставки и удаления.
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, mdbx::slice(key), mdbx::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, mdbx::slice("k" + std::to_string(round * 1000 + i)));
    del.commit();
}
```

```

auto txn = env.start_write();
auto table = txn.open_map(nullptr);
txn.insert(table, mdbx::slice("last"), mdbx::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";

```

Полный код: [33-profiler.c++](#).

6.6.4 26.4. Типичные ошибки бенчмаркинга

- **Ассерты:** отладочная сборка (`MDBX_CHECKING>0`) замедляет кратно.
- **Некогерентность page cache** (#269) на Linux при нескольких процессах.
- **Методология:** «мелкие транзакции + DURABLE на Windows» — это LockFileEx, а не движок.
- **Сравнение без контекста:** режим, страница, высота дерева, оборудование.

6.6.5 26.5. Почему libmdbx может быть «в 6–7 раз медленнее LMDB»

Обычно означает: включены ассерты, либо некорректный бенчмарк (мелкие транзакции на Windows с LockFileEx), либо измеряется single-ор режим, где libmdbx сознательно платит за более строгие гарантии (двухфазная мета, проверки). При корректной методологии и тех же режимах разрыв исчезает или обращается в пользу libmdbx.

6.6.6 26.6. Ориентиры

- Запись: ~200 TPS одиночных durable-транзакций на обычном SSD; с батчингом и SAFE_NOSYNC — десятки– сотни тысяч и миллионы операций/с.
- Чтение: 1–3 млн get/с (короткие ключи, тёплый кэш); линейное масштабирование по ядрам.
- Вставки: 20K–10M/с в зависимости от режима и железа.

Внимание: всегда указывайте контекст числа — режим сунс, размер страницы, размер транзакции, железо.

6.6.7 26.7. Масштабирование чтения

Чтение wait-free и линейно масштабируется по ядрам — до упора в память (bandwidth/TLB). Узкое место обычно не движок, а кэш/память.

Примеры к главе: [examples/c++/33-profiler.c++](#).

6.6.8 26.8. Резюме главы 26

- ioarena — стандартный бенчмарк; TPS/get/s/WAF/latency.
- commit_latency + PROFGC — диагностика узких мест.
- Ассерты и методология — главные ловушки.
- «6–7× медленнее LMDB» почти всегда ошибка измерения.
- Ориентиры: 200 TPS durable, 1–3M get/s, 20K–10M вставок/с.

6.6.9 26.9. Упражнения

1. Измерьте commit_latency на своей конфигурации и определите доминирующую стадию.
2. Объясните, почему сравнение «одна транзакция на операцию» некорректно для оценки движка.

6.6.10 26.10. Чек-лист главы 26

- [] Знаю, что измерять: TPS записи, get/s чтения, WAF (через счётчики страниц), задержку коммита по стадиям.
 - [] Умею читать `IMDBX_commit_latency` (`preparation/gc_wallclock/audit/write/sync/ending/whole`) и определять доминирующую стадию.
 - [] Понимаю, о чём говорят поля `gc_prof` (`max_reader_lag`, `max_retained_pages`, `kicks`) — о долгих читателях и срабатываниях HSR.
 - [] Избегаю типичных ошибок: замер на сборке с ассертами, «мелкие транзакции + DURABLE на Windows», сравнение без контекста.
 - [] Могу объяснить, почему «в 6–7 раз медленнее LMDB» почти всегда ошибка измерения или плата за более строгие гарантии.
 - [] Знаю ориентиры (~200 TPS durable, 1–3 млн get/s) и всегда указываю контекст числа.
 - [] Объясняю, почему чтение масштабируется линейно по ядрам и где реальное узкое место (память/кэш, а не движок).
-

6.7 Итог тома

Вы умеете измерять и настраивать `libmdbx`: понимаете WAF, выбираете конфигурацию под сценарий, используете микрооптимизации, `get-cached` и массовые операции, читаете профиль GC и `latency`.

Что дальше: Том V — экспертные темы: правила и советы из базы знаний, платформенные нюансы, HSR, диагностика и отладка, миграция с LMDB, паттерны проектирования, roadmap.

7. Том V. Экспертные темы и edge cases

Уровень: для продвинутых пользователей. **Цель тома:** вы знаете тонкости и грабли, умеете отлаживать проблемы, принимать архитектурные решения, мигрировать с LMDB и строить устойчивые паттерны.

7.1 Глава 27. Правила и советы из базы знаний

Полный перечень правил и советов — в базе знаний ([knowledge-base/rules-and-checklist/](#), [knowledge-base/tips/](#)). Здесь — группировка по темам с акцентами критичности.

7.1.1 27.1. Транзакции и потоки (критично)

- **Одна транзакция — один поток.** Передача объекта транзакции другому потоку — UB/ошибка (`MDBX_THREAD_MISMATCH`).
- **Не передавайте транзакцию между потоками** даже с `NOSTICKYTHREADS`, если другой поток может синхронно вызвать `write`-функцию среды — `deadlock`.
- **Не используйте объекты транзакций из двух потоков одновременно.**
- После `commit / abort` указатели на данные в `map` недействительны.

7.1.2 27.2. Жизненный цикл и целостность

- Не открывайте БД дважды в одном процессе; `fork()` — только с `resurrect_after_fork` в наследнике.
- Незакрытая `read`-транзакция «морозит» переработку страниц — закрывайте/паркуйте.
- Никогда не копируйте файл базы «на лету» — используйте `mdbx_copy`.
- `MDBX_UTTERLY_NOSYNC` — только для некритичных данных.

7.1.3 27.3. Пространство и рост

- Задавайте геометрию один раз до `open`; `upper` не занижать.
- Уменьшать `rp_augment_limit` — только вместе с `gc_time_limit`.
- `MDBX_ENABLE_REFUND=0` в `production` не использовать (отладочная опция).
- Длинные значения: увеличивайте страницу (до 64 КБ), дробите записи.
- Значение, не помещающееся в одну страницу, требует прогона **подряд идущих** страниц: при частой перезаписи больших значений следите за фрагментацией (гистограммы `mdbx_chk`, число `overflow`-страниц в `mdbx_env_stat`), при хранении «тяжёлых» данных рассмотрите вынос из БД.
- Если длинное значение нужно хранить в БД — разбивайте его на чанки, помещающиеся в одну страницу (`mdbx_limits_valsize4page_max()`): в обычной таблице — суффикс номера чанка в ключе, в `DUPSORT`-таблице — префикс номера мультизначения и номера чанка. Ручное разбиение требуется только до появления планируемого **поточкового API для длинных значений** («Streaming BLOBs»). Подробности — [Ограничения и подводные камни](#).

7.1.4 27.4. Операции

- Удаления делайте до вставок в той же транзакции; массовые удаления — `bunch_delete / delete_range`.
- Не стройте `std::map` поверх БД ради упорядоченности — выигрыш обычно меньше затрат.
- Для `DUPSORT`-удаления при итерации — два курсора.

7.1.5 27.5. Типичные контрпримеры

- Нарушение «одна транзакция — один поток» → случайные `BAD_RSL0T` /пересечения.
- Открытие БД дважды → гонки и повреждение регистраций.
- Забытый читатель → рост файла до `MDBX_MAP_FULL`.
- Отладочная сборка в проде → кратные замедления.

Фрагмент из `examples/c++/34-rules-counter.c++` — контрпример «забытого» read-снапшота: поток открыл снапшот (`env.start_read()`) и держит его (`abort()` только по внешнему флагу); пока читатель жив, писатель упирается в `MDBX_MAP_FULL` (GC заморожен, файл растёт):

```
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();
```

Полный код: `34-rules-counter.c++`.

Примеры к главе: `examples/c++/34-rules-counter.c++`.

7.1.6 27.6. Резюме главы 27

- Одна транзакция — один поток: передача транзакции между потоками — UB/ошибка (`MDBX_THREAD_MISMATCH`), даже с `NOSTICKYTHREADS` возможен deadlock.
- Незакрывающая read-транзакция «морозит» переработку страниц — файл растёт до `MDBX_MAP_FULL`; закрывайте/паркуйте.
- Не открывайте БД дважды в одном процессе; `fork()` — только с `resurrect_after_fork` в наследнике; копирование — только `mdbx_copy`.
- `MDBX_UTTERLY_NOSYNC` и отладочные опции (`MDBX_ENABLE_REFUND=0`) в production не использовать.
- Геометрию задавайте один раз до `open`, `upper` не занижать; длинные значения — крупнее страница (до 64 КБ), дробите на чанки, помещающиеся в одну страницу, или выносите из БД; значения, не помещающиеся в одну страницу, требуют континуального прогона и чувствительны к фрагментации — до появления потокового API («Streaming BLOBs»).
- Удаления — до вставок; массовые удаления — `bunch_delete` / `delete_range`; DUPSORT-удаление при итерации — два курсора.
- Типичные контрпримеры: нарушение потоковой дисциплины (`BAD_RSL0T`), двойное открытие, забытый читатель, отладочная сборка в проде.

7.1.7 27.7. Чек-лист главы 27

- [] Убедиться, что транзакции не передаются между потоками и не используются из двух потоков одновременно.
- [] Закрывать/парковать read-транзакции сразу после использования.
- [] Копировать файл базы только через `mdbx_copy`.
- [] Задать геометрию до `open` и не занижать `upper`.
- [] Исключить `MDBX_UTTERLY_NOSYNC` и `MDBX_ENABLE_REFUND=0` из production-конфигурации.
- [] Выполнять удаления до вставок; массовые удаления — через `bunch_delete` / `delete_range`.
- [] Для DUPSORT-удаления при итерации использовать два курсора.

7.2 Глава 28. Платформенные нюансы

7.2.1 28.1. Linux

- `boot_id` учитывается при откате слабых мет; в LXC — общий/отсутствует.
- `/dev/shm` (tmpfs): создание с запасом; `ENOSPC` от `fallocate()` можно игнорировать; с 0.13.8 `fallocate` защищает от `SIGBUS`.
- Page cache: некогерентность unified page cache (#269) — `MDBX_FORCE_CHECK_MMAP_COHERENCY`.
- `mincore` / `madvise(MADV_NOHUGEPAGE)` — отказ от THP.
- Linux < 4.x: `mdbx_env_create()` → `MDBX_INCOMPATIBLE`.

7.2.2 28.2. Windows

- `LockFileEx` медленнее именованных мьютексов; мелкие транзакции дороги.
- 32-бит: лимиты адресного пространства; `/LARGEADDRESSAWARE` (+1 ГБ к `MAX_MAPSIZE32`).
- Сжатие файла — только в однопроцессном сценарии; расширение через Native API; изменение геометрии приостанавливает потоки (SRWL — Windows Slim Reader/Writer Lock, тонкозернистая блокировка Windows).
- WSL1: `ENOLCK` — работа принципиально невозможна.

7.2.3 28.3. macOS / iOS

- `fcntl(F_FULLFSYNC)` по умолчанию (максимум долговечности); `MDBX_APPLE_SPEED_INSTEADOF_DURABILITY` — жертва durability ради скорости.
- SysV-семафоры по умолчанию (flavour `SYSV`).
- `fcntl(F_PREALLOCATE)` резервирует в конец — БД растёт вдвое быстрее штатного.

7.2.4 28.4. Android / bionic

- TLS и атомики отличаются от glibc; 32-бит Android — возможное зависание при откате коммита (исторически; с 0.11.7 проблем нет).

7.2.5 28.5. Контейнеры

- LXC: `boot_id`; Docker — PID uniqueness.
- tmpfs: `ENOSPC` от `fallocate` (игнорировать).
- NFS/CIFS/SMB: только эксклюзивный режим или кооперативный read-only; копирование работает.

7.2.6 28.6. Wine

- Не реализованы механизмы динамического изменения размера БД — задавайте фиксированный достаточный размер; `mdbx_module_handler(...)` при запуске для статической линковки.

Фрагмент из [examples/c++/35-platform-notes.c++](#) — платформенная оговорка в коде: «абсурдный» запрос геометрии обрабатывается через try/catch, потому что на tmpfs/WSL1/32-битной Windows `ENOSPC` / `ENOLCK` — штатная реакция, а не фатальный сбой:

```
// Проверка границ геометрии не обязана падать: запрашиваем «абсурдную»
// верхнюю границу и аккуратно обрабатываем результат.
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) {
    // На tmpfs/WSL1/32-битной Windows это может завершиться ошибкой —
    // для примера важно, что она перехвачена, а не уронила процесс.
    std::cout << "geometry probe: rejected (" << ex.what() << ")\n";
}
```

Полный код: [35-platform-notes.c++](#).

Примеры к главе: [examples/c++/35-platform-notes.c++](#).

7.2.7 28.7. Резюме главы 28

- Платформенные различия касаются блокировок, синхронизации и размеров адресного пространства.
- LXC/boot_id и page-cache — специфика контейнеров/Linux.
- Windows LockFileEx и F_FULLFSYNC на macOS — главные платформенные «грабли».

7.2.8 28.8. Чек-лист главы 28

- [] Linux: учесть boot_id (LXC), tmpfs ENOSPC от fallocate, при необходимости отказаться от THP.
- [] Windows: помнить о дорогих мелких транзакциях и лимитах 32-бит (/LARGEADDRESSAWARE).
- [] macOS: осознанно выбрать междуfcntl(F_FULLFSYNC) и MDBX_APPLE_SPEED_INSTEADOF_DURABILITY.
- [] Android/bionic: проверить поведение на 32-бит (историческое зависание при откате коммита — до 0.11.7).
- [] Контейнеры: LXC — boot_id, Docker — PID uniqueness; NFS/CIFS/SMB — только эксклюзивный или кооперативный read-only.
- [] Wine: задать фиксированный достаточный размер БД (динамическое изменение размера не работает).
- [] Обращать ENOSPC / ENOLCK на tmpfs/WSL1/32-бит Windows как штатные ошибки, а не фатальные сбои.

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

7.3.1 29.1. Что такое HSR и зачем он нужен

Когда база заполнена **из-за долгих читателей** (GC заморожен), библиотека вызывает колбэк HSR — единственный штатный способ приложения повлиять на исход конфликта «долгий читатель против писателя».

7.3.2 29.2. Сигнатура и параметры

```
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 — отставание проблемного читателя;
- gap — число попыток;
- space — объём, который освободится после завершения читателя;
- retry — счётчик повторных вызовов.

7.3.3 29.3. Возвращаемые значения

Значение	Действие
0	Колбэк подождал/решил; libmdbx пересканирует RLT и повторяет
1	Читательская транзакция прервана асинхронно; слот очистить немедленно
2+	Процесс-читатель убит; libmdbx сбрасывает его регистрацию

Вызывается только когда база заполнена из-за читателей.

7.3.4 29.4. Типовые реализации

- **Логирование** + возврат 0 (ждать).
- **Мягкое завершение**: пометить читателю флаг, вернуть 1.
- **Принудительное**: убить процесс читателя, вернуть 2.
- **Согласие на рост**: если позволяет геометрия — увеличить `upper` и вернуть 0.

7.3.5 29.5. HSR и SAFE_NOSYNC

В `SAFE_NOSYNC` «квази-долгий читатель» — это `steady`-коммит, а не живой слот: HSR на него не влияет. Управлять ростом нужно `авто-sync` (`syncbytes` / `syncperiod`).

Фрагмент из `examples/c++/36-hsr.c++` — HSR-колбэк: сообщает о проблемном читателе (`pid/lag/space`), просит его освободить снапшот и ждёт фактического освобождения, после чего возвращает `MDBX_RESULT_TRUE`. Регистрация — `env.set_HandleSlowReaders(hsr_callback)`; вместе с `SAFE_NOSYNC` колбэк остаётся главным средством против долгих читателей:

```
// Колбэк вызывается, когда БД «упирается» в предел из-за читателей,
// удерживающих старые снапшоты. Здесь: сообщаем о событии, просим читателя
// освободить снапшот и ждём его, после чего возвращаем MDBX_RESULT_TRUE
// («проблема устранена» — писатель продолжит).
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(); // дождаться фактического освобождения
    return MDBX_RESULT_TRUE;
}
```

Полный код: [36-hsr.c++](#).

7.3.6 29.6. Пример реализации

Фрагмент (C, иллюстрация): полная компилируемая версия — в [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; /* убить проблемного читателя */
    return 0; /* ждём ещё */
}
/* установка */
mdbx_env_set_hsr(env, my_hsr);
```

Примеры к главе: [examples/c++/36-hsr.c++](#).

7.3.7 29.7. Резюме главы 29

- HSR — колбэк разрешения конфликта с долгими читателями.
- Параметры: `laggard/gap/space/retry`; возвраты 0/1/2+.
- Типовые стратегии: ждать/убить/расти.
- С `SAFE_NOSYNC` управляйте ростом через `авто-sync`.

7.3.8 29.8. Чек-лист главы 29

- [] Установить HSR-колбэк (`mdbx_env_set_hsr` / `set_HandleSlowReaders`).
- [] Определить политику по `retry`: логирование, мягкое завершение, принудительное убийство, согласие на рост.
- [] Использовать параметры `laggard / gap / space / retry` для принятия решения в колбэке.

- [] Помнить семантику возвратов: 0 — ждать, 1 — прервать читателя, 2+ — убить процесс-читатель.
- [] При `SAFE_NOSYNC` настроить авто-супс (`syncbytes / syncperiod`) — HSR на steady-коммит не влияет.

7.4 Глава 30. Диагностика и отладка

7.4.1 30.1. Инструменты

`mdbx_chk` — проверка целостности. Ключевые опции:

Опция	Что делает
<code>-v...-vvvvv</code>	Подробность вывода
<code>-q</code>	Полная тишина
<code>-c</code>	Кооперативный (не эксклюзивный) режим; полная проверка — только эксклюзивный
<code>-w</code>	Read-write: откат к steady, проверка txnid мет
<code>-d</code>	Постраничный обход B+tree; без него нельзя найти lost/double-used страницы
<code>-i</code>	Игнор ложных ошибок порядка (кастомные компараторы)
<code>-s table</code>	Проверить только таблицу
<code>-0/-1/-2</code>	Конкретная мета; <code>-t / -T</code> — переключиться на неё

Exit-код: 0 = ошибок нет.

`MDBX_ENABLE_PROFGC` — профиль GC в `commit_latency.gc_prof`. Ключевые поля: `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` — стадии коммита: `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`. Для долгих читателей важны `txn_space_retired` и `txn_space_leftover`.

Отладочная сборка — `MDBX_DEBUG` / `MDBX_CHECKING` (-1..3); `MDBX_FORCE_ASSERTIONS` deprecated. Санитайзеры: `ENABLE_ASAN` / `ENABLE_UBSAN` / `ENABLE_MEMCHECK`; TSAN в CMake нет (вручную).

Фрагмент из `examples/c++/37-diagnostics.c++` — чтение `mdbx_txn_info` (через `txn.get_info(...): id/лаг/использованное/dirty`) и списка читателей (`mdbx_reader_list`, здесь `env.enumerate_readers(...)`) — те самые данные для алгоритмов «рост БД» и `MDBX_MAP_FULL` выше:

```
// Информация о текущей пишущей транзакции.
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();
}

// Список читателей (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);
```

Полный код: [37-diagnostics.c++](#).

7.4.2 30.2. Пошаговый алгоритм: рост БД

1. `mdbx_env_info_ex()` — меты и геометрия (`size_now` vs `size_upper`).

2. `mdbx_reader_list()` / `mdbx_reader_check()` — кто держит снапшоты.
3. `mdbx_stat -r` — `retained`.
4. PROFGC: `max_reader_lag`, `max_retained_pages`, `kicks`.
5. `mdbx_stat -p` — `newly` vs `cow` (новые страницы вместо переиспользования).
6. Решение: парковка/HSR/steady-point/дефрагментация; при `SAFE_NOSYNC` — авто-sync.

7.4.3 30.3. Пошаговый алгоритм: тормоз коммита

1. `mdbx_txn_commit_ex()` — разложить `whole` на стадии.
2. `sync` доминирует → режим долговечности; слишком частые `fsync` → батч или `NOMETASYNC` / `SAFE_NOSYNC`.
3. `gc_wallclock` велик → PROFGC: `work_rsteps` / `work_xpages` / `work_majflt` (фрагментация/большие значения).
4. `write` велик при малых данных → спилл рано (`dp_limit`) или дублирование грязных страниц.
5. `WRITEMAP` + `page-fault`'ы → `prefault_write_enable`.
6. `mdbx_stat -p` — `msync/fsync` счётчики.

7.4.4 30.4. Пошаговый алгоритм: MDBX_MAP_FULL

1. **Abort текущей write-транзакции** (продолжение после `resize` → `BAD_TXN`).
2. `mdbx_txn_info()` — `txn_space_limit_hard` vs `used`.
3. `mdbx_reader_list()` — старые снапшоты пинят пространство.
4. PROFGC — `kicks`, `max_reader_lag`, `max_retained_pages`.
5. Проверить HSR (`mdbx_env_set_hsr`), парковку, геометрию (`upper` занижен?).
6. Решение: HSR (ждать/убить/расти), выселение припаркованных, новый `steady`, увеличение `upper` до `open`, дефрагментация.

7.4.5 30.5. Пошаговый алгоритм: deadlock / MDBX_BUSY

1. Проверить дисциплину потоков: коды `THREAD_MISMATCH` / `TXN_OVERLAPPING` / `BAD_RSLT`.
2. При `NOSTICKYTHREADS` — исключить синхронные `write`-функции (`set_option/set_flags/set_geometry/sync/stat/defrag/close`) из потока, не владеющего пишущей транзакцией.
3. Аудит `fork()` (`resurrect`) и повторных `env_open`.
4. `mdbx_reader_list()` покажет чужих владельцев.

7.4.6 30.6. Пошаговый алгоритм: повреждение БД

1. `mdbx_chk -w -vvv` — локализовать (мета? дерево? GC? порядок?).
2. Проверить среду: `boot_id` (LXC?), некогерентность `page cache` (#269, `incoherence`).
3. Повреждена одна мета → `mdbx_chk -1/-2 + -T` (переключиться на валидную).
4. `MDBX_WANNA_RECOVERY` → `read-write` или `mdbx_env_open_for_recovery()` (с 0.12.7 не изменяет базу).
5. Recovery-проверки безопасны; восстановление из бэкапа (`mdbx_copy`).

7.4.7 30.7. Пошаговый алгоритм: утечка слотов читателей

1. `mdbx_reader_check(env, &dead)` — сколько мёртвых слотов.
2. Причина: потоки завершились без очистки (TLS-деструктор; `glibc #21031/#21032`; DSO-выгрузка).
3. Решение: явная `mdbx_thread_register/unregister`; паттерн `reset+renew`; `resurrect_after_fork`.

7.4.8 30.8. Чек-листы перед production-деплойем

- [] Геометрия задана до open (upper адекватен; движок сам выбирает дефолт $\approx$ золотое сечение ОЗУ).
- [] HSR-колбэк установлен.
- [] Режим синхронизации осознанно выбран.
- [] Release-сборка без ассертов.
- [] maxreaders покрывает число потоков.
- [] Долгие чтения используют парковку/reset+renew.
- [] Тест после сбоя: kill -9 + открытие + mdbx_chk.

Примеры к главе: [examples/c++/37-diagnostics.c++](#).

7.4.9 30.9. Резюме главы 30

- Инструменты: chk, PROFGC, commit_latency, txn_info, reader_check, отладочная сборка.
- 6 пошаговых алгоритмов: рост, тормоз коммита, MAP_FULL, deadlock, повреждение, утечка слотов.
- Чек-лист перед деплоем обязателен.

7.4.10 30.10. Чек-лист главы 30

- [] Освоить mdbx_chk (опции -d, -w, -0/-1/-2 / -T) как основной инструмент целостности.
- [] Диагностировать рост БД по алгоритму §30.2 (mdbx_reader_list, mdbx_stat -r / -p, PROFGC).
- [] Тормоз коммита раскладывать по стадиям MDBX_commit_latency (mdbx_txn_commit_ex) — §30.3.
- [] При MDBX_MAP_FULL — abort транзакции, проверка читателей, HSR и геометрии — §30.4.
- [] Deadlock/ MDBX_BUSY — аудит потоковой дисциплины и NOSTICKYTHREADS — §30.5.
- [] Повреждение БД — mdbx_chk -w -vvv, восстановление из бэкапа (mdbx_copy) — §30.6.
- [] Утечку слотов читателей проверять через mdbx_reader_check — §30.7.
- [] Пройти чек-лист production-деплойа §30.8 (включая kill -9 + открытие + mdbx_chk).

7.5 Глава 31. Миграция с LMDB

7.5.1 31.1. Зачем мигрировать

libmdbx — переработанный LMDB: больше режимов долговечности, GC вместо free-list, жёстче гарантии восстановления, выше производительность записи при батчинге. **Формат данных не совместим с LMDB**: тройка мета-страниц, двухфазный коммит, контрольные суммы и GC-дерево — собственные изменения; файлы LMDB напрямую не открываются, миграция выполняется переносом данных.

7.5.2 31.2. Совместимость форматов

- Файлы **LMDB не открываются** libmdbx напрямую: форматы разные (тройка мета-страниц, двухфазный коммит, контрольные суммы, GC-дерево).
- Между собой совместимы БД libmdbx 0.11.x $\Leftrightarrow$ 0.12.x (формат заморожен с v11.3).
- Миграция с LMDB — перенос данных (mdbx_dump / mdbx_load, mdbx_copy), а не переименование файла.

7.5.3 31.3. Breaking changes

- MDBX_NOLOCK убран.
- MDBX_NOTLS $\rightarrow$ MDBX_NOSTICKYTHREADS.

- Другие переименования и новые требования к флагам — сверяйте с `mbdx.h`.

7.5.4 31.4. Таблица соответствия `mdb_` → `mbdx_`

LMDB	libmdbx
<code>mdb_env_create</code>	<code>mbdx_env_create</code>
<code>mdb_env_open</code>	<code>mbdx_env_open</code>
<code>mdb_txn_begin</code>	<code>mbdx_txn_begin</code>
<code>mdb_dbi_open</code>	<code>mbdx_dbi_open</code>
<code>mdb_put/get/del</code>	<code>mbdx_put/get/del</code>
<code>mdb_cursor_get</code>	<code>mbdx_cursor_get</code>
<code>MDB_NOTLS</code>	<code>MDBX_NOSTICKYTHREADS</code>

Фрагмент из [examples/c/38-migration.c](#) — карта соответствия `mdb_*` → `mbdx_*` из заголовка примера:

```
// Карта макросов/имён при переносе кода с LMDB (полная таблица — в §31.4):
//  mdb_env_create   → mbdx_env_create
//  mdb_env_open     → mbdx_env_open
//  mdb_env_close    → mbdx_env_close
//  mdb_env_set_mapsize → mbdx_env_set_geometry (другой API!)
//  mdb_txn_begin    → mbdx_txn_begin
//  mdb_txn_commit   → mbdx_txn_commit
//  mdb_txn_abort    → mbdx_txn_abort
//  mdb_dbi_open     → mbdx_dbi_open
//  mdb_get/put/del  → mbdx_get/mbdx_put/mbdx_del
//  mdb_cursor_open/get → mbdx_cursor_open/mbdx_cursor_get
//  MDB_NOTLS        → MDBX_NOSTICKYTHREADS
//  MDB_NOSYNC       → MDBX_SAFE_NOSYNC или MDBX_UTCALLY_NOSYNC
//  MDB_APPEND       → MDBX_APPEND
//  MDB_INTEGERKEY   → MDBX_INTEGERKEY
//  mdb_strerror     → mbdx_strerror
```

Полный код: [38-migration.c](#).

7.5.5 31.5. Поведенческие отличия

- Три меты vs две; двухфазный коммит.
- Контрольные суммы страниц.
- Больше диагностических кодов (`WANNA_RECOVERY`, `MVCC_RETARDED` и др.).
- Пустые ключи/значения разрешены.

7.5.6 31.6. Чек-лист миграции

1. Сделайте бэкап (`mbdx_copy -c` или `dump+load`).
2. Откройте базу в read-only — проверьте `mbdx_chk`.
3. Замените вызовы по таблице соответствия.
4. Прогоните тесты под ASAN/UBSAN.
5. Настройте геометрию/режимы под новый движок (не переносите вслепую LMDB-настройки).

7.5.7 31.7. Резюме главы 31

- Формат данных `libmdbx` не совместим с LMDB: тройка мета-страниц, двухфазный коммит, контрольные суммы, GC-дерево; файлы LMDB напрямую не открываются.
- Миграция — перенос данных (`mbdx_dump` / `mbdx_load`, `mbdx_copy`), а не переименование файла; между собой совместимы БД 0.11.x ↔ 0.12.x (формат заморожен с v11.3).

- Breaking changes: `MDBX_NOLOCK` убран, `MDBX_NOTLS` → `MDBX_NOSTICKYTHREADS`; остальные переименования сверяйте с `mdbx.h`.
- Имена функций меняются предсказуемо (`mdb_env_create` → `mdbx_env_create`), но `mdb_env_set_mapsize` → `mdbx_env_set_geometry` — другой API.
- Поведенческие отличия: три меты vs две, контрольные суммы страниц, больше диагностических кодов, разрешены пустые ключи/значения.
- Порядок миграции: бэкап → read-only проверка `mdbx_chk` → замена вызовов → тесты под ASAN/UBSAN → настройка геометрии/режимов заново.

7.5.8 31.8. Чек-лист главы 31

- [] Сделать бэкап (`mdbx_copy -c` или `dump+load`) до любых изменений.
- [] Открыть базу в read-only и проверить её `mdbx_chk`.
- [] Заменить вызовы по таблице соответствия §31.4, включая `mdb_env_set_mapsize` → `mdbx_env_set_geometry`.
- [] Заменить флаги: `MDBX_NOLOCK` убран, `MDBX_NOTLS` → `MDBX_NOSTICKYTHREADS`.
- [] Прогнать тесты под ASAN/UBSAN.
- [] Настроить геометрию и режимы под `libmdbx` заново, не перенося LMDB-настройки вслепую.

Примеры к главе: [examples/c/38-migration.c](#) (карта соответствия `mdb_` → `mdbx_` — в заголовке файла).

7.6 Глава 32. Паттерны проектирования на `libmdbx`

7.6.1 32.1. Паттерн 1: Key-value с автоинкрементным ID

Для монотонных ID есть и встроенный механизм — `mdbx_dbi_sequence()` (атомарный счётчик таблицы), и классический переносимый подход — счётчик в служебной таблице. Пример ниже демонстрирует второй вариант: он одинаково работает во всех версиях `libmdbx` и не зависит от `MDBX_LIFORECLAIM`-тонкостей.

Фрагмент из [examples/c++/39-pattern-sequence-id.c++](#) — автоинкрементный ID: счётчик лежит в таблице `meta` и инкрементируется в той же пишущей транзакции, что и вставка записи:

```
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;
}
```

Полный код: [39-pattern-sequence-id.c++](#).

7.6.2 32.2. Паттерн 2: Вторичный индекс (DUPSORT)

Основная таблица + индекс «поле → список ID» (Том II, глава 8). Обновлять в одной транзакции.

Фрагмент из [examples/c++/40-pattern-secondary-index.c++](#) — вторичный индекс «поле → список ID»: основная таблица `users` и индекс `by_role` (DUPSORT, `value_mode::multi`) обновляются в одной транзакции:

```
auto txn = env.start_write();
// Основная таблица: id → {name, role}.
auto users = txn.create_map("users", mdbx::key_mode::ordinal, mdbx::value_mode::single);
// Индекс: role → список id (мультизначения).
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();
```

Полный код: [40-pattern-secondary-index.c++](#).

7.6.3 32.3. Паттерн 3: Составной ключ

Конкатенация полей + компаратор (или big-endian для чисел). Поиск по префиксу — `SET_RANGE`.

Фрагмент из [examples/c++/41-pattern-composite-key.c++](#) — составной ключ: поля фиксированной ширины упаковываются в big-endian, при котором лексикографический порядок совпадает с числовым (диапазонный поиск — `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]);
}
```

Полный код: [41-pattern-composite-key.c++](#).

7.6.4 32.4. Паттерн 4: Очередь задач (table-as-queue)

`MDBX_DUPSORT` + sequence: ключ — приоритет/номер, значение — задача. Естественный порядок выдачи.

Фрагмент из [examples/c++/42-pattern-queue.c++](#) — очередь задач: ключ — монотонный номер (ordinal), потребитель забирает первый элемент курсором и удаляет его:

```
// Потребитель: забирает первый элемент через курсор и удаляет.
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";
```

Полный код: [42-pattern-queue.c++](#).

7.6.5 32.5. Паттерн 5: Кольцевой буфер

Ограниченный размер: при превышении удаляем самые старые ключи (курсор + `del`).

Фрагмент из [examples/c++/43-pattern-ring-buffer.c++](#) — кольцевой буфер последних N: фиксированные слоты `0..N-1`, запись перезаписывает `counter % N`:

```
// Пишем 12 элементов в буфер на 5 слотов.
{
    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();
}

```

Полный код: [43-pattern-ring-buffer.c++](#).

7.6.6 32.6. Паттерн 6: Полносканирующий итератор

Курсор + `get_batch / bunch_delete` для массовой обработки.

Фрагмент из [examples/c++/44-pattern-full-scan.c++](#) — полное сканирование курсором `to_first()` → `to_next()` с фильтром:

```

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

// Полное сканирование с фильтром «чётные ключи».
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";

```

Полный код: [44-pattern-full-scan.c++](#).

7.6.7 32.7. Паттерн 7: Репликация через per-table txnid

Маркеры изменений: хранить номер транзакции последнего изменения; вторичный процесс опрашивает и копирует дельты.

Фрагмент из [examples/c++/45-pattern-replication.c++](#) — репликация через per-table txnid: значение хранит версию (`data|N`), реплика читает только записи новее своего `last_seen`:

```

// Реплика читает всё новее своего последнего 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;
};

```

Полный код: [45-pattern-replication.c++](#).

7.6.8 32.8. Паттерн 8: Read-your-writes через clone / embark_read

`mdbx_txn_clone / embark_read` — собственные записи видны немедленно в той же логической операции без отдельной транзакции.

Фрагмент из [examples/c++/46-pattern-read-your-writes.c++](#) — read-your-writes: та же пишущая транзакция немедленно читает собственное изменение; после `commit` оно видно и свежему читателю:

```

// Пишущая транзакция сразу читает собственное изменение.
{
    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();
}

// После коммита изменение видно и свежему читателю.
{
    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();
}

```

Полный код: [46-pattern-read-your-writes.c++](#).

Каждый паттерн: постановка задачи → решение с кодом → компромиссы → альтернативы (в полной версии базы знаний — `knowledge-base/`).

7.6.9 32.9. Резюме главы 32

- Автоинкрементный ID: счётчик в служебной таблице, инкрементируется в той же write-транзакции, что и вставка; альтернатива — встроенный `mdbx_dbi_sequence()`.
- Вторичный индекс (DUPSORT) «поле → список ID» обновляется в одной транзакции с основной таблицей.
- Составной ключ: поля фиксированной ширины в big-endian (лексикографический порядок = числовой), диапазонный поиск — `SET_RANGE`.
- Очередь задач: ordinal-ключ (монотонный номер), потребитель забирает первый элемент курсором и удаляет.
- Кольцевой буфер: фиксированные слоты `0..N-1`, запись перезаписывает `counter % N`.
- Полное сканирование — курсор `to_first()` → `to_next()`, массовая обработка — `get_batch / bunch_delete`.
- Репликация через per-table txnid (маркер версии в значении) и read-your-writes через `mdbx_txn_clone / embark_read`.

7.6.10 32.10. Чек-лист главы 32

- [] Для монотонных ID выбрать `mdbx_dbi_sequence()` или счётчик в служебной таблице (переносимый вариант).
- [] Вторичный индекс (DUPSORT, `value_mode::multi`) обновлять в той же транзакции, что и основную запись.
- [] Для составных ключей использовать big-endian/компаратор и диапазонный поиск `SET_RANGE`.
- [] Очередь задач реализовать через ordinal-ключ и удаление первого элемента курсором.
- [] Для ограниченных структур (кольцевой буфер) удалять самые старые записи при переполнении.
- [] Массовые выборки/удаления выполнять через `get_batch / bunch_delete`.
- [] Репликацию строить на маркерах txnid, реплику — на read-only снапшоте с фильтром по `last_seen`.

Примеры к главе: [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 Глава 33. Roadmap и будущее: MithrilDB

7.7.1 33.1. Текущее состояние

libmdbx на 2026 год — зрелый движок (0.15.x devel-канон), формат БД заморожен с 2018 года. Фундаментальные улучшения GC/freelist планируются только в следующем поколении.

7.7.2 33.2. MithrilDB

Общий API для нескольких форматов хранения; амальгамация упрощает распространение.

7.7.3 33.3. Репликация

Предпосылки: подписка на изменения, ранняя очистка GC (2025), нелинейная обработка GC (сквозное отслеживание использования страниц читаемыми снапшотами).

7.7.4 33.4. Прочее в roadmap

- Подписка на изменения (почтовые ящики, кольцевые буферы).

- Шифрование и сжатие на уровне движка.
- Поточковый режим чтения/записи длинных значений («Streaming BLOBs») — без ручного фрагментирования на чанки, см. [Ограничения и подводные камни](#).
- SWIG и кроссязыковое взаимодействие.

7.7.5 33.5. Резюме главы 33

- libmdbx на 2026 год — зрелый движок (0.15.x devel-канон); формат БД заморожен с 2018 года.
- Фундаментальные улучшения GC/freelist планируются только в следующем поколении — MithrilDB: общий API для нескольких форматов хранения, амальгамация упрощает распространение.
- Предпосылки репликации: подписка на изменения, ранняя очистка GC (2025), нелинейная обработка GC (сквозное отслеживание использования страниц читаемыми снапшотами).
- Прочее в roadmap: шифрование и сжатие на уровне движка, потоковый режим чтения/записи длинных значений («Streaming BLOBs»), SWIG и кроссязыковое взаимодействие.

7.7.6 33.6. Чек-лист главы 33

- [] При долгосрочном планировании учитывать, что формат БД заморожен с 2018 года (совместимость 0.11.x ↔ 0.12.x).
 - [] Не ждать фундаментальных улучшений GC/freelist в текущем поколении — следить за MithrilDB.
 - [] Для репликации опираться на доступные предпосылки (подписка на изменения, ранняя очистка GC).
 - [] Оценивать roadmap-фичи (шифрование, сжатие, потоковый режим длинных значений / Streaming BLOBs, SWIG) как будущие, а не текущие возможности.
-

7.8 Итог тома

Вы знаете правила и советы, платформенные нюансы, HSR, диагностику по проверенным алгоритмам, миграцию с LMDB и паттерны проектирования.

Что дальше: Том VI — привязки (bindings): как использовать libmdbx из Rust, Go, Python, Node.js, .NET, C++, Dart, Nim, Java, Haskell, Ruby, Scala.

8. Том VI. Привязки (bindings)

Уровень: для разработчиков, работающих на языках, отличных от C/C++. **Цель тома:** вы знаете, как использовать libmdbx из своего языка, понимаете специфику и ограничения каждого биндинга. **Честность:** факты по конкретным обёрткам (репозиторий, версия, баги) требуют анализа их репозитория и issues. Здесь приведены архитектурные следствия (NoStickyThreads, Send/Sync, GC/pinning, GIL — Global Interpreter Lock, глобальная блокировка интерпретатора в Python); позиции, требующие сверки с репозиторием, помечены явно.

8.1 Глава 34. Обзор экосистемы биндингов

Официально отслеживаемые биндинги: **Rust, Go, Node.js, Zig, Python, .NET (C#), C++, Dart, Nim, Java, Haskell, Ruby, Scala**. Плюс неофициальные (openresty/luajit и др.).

Биндинг	Язык	Статус	Подробнее
mdbx.h++	C++	Зрелый (официальный C++ API)	Гл. 40
mdbx-rs / mdbx-sys	Rust	Зрелый/активный	Гл. 35
mdbx-go	Go	Активный	Гл. 36
python-lmdbx / mdbx-py	Python	Активный	Гл. 37
node-mdbx	Node.js	Активный	Гл. 38
mdbx-zig (lmdbx-zig)	Zig	Официально отслеживаемый	\$34.2
libmdbx-dotnet	.NET / C#	Активный	Гл. 39
mdbx-dart / Isar	Dart (Flutter)	Активный (Isar использует libmdbx)	Гл. 41
Nim	Nim	—	Гл. 42
Java	Java	—	Гл. 43
Haskell	Haskell	—	Гл. 44
Ruby	Ruby	—	Гл. 45
Scala	Scala	—	Гл. 46

Нюанс: карточки Гл. 42–46 (Nim, Java, Haskell, Ruby, Scala) — краткие; детали репозитория требуют анализа (см. пометки в каждой карточке).

8.1.1 34.1. Что покрывает каждый биндинг

Типичная картина: биндинг оборачивает **C API** (`mbx_env_*`, `mbx_txn_*`, `mbx_dbi_*`, `mbx_cursor_*`, `MDBX_val`) в идиоматику языка. Полнота покрытия различается: ядро (`env/txn/get/put/del/cursor`) покрыто везде; продвинутое API (`cache_get`, `defrag`, `clone`, `embark_read`, `estimate_*`) — не всегда. Сверяйте с документацией конкретного биндинга.

8.1.2 34.2. Zig (краткая справка)

`mbx-zig` (репозиторий `lmdbx-zig`) — официально отслеживаемый биндинг для Zig. Известные особенности: Zig активно используется для кросскомпиляции C/C++; для обхода багов тулчейна может понадобиться `MDBX_HAVE_BUILTIN_CPU_SUPPORTS=0`. Отдельная карточка не выделена — детали репозитория и версии требуют анализа.

8.1.3 34.3. Резюме главы 34

- Официально отслеживаемые биндинги: Rust, Go, Node.js, Zig, Python, .NET (C#), C++, Dart, Nim, Java, Haskell, Ruby, Scala; плюс неофициальные (openresty/luar и др.).
- Каждый биндинг оборачивает С API (`mdbx_env_*`, `mdbx_txn_*`, `mdbx_dbi_*`, `mdbx_cursor_*`) в идиоматику языка.
- Ядро (`env/txn/get/put/del/cursor`) покрыто везде; продвинутое API (`cache_get`, `defrag`, `clone`, `embark_read`, `estimate_*`) — не всегда.
- Зрелые/активные: `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`) — официально отслеживаемый; для обхода багов тулчейна может понадобиться `MDBX_HAVE_BUILTIN_CPU_SUPPORTS=0`.
- Карточки Nim/Java/Haskell/Ruby/Scala — краткие; детали репозитория требуют анализа.

8.1.4 34.4. Чек-лист главы 34

- [] Выбрать биндинг по таблице главы и сверить его статус с репозиторием.
- [] Проверить, какие продвинутое API (`cache_get`, `defrag`, `clone`, `embark_read`, `estimate_*`) поддерживает биндинг.
- [] Для Zig при проблемах тулчейна учесть `MDBX_HAVE_BUILTIN_CPU_SUPPORTS=0`.
- [] Для кратких карточек (Nim/Java/Haskell/Ruby/Scala) проанализировать репозиторий биндинга до production.

8.2 Глава 35. Rust (`mdbx-sys` / `mdbx-rs`)

Обзор. `mdbx-sys` — низкоуровневые FFI-привязки к С API; `mdbx-rs` — безопасная обёртка (идиоматичный Rust). Статус — активный/зрелый. (Детали репозитория требуют сверки.)

Установка. Через cargo: `mdbx-sys = "..."`, `mdbx-rs = "..."`.

Базовый пример (упрощённый, общий для типа):

```
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(())
}
```

(Точный синтаксис зависит от версии биндинга — сверить с его README.)

Критические нюансы (из архитектуры libmdbx):

- **Send/Sync.** Транзакция и курсор привязаны к потоку (sticky). Безопасная обёртка должна отражать это в типах: транзакция — `!Sync` (использование из двух потоков = `THREAD_MISMATCH`). Для `async-runtime` (`tokio` и др.) обязателен `MDBX_NOSTICKYTHREADS`, т.к. `future` может продолжиться в другом OS-потоке.
- **Use-after-free после commit/abort.** `MDBX_val` указывает в mmap; после завершения транзакции указатели недействительны. Обёртка должна ограничить время жизни значений временем транзакции (lifetime), а не передавать `&[u8]` наружу.
- **Блокирующие вызовы в async-контексте.** Commit и запись блокируют поток; в `async-runtime` — только через `spawn_blocking`, иначе блокируется весь воркер.

Производительность. Накладные биндинга минимальны (FFI + ownership-проверки); чтение остаётся близким к С. (Сверка с бенчмарками биндинга.)

Пробелы. Зависит от версии: `cache_get`, `defrag`, `clone` и др. могут отсутствовать.

Резюме карточки. Двухслойная связка: `mdbx-sys` — низкоуровневые FFI-привязки к С API, `mdbx-rs` — безопасная идиоматичная обёртка (установка через cargo). Главные архитектурные следствия — sticky-транзакции (`!Sync`, `MDBX_NOSTICKYTHREADS` обязателен для `async-runtime`) и недействительность `MDBX_val`-указателей после `commit/abort`; блокирующие вызовы в `async` — только через

`spawn_blocking`. Накладные расходы минимальны (FFI + ownership-проверки), продвинутые API (`cache_get`, `defrag`, `clone`) зависят от версии.

Чек-лист карточки.

- [] Для `async-runtime` (tokio и др.) выставить `MDBX_NOSTICKYTHREADS`.
- [] Убедиться, что обёртка ограничивает время жизни значений транзакцией (не выдаёт `&[u8]` после `commit/abort`).
- [] Блокирующие `commit/записи` выполнять через `spawn_blocking`.
- [] Сверить наличие `cache_get/defrag/clone` с версией биндинга.

8.3 Глава 36. Go (mdbx-go)

Обзор. `mdbx-go` — Go-биндинг через CGO. Статус — активный.

Установка. `go get github.com/.../mdbx-go`.

Базовый пример:

```
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()
```

(Сигнатуры сверить с репозиторием.)

Критические нюансы (из архитектуры):

- **NoStickyThreads** всегда **выставляется**. Go-горутина может мигрировать между OS-потоками, поэтому биндинг открывает окружение с `MDBX_NOSTICKYTHREADS`.
- **Один объект транзакции нельзя использовать из двух горутин одновременно** — это гонка.
- **Deadlock-риск:** функции, требующие write-блокировку среды (`Env.SetOption`, `Env.Sync`, `Env.Stat`, `Env.Defrag`, `Env.Close`), при активной write-транзакции в другой горутине могут взаимоблокироваться. Паттерн: одна транзакция — одна горутина, либо явная сериализация через `channel/mutex`.
- `runtime.LockOSThread` нужен, если хотите зафиксировать горутину на OS-потоке (снижает риски при работе с TLS-слотом читателя).
- **CGO overhead** — на каждую операцию границы CGO; для хот-путей группируйте операции в батч.

Пробелы. Полнота продвинутого API — сверять с репозиторием.

Резюме карточки. `mdbx-go` — активный CGO-биндинг (установка — `go get`). Горутины мигрируют между OS-потоками, поэтому биндинг всегда открывает среду с `MDBX_NOSTICKYTHREADS`; один объект транзакции нельзя использовать из двух горутин одновременно. Главные риски — deadlock на write-функциях среды и CGO-overhead на каждой операции, который лечится батчингом.

Чек-лист карточки.

- [] Одна транзакция — одна горутина (либо явная сериализация через `channel/mutex`).
- [] Не вызывать `Env.SetOption / Env.Sync / Env.Stat / Env.Defrag / Env.Close` из другой горутины при активной write-транзакции.
- [] При необходимости зафиксировать горутину через `runtime.LockOSThread`.
- [] Для хот-путей группировать операции в батчи (граница CGO).

8.4 Глава 37. Python (python-lmdbx / mdbx-py)

Обзор. Python-биндинг поверх C API. Статус — активный (*сверка*).

Критические нюансы (из архитектуры):

- **GIL и блокирующие вызовы.** Долгие read-транзакции и массовые записи держат GIL → другие потоки Python блокируются. Освобождайте GIL в биндинге при длительных операциях.
- **Указатели в mmap действительны, пока жива транзакция.** Python GC не управляет этим — нужен явный context manager (`with env.begin(...)`), чтобы гарантировать commit/abort вовремя.
- **fork() после import.** При multiprocessing вызывайте `resurrect_after_fork` в дочернем процессе.
- **ctypes vs CFFI vs pybind.** Выбор влияет на накладные; для батчей — минимизируйте число вызовов границы.

Пробелы. Сверка с репозиторием (Python 3.x версии, wheels, продвинутое API).

Резюме карточки. Python-биндинг поверх C API, статус активный (*сверка*). Ключевое — GIL: долгие read-транзакции и массовые записи блокируют другие потоки Python; время жизни mmap-указателей нужно гарантировать явно — context manager'ом (`with env.begin(...)`). При multiprocessing в дочернем процессе обязателен `resurrect_after_fork`; выбор ctypes/CFFI/pybind влияет на накладные расходы.

Чек-лист карточки.

- [] Гарантировать commit/abort context manager'ом (`with env.begin(...)`).
 - [] Освободить GIL в биндинге при длительных операциях.
 - [] В дочернем процессе после `fork()` вызывать `resurrect_after_fork`.
 - [] Для батчей минимизировать число вызовов границы (ctypes/CFFI/pybind).
-

8.5 Глава 38. Node.js (node-mdbx)

Обзор. Node.js-биндинг (N-API). Статус — активный (*сверка*).

Критические нюансы (из архитектуры):

- **Event loop и блокирующие вызовы.** Синхронный C API блокирует цикл событий. Обёртка должна предлагать async-варианты (worker threads / libuv) для больших операций.
- **Buffer ↔ MDBX_val.** Node.js Buffer — владеющая обёртка; при передаче в libmdbx следите за временем жизни (не передавайте буфер, который может быть освобождён).
- **NoStickyThreads** — аналогично Go, если используется worker threads.

Пробелы. Сверка с репозиторием.

Резюме карточки. node-mdbx — активный N-API-биндинг (*сверка*). Синхронный C API блокирует event loop — обёртка должна предлагать async-варианты (worker threads/libuv) для больших операций; при worker threads нужен `MDBX_NOSTICKYTHREADS` (как в Go). Node.js Buffer — владеющая обёртка: следите за её временем жизни при передаче в libmdbx.

Чек-лист карточки.

- [] Большие операции выполнять через async-варианты (worker threads/libuv), не блокируя event loop.
 - [] Не передавать буферы, время жизни которых не гарантировано.
 - [] С worker threads выставить `MDBX_NOSTICKYTHREADS`.
 - [] Сверить возможности биндинга с репозиторием.
-

8.6 Глава 39. .NET / C# (libmdbx-dotnet)

Обзор. .NET-биндинг поверх C API. Альтернатива в экосистеме — LightningDB (не libmdbx!) — проверяйте, что именно вы используете). Статус — активный (*сверка*).

Критические нюансы (из архитектуры):

- **Pinning.** `MDBX_val` указывает в `mmap`; если значение — управляемый буфер, .NET GC может его переместить. Решение: `fixed / pinned`, либо `marshal`-копия, либо `Span<T>`.
- **SafeHandle для env/txn/cursor.** Корректное освобождение через `SafeHandle/finalizer`.
- **GC finalizer и порядок.** Закрывайте в правильном порядке: `env` → `txn` → `cursor`; висячие хендлы дают `BAD_DBI / BAD_TXN`.
- **Непредсказуемое время жизни** из-за GC — не держите длинные транзакции между сборками.

Пробелы. Сверка с репозиторием (версии, платформы, продвинутое API).

Резюме карточки. `libmdbx-dotnet` — активный .NET-биндинг поверх C API (*сверка*); не путать с LightningDB (это не libmdbx). Главное — `pinning`: `MDBX_val` указывает в `mmap`, а .NET GC может переместить управляемый буфер (`fixed / pinned`, `marshal`-копия или `Span<T>`). Освобождение — через `SafeHandle/finalizer` в порядке `env` → `txn` → `cursor`, висячие хендлы дают `BAD_DBI / BAD_TXN`.

Чек-лист карточки.

- [] Пиннировать управляемые буферы (`fixed / marshal`-копия/ `Span<T>`), не полагаясь на GC.
- [] Освободить `env/txn/cursor` через `SafeHandle/finalizer` в правильном порядке.
- [] Не держать длинные транзакции между сборками GC.
- [] Проверить, что используется именно libmdbx-биндинг, а не LightningDB.

8.7 Глава 40. C++ (mdbx.h++)

Обзор. Официальный C++ API (`mdbx.h++`), канон v0.15.0-263. Зрелый.

Иерархия классов:

- `env / env_managed` — окружение; `txn / txn_managed` — транзакции; `cursor / cursor_managed` — курсоры; `map_handle` — таблица; `slice / buffer` — ключи/значения.
- Управляемые варианты (`*_managed`) — RAII: автоматическое закрытие при выходе из области видимости.

Нюанс: `slice` — невладеющий, `buffer` — владеющий. `mdbx::slice` — лишь представление над `MDBX_val` (аналог `std::string_view`): ссылка на чужие байты без копирования. Поэтому `mdbx::slice(std::to_string(k))` в аргументе вызова безопасна (временная строка живёт до конца полного выражения, а `insert()` копирует байты в базу синхронно), но хранить такой срез дольше оператора — висячая ссылка. `mdbx::buffer` по умолчанию хранит собственную копию; режим «ссылка» включается явно (`make_reference=true`) — тогда действуют правила времени жизни `slice`.

Типизированные операции. Map-over-keys/values через трансляцию типов; параметры-структуры с fluent-установщиками (геометрия, режим, долговечность, reclaiming).

Исключения. Иерархия `mdbx::error` → `mdbx::exception` (от `std::runtime_error`) → `mdbx::fatal`; 35 типизированных исключений (`bad_map_id`, `db_corrupted`, `db_full`, `key_exists`, `not_found`, `transaction_ousted` и др.). Фатальные ошибки приводят к завершению.

Фрагмент из `examples/c++/47-cpp-api.cpp` — типизированные map-операции (`insert / upsert` через `map_handle`) и типизированные исключения (`mdbx::key_exists` и др.):

```
{
    auto txn = env.start_write(); // RAII: сам закроется/откатится при исключении
    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"));

    // Типизированные исключения по коду ошибки.
```

```
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();
}
```

Полный код: [47-cpp-api.c++](#).

C++20. Концепты, аллокаторы (`buffer<>` с политиками владения), `inplace_storage_size_rounding`.

Пример:

Фрагмент (C++, иллюстрация): полная компилируемая версия — в [examples/c++/47-cpp-api.c++](#). Операции map-стиля (`insert`, `upsert`, `get`, `erase`) — методы транзакции, а не таблицы.

```
#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();
}
```

Проверьте точный синтаксис по вашей версии `mdbx.h++`.

Критические нюансы:

- Значение `upper` для новой БД движков выбирает сам (~золотое сечение ОЗУ, ограничено mmap-лимитом ≈ 140 ТБ на 64-бит); `TOO_LARGE / ENOMEM` возможны при явно чрезмерном `upper` (например, под ASAN/Valgrind). Всегда задавайте геометрию явно.
- Исключения vs коды возврата: выбирайте единый стиль; `mdbx::error` ловится по типам.

Пробелы. `key_mode::msgpack` объявлен, но не реализован.

Резюме карточки. `mdbx.h++` — официальный зрелый C++ API (канон v0.15.0-263): RAII-классы `env_managed` / `txn_managed` / `cursor_managed`, типизированные операции и 35 типизированных исключений через `mdbx::error`. Геометрию задавайте явно — `upper` для новой БД движков выбирает сам (~золотое сечение ОЗУ). Известный пробел: `key_mode::msgpack` объявлен, но не реализован.

Чек-лист карточки.

- [] Использовать RAII-варианты (`*_managed`) для автоматического закрытия.
- [] Всегда задавать геометрию явно (умолчательный `upper` выбирает движок).
- [] Выбрать единый стиль ошибок: исключения `mdbx::error` (ловятся по типам) или коды возврата.
- [] Не использовать `key_mode::msgpack` (объявлен, но не реализован).

Примеры к главе: [examples/c++/47-cpp-api.c++](#).

8.8 Глава 41. Dart (mdbx-dart / Isar)

Обзор. `mdbx-dart` — биндинг для Dart/Flutter; **Isar** — популярная локальная БД Flutter, использующая `libmdbx`. Статус — активный.

Критические нюансы (из архитектуры):

- **Isolate-специфика.** Dart-изоляты — отдельные потоки; транзакции не пересекают изоляты (sticky thread). Каждый изолят — свои транзакции.
- **Мобильные платформы.** Встроенная БД на Android/iOS: аккуратно с парковкой длинных чтений и малым WAF (продление жизни flash).

Пробелы. Сверка с репозиторием.

Резюме карточки. `mdbx-dart` — активный биндинг для Dart/Flutter; на `libmdbx` построена популярная локальная БД Isar. Транзакции не пересекают изоляты (`sticky thread`) — каждый изолят работает со своими транзакциями. На мобильных платформах (Android/iOS) важны парковка длинных чтений и малый WAF (продление жизни `flash`).

Чек-лист карточки.

- [] Не передавать транзакции между изолятами (`sticky thread`).
- [] На Android/iOS аккуратно парковать длинные чтения.
- [] Следить за малым WAF для продления жизни `flash`.
- [] Сверить детали с репозиторием.

8.9 Глава 42. Nim

Обзор. Nim-биндинг. *(Данные о репозитории и версии требуют анализа репозитория биндинга.)*

Ожидаемые нюансы (из архитектуры): управление памятью (GC Nim не владеет `mmap`-указателями — нужны ручные ограничения времени жизни); привязка к потокам (`sticky`).

Резюме карточки. Nim-биндинг; данные о репозитории и версии требуют анализа репозитория. Ожидаемые нюансы: Nim GC не владеет `mmap`-указателями — нужны ручные ограничения времени жизни; транзакции привязаны к потокам (`sticky`).

Чек-лист карточки.

- [] Ручно ограничивать время жизни `mmap`-указателей (GC Nim ими не владеет).
- [] Держать транзакции в породившем их потоке (`sticky`).
- [] Сверить репозиторий и версию биндинга.

8.10 Глава 43. Java

Обзор. Java-биндинг (JNI). *(Данные требуют анализа репозитория.)*

Ожидаемые нюансы (из архитектуры): pinning критических буферов (JNI `GetByteArrayElements` или `direct ByteBuffer`); освобождение `native`-хендлов (`env/txn/cursor`) в правильном порядке; многопоточность (каждый поток — свои транзакции).

Резюме карточки. Java-биндинг через JNI; данные требуют анализа репозитория. Ожидаемые нюансы: pinning критических буферов (JNI `GetByteArrayElements` или `direct ByteBuffer`), освобождение `native`-хендлов (`env/txn/cursor`) в правильном порядке, у каждого потока — свои транзакции.

Чек-лист карточки.

- [] Пиннировать критические буферы (JNI `GetByteArrayElements` / `direct ByteBuffer`).
- [] Освобождать `native`-хендлы (`env/txn/cursor`) в правильном порядке.
- [] Не использовать одну транзакцию из нескольких потоков.
- [] Сверить данные с репозиторием биндинга.

8.11 Глава 44. Haskell

Обзор. Haskell-биндинг (FFI). *(Данные требуют анализа репозитория.)*

Ожидаемые нюансы (из архитектуры): чистота и эффекты (STM vs IO); привязка транзакций к OS-потокам; время жизни указателей в `mmap`.

Резюме карточки. Haskell-биндинг через FFI; данные требуют анализа репозитория. Ожидаемые нюансы: согласование чистоты и эффектов (STM vs IO), привязка транзакций к OS-потокам, время жизни указателей в mmap.

Чек-лист карточки.

- [] Разрешить чистоту/эффекты транзакций (STM vs IO).
- [] Учитывать привязку транзакций к OS-потокам.
- [] Ограничить время жизни указателей в mmap.
- [] Сверить данные с репозиторием биндинга.

8.12 Глава 45. Ruby

Обзор. Ruby-биндинг. *(Данные требуют анализа репозитория.)*

Ожидаемые нюансы (из архитектуры): GVL (глобальная блокировка VM Ruby — аналог Python GIL) и блокирующие вызовы; освобождение native-хендлов через finalizer/ensure.

Резюме карточки. Ruby-биндинг; данные требуют анализа репозитория. Ожидаемые нюансы: GVL (глобальная блокировка VM Ruby — аналог Python GIL) и блокирующие вызовы; освобождение native-хендлов через finalizer/ensure.

Чек-лист карточки.

- [] Учитывать GVL при длительных/блокирующих операциях.
- [] Освобождать native-хендлы через finalizer/ensure.
- [] Сверить данные с репозиторием биндинга.

8.13 Глава 46. Scala

Обзор. Scala-биндинг (через JVM). *(Данные требуют анализа репозитория.)*

Ожидаемые нюансы (из архитектуры): те же, что Java (JNI, pinning, хендлы), плюс особенности Scala-идиом (futures — аккуратно с потоками).

Резюме карточки. Scala-биндинг через JVM; данные требуют анализа репозитория. Ожидаемые нюансы — те же, что у Java (JNI, pinning, освобождение хендлов), плюс особенности Scala-идиом: futures требуют аккуратности с потоками.

Чек-лист карточки.

- [] Учитывать JNI-нюансы Java (pinning, порядок освобождения хендлов).
- [] В futures не рассчитывать на фиксированный поток.
- [] Сверить данные с репозиторием биндинга.

8.14 Итог тома

- Ядро биндингов (env/txn/CRUD/cursor) покрыто везде; продвинутое API — не всегда.
- Критические архитектурные следствия повторяются: NoStickyThreads (Go/async), Send/Sync и use-after-free (Rust), pinning/SafeHandle (Java/.NET), GIL (Python/Ruby), время жизни указателей в mmap (все).
- Карточки Nim/Java/Haskell/Ruby/Scala требуют анализа репозитория — помечены явно.

8.15 Заключение учебника (тома I–VI)

Вы прошли путь от «что такое key-value база» до «почему GC-цикл деградирует при долгих читателях» и «как использовать libmdbx из любого языка». Освежите понятия по мере необходимости, проверьте свои конфигурации по чек-листам Тома V, главы 30, и измеряйте — числа без контекста не имеют смысла.