**Zaxon Working Group**
Request for Discussion and Implementation Record

**ZDS 0010**
Engineering Discussion

# zxlite: A Native Python SDK for zaxonlite

**STATE**
committed

**CREATED**
2026-07-29

**LAST UPDATED**
2026-07-30

**INTENDED STATUS**
Implemented for 0.2.0

**AUTHORS**
paxos-zig project

**DISCUSSION**

CPython Stable ABI, Python-hosted backends, redundant cluster connections, concurrent write queueing, DB-API, PyPI, and SQLAlchemy

**LABELS**
zaxonlite, zxlite, python, db-api, sqlalchemy, pypi, sdk, write-queue

**Status of This Memo**

This document is an internal Zaxon discussion record authored in Typst and tracked in git. It intentionally follows RFC-style structure so the design scope, rationale, trade-offs, and operational constraints remain explicit. Documents that still use the placeholder number XXXXX are provisional drafts. Numbered ZDS documents are part of the project record.

## Contents

# 1. Abstract

zaxonlite exposes a small C ABI, but Python applications cannot install or use it as a normal Python database driver. The current query ABI serializes every non-null cell as text, write results omit generated row IDs, and the explicit transaction builder does not behave like a live SQLite transaction. Wrapping that surface with `ctypes` would produce a Python-shaped interface, but it would not produce faithful Python DB-API or SQLAlchemy behavior.

This record proposes a Python language SDK under `zaxonlite/languages/python`, a PyPI distribution and import namespace named `zxlite`, and a native `_zxlite` extension linked statically to Zig-built zaxonlite. The extension targets CPython 3.12's Limited API and Stable ABI, so one `cp312-abi3` wheel per operating-system and architecture pair can serve supported later CPython releases.

Delivery has three compatibility gates. The first provides a useful, typed, sqlite-shaped DB-API subset in replicated autocommit mode. The second adds a redundant remote connection over multiple cluster seeds, exactly-once write retry, and a bounded pool that can distribute explicitly stale-tolerant reads across healthy read-serving members. It also exposes the existing transport-owning embedded member as an explicit Python `Server` lifecycle, so tests and applications may host a backend in-process and connect to it through the same client path. The third adds a local embedded transaction surface with rollback, read-your-writes, savepoints, `lastrowid`, and `RETURNING`, then publishes a dedicated SQLAlchemy dialect. In every gate, concurrent server writes pass through zaxonlite's writer gate instead of competing for SQLite's writer lock, and Paxos orders each admitted write into one replicated commit. The project does not claim that API resemblance alone makes arbitrary `sqlite3` applications portable.

# 2. Introduction

Python's standard `sqlite3` module is a DB-API 2.0 driver with a broad behavioral contract. Programs rely not only on `connect()` and `execute()`, but also on transaction timing, cursor metadata, native value types, generated keys, exception classes, row factories, savepoints, and cleanup. SQLAlchemy relies on an overlapping but distinct set of driver hooks for reflection, pooling, isolation, generated identifiers, and nested transactions.

zaxonlite deliberately differs from a direct SQLite connection. Application writes execute inside a zaxonlite-owned outer transaction. On commit, zaxonlite captures SQLite WAL pages and replicates those page images through Paxos. Public SQL may not end that outer transaction, change capture-critical pragmas, attach another database, or access the reserved `__zaxon_` namespace.

The existing C ABI reflects those rules:

- `zaxonlite_exec_prepared` commits one prepared write as one replicated transaction and returns only its change count;
- `zaxonlite_query_prepared_json` executes a read-only statement and returns a materialized JSON object;
- the JSON query path converts every non-null SQLite value to text;
- `zaxonlite_transaction_*` accumulates copied statements and executes them only at commit;
- one native handle is single-threaded unless its caller provides a lock.

These are sound embedding primitives. They are not yet sufficient to implement the standard-library cursor contract or SQLAlchemy's unit-of-work behavior. The Python SDK therefore starts at the native boundary rather than papering over missing semantics in Python.

One server property anchors everything this record says about write concurrency. A node holds an exclusive directory lock against another node process, owns one live SQLite writer connection while leading, opens each replicated write with `BEGIN IMMEDIATE`, and admits one replicated write at a time behind `Server.runWrite`'s `writer_busy` condition. Contending server requests wait before entering that writer connection. This specifically prevents concurrent zxlite writers from manufacturing SQLite writer-lock contention. It does not make the broader `SqliteBusy` error category unreachable: the native layer still maps `SQLITE_BUSY` and `SQLITE_LOCKED` from SQLite maintenance, checkpoint, or unexpected engine paths.

## 3. Terminology and Scope

- **distribution**: the installable PyPI project, named `zxlite`
- **import package**: the Python package imported with `import zxlite`
- **native extension**: the private `_zxlite` CPython extension module
- **DB-API**: PEP 249's Python Database API 2.0 contract
- **sqlite-shaped**: names and ordinary usage resemble Python's `sqlite3` module without claiming complete substitutability
- **Limited API**: the subset of the CPython C API selected with `Py_LIMITED_API`
- **Stable ABI**: the cross-minor CPython binary ABI used by `abi3` wheels
- **local embedded connection**: one in-process zaxonlite node owning one data directory
- **embedded server**: one transport-owning zaxonlite `Embedded` facade that starts a member listener and routing client on a native background thread
- **server handle**: the Python lifecycle object owning one embedded server, its listener, background thread, data directory, and shutdown
- **autocommit operation**: one statement or explicit batch that becomes one complete replicated zaxonlite transaction before returning
- **live transaction**: a connection-scoped SQLite transaction in which later calls observe earlier uncommitted writes
- **DB-API transaction mode**: live transaction behavior exposed through `commit()`, `rollback()`, context managers, and savepoints
- **cluster client**: a network or transport-owning handle that may redirect operations to a leader
- **seed endpoint**: one configured `host:port` used to discover and contact a cluster; it is not permanently preferred over other healthy seeds
- **remote connection**: one logical DB-API connection backed by a pool of independent authenticated zaxonlite client connections
- **write lane**: the one serialized write path on a logical connection; it owns the connection's replicated session and assigns write sequences
- **write queue**: the ordered set of writes waiting for a write lane or for the server's writer gate
- **writer gate**: the server's node-global admission control that runs one replicated write at a time; contending writers wait, bounded by the server operation deadline, before entering the live SQLite writer
- **read level**: the server consistency contract: `any`, `leader`, or `linearizable`
- **read policy**: the client-side choice of an eligible physical connection, never a change to the requested read level
- **least-in-flight**: select the healthy eligible pool slot currently serving the fewest operations, with round-robin as the tie-break
- **materialized cursor**: a cursor whose complete result is copied into extension-owned or Python-owned memory before `execute()` returns

This record covers local embedded and redundant remote connections, embedded server lifecycle, additive C and client-RPC work, Python interfaces, packaging, binary wheels, SQLAlchemy integration, and release verification. It does not define an async driver, arbitrary Python SQLite callbacks, a distributed live transaction, a general process supervisor, or substitution of the standard-library `sqlite3` module.

# 4. Problem Statement

Seven gaps must be closed before zaxonlite can be a credible Python database package.

First, the C query ABI destroys SQLite type information. Python's default mapping is `NULL` to `None`, INTEGER to `int`, REAL to `float`, TEXT to `str`, and BLOB to `bytes`. JSON strings cannot distinguish integer `1`, text `"1"`, or the byte sequence containing `1`.

Second, write results do not expose `sqlite3_last_insert_rowid`, result-column metadata, or rows from `RETURNING`. ORMs commonly need a generated primary key immediately after an insert.

Third, the existing transaction builder is intentionally deferred. It copies statements without touching SQLite until commit. A query between two queued writes cannot observe those writes; a failed transaction cannot provide statement-local result rows; and a caller cannot use savepoints. Relabeling that builder as a DB-API transaction would be incorrect.

Fourth, the build installs only a static C library and header. PyPI needs self-contained, platform-tagged wheels, reproducible native builds, binary dependency inspection, and tests run against the installed wheel.

Fifth, SQLAlchemy does not support a new driver merely because that driver has methods named like `sqlite3`. Its SQLite dialect has pysqlite-specific connection behavior. zaxonlite needs its own dialect and acceptance suite.

Sixth, the existing reusable `client.ClusterConnection` accepts multiple endpoints, rotates through unreachable seeds, retains a connection to one answering server, and follows authenticated leader redirects. The C ABI does not expose that external client, and one `ClusterConnection` is intentionally single-call-at-a-time. A Python wrapper over one instance would provide failover but would serialize every Python thread and would not distribute `any` reads across cluster members.

Seventh, no public contract states how concurrent writes behave. `sqlite3` programmers defend against `database is locked` with busy timeouts and retry loops; programmers arriving from server databases such as PostgreSQL expect concurrent writes to queue silently and each to receive its own result. zaxonlite's server already waits contending replicated writers behind its node-global writer gate before they enter the live writer. That admission wait is unordered, its expiry shares one `timeout` error with the post-execution consensus wait whose outcome is genuinely unknown, and the local C handle fails fast with `WriteInFlight` instead of waiting. The native SQLite adapter still preserves `SqliteBusy` as a possible engine error. Without defined queue semantics, a Python application cannot distinguish "never executed, retry freely" from "fate unknown, replay by sequence".

# 5. Goals and Non-Goals

## 5.1. Goals

- Create an independent Python SDK project at `zaxonlite/languages/python`, inside the zaxonlite source tree, so language packages travel with the published zaxonlite repository.

- Support CPython 3.12 and later through the CPython Stable ABI.
- Ship self-contained wheels that need neither Zig nor a system zaxonlite installation at runtime.
- Preserve all five SQLite storage classes across the native boundary.
- Expose ZDS 0009′s validated lexical, vector, and hybrid search operation as a first-class `zxlite` API.
- Provide PEP 249 module globals, exceptions, connections, cursors, and fetch behavior for the supported surface.
- Keep zaxonlite replication and WAL-capture invariants authoritative.
- Release the GIL around blocking native operations.
- Enforce one-call-at-a-time access to a native handle.
- Make autocommit semantics explicit in the first release.
- Queue concurrent writes so concurrent zxlite writers do not surface SQLite's `database is locked`; preserve real SQLite busy errors from other engine paths as typed operational failures.
- Admit waiting writes in arrival order and bound every write-queue wait with a configurable timeout that raises a typed exception.
- Distinguish a write that was never admitted from a write whose fate is unknown, and retry only the latter through its replicated session and sequence.
- Accept redundant multi-server connection strings for remote clusters.
- Expose a synchronous Python `Server` lifecycle over the existing transport-owning C cluster facade.
- Support a single-node Unix-domain server on platforms where the native listener safely applies filesystem permissions.
- Support an explicitly development-only three-process cluster over numeric loopback with one shared PSK, matching the existing CLI transport policy.
- Reuse the existing leader-redirect and endpoint-rotation client behavior.
- Make remote connections safe to share across Python threads through a bounded pool of independent native client connections.
- Distribute only `level="any"` reads across healthy read-serving endpoints.
- Preserve `linearizable` as the default read level and never downgrade it during retry or load balancing.
- Use replicated sessions and sequences for exactly-once retry of remote writes whose response may be lost.
- Add genuine rollback and read-your-writes before enabling DB-API transaction mode.
- Provide a dedicated SQLAlchemy URL and dialect only after its behavioral release gates pass.
- Test installed wheels on every supported operating system and CPython minor.
- Keep the public Python layer type-annotated and documented.
- Preserve the existing C ABI by making all native changes additive.

## 5.2. Non-Goals

- No top-level module that shadows or replaces Python's `sqlite3`.
- No claim that every SQLite database file is a zaxonlite data directory.
- No direct access to zaxonlite's `current.db`.
- No `ATTACH`, runtime extension loading, or capture-changing pragmas.
- No arbitrary Python `create_function`, `create_aggregate`, or `create_window_function` in the initial releases.
- No Python callback executed from a Zig or SQLite worker thread.
- No faithful distributed transaction spanning separate Python calls across leader changes.
- No transparent distribution of `leader` or `linearizable` reads to followers or read replicas.
- No automatic consistency downgrade when the leader or quorum is unavailable.

- No asyncio API in the first release.
- No child-process manager, daemonization, port allocator, certificate authority, or production orchestration framework in the SDK.
- No embedding model, NumPy dependency, media decoder, or model inference in the Python package.
- No PyPy, GraalPy, 32-bit, musllinux, mobile, or WebAssembly wheel initially.
- No source-distribution promise until the monorepo source-staging design is reproducible and tested.

# 6. Invariants

1. A successful acknowledged write remains a committed zaxonlite operation, not merely a successful call to SQLite.
2. Python never obtains or modifies the materialized SQLite file directly.
3. The native layer preserves NULL, integer, real, text, and blob values without routing them through JSON.
4. Every owned native result, connection, transaction, and buffer has one explicit release path.
5. The Python extension does not expose a pointer whose lifetime can outlive its native owner.
6. One local connection serializes native calls even when the GIL is released. One remote logical connection may run calls concurrently only on distinct pool slots; each physical client connection remains single-call-at-a-time.
7. The module advertises `threadsafety = 2`: threads may share modules and connections, but not use one cursor concurrently. Local connections retain `check_same_thread=True` by default; remote pooled connections default to thread sharing.
8. The first release never pretends that a no-op `rollback()` can undo an already replicated write.
9. A live transaction is enabled only for a local embedded, single-member handle whose ownership and failure semantics are defined by this record.
10. Remote cluster connections remain autocommit-only. A transaction request on a remote DSN fails before sending SQL.
11. The extension is compiled against `Py_LIMITED_API=0x030C0000` and may use only symbols documented in the CPython 3.12 Limited API.
12. The wheel contains the exact zaxonlite, paxos, SQLite, and sqlite-vec versions named by its build provenance.
13. The import package and native library versions agree at import time or import fails.
14. SQLAlchemy compatibility is a tested dialect contract, never an inference from DB-API method names.
15. Python's typed search API delegates validation and SQL-plan construction to zaxonlite's ZDS 0009 implementation; it never reconstructs the canonical hybrid SQL with Python string interpolation.
16. A remote connection pins the database identity learned from its first authenticated seed and refuses results from an endpoint reporting a different database identity.
17. `linearizable` is the default for remote reads and search. Load balancing never changes a request's level.
18. A remote write is retried after an ambiguous transport failure only with the same replicated session and sequence.
19. Replicated server writes do not concurrently enter SQLite's writer connection. `Server.runWrite` admits one request, and `Node.writeRequest` opens that request with `BEGIN IMMEDIATE`. A `SqliteBusy` returned by another SQLite path remains a real typed error; it is never relabeled as writer-queue contention.

20. Writes waiting on one logical connection are admitted in arrival order through an ordered ticket wait, never an unordered lock scramble.
21. A write that leaves its queue by timeout either provably never executed, permitting a plain retry, or is treated as ambiguous and resolved only through its replicated session and sequence.
22. Starting a Python server and opening a DB-API client are separate operations with separate handles. `connect()` never starts a listener as a side effect.
23. Every member process receives the same ordered membership, roles, and cluster ID, a unique non-zero node ID and data directory, and an endpoint it alone owns.
24. Unix-domain transport is single-node only. Multi-member clusters use TCP; PSK-only TCP is accepted only with an explicit development option and numeric loopback endpoints.

# 7. Design Overview

## 7.1. Package architecture

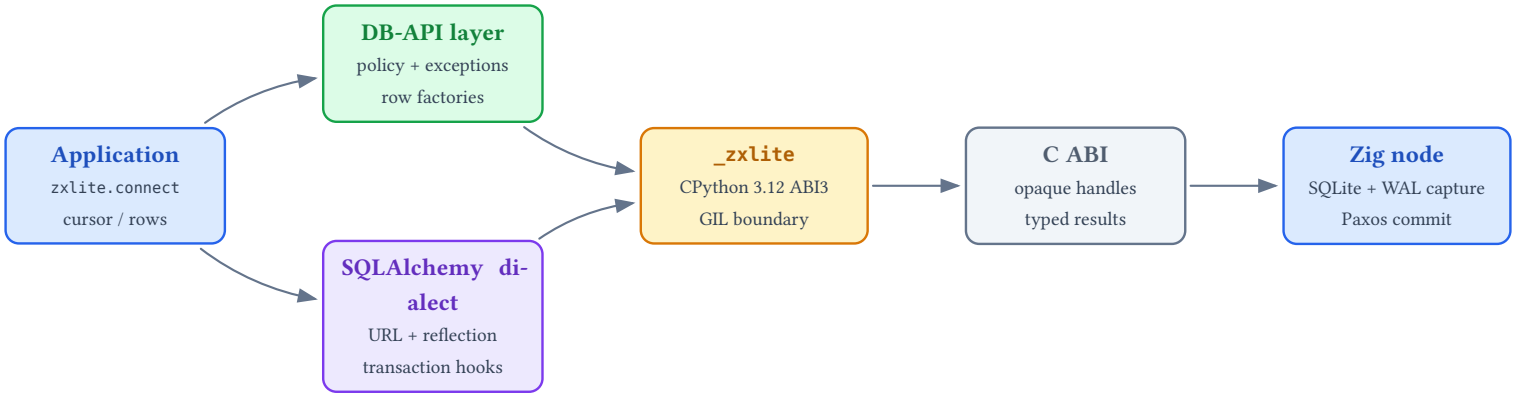


Figure 1: The public Python layers contain policy. The private extension is a narrow ownership, conversion, locking, and GIL boundary over the C ABI.

The distribution uses a `src` layout:

```
zaxonlite/languages/python/
  pyproject.toml
  uv.lock
  README.md
  LICENSE
  src/
    zxlite/
      __init__.py
      dbapi.py
      rows.py
      server.py
      sqlalchemy.py
      py.typed
    native/
      module.c
      connection.c
      result.c
      conversion.c
  tests/
    dbapi/
    sqlalchemy/
    packaging/
```

```
examples/
  unix_server.py
  loopback_cluster.py
```

`zxlite.__init__` re-exports the stable DB-API surface. `_zxlite` stays private and may change between SDK releases. `zxlite.sqlalchemy` imports SQLAlchemy only when the optional dialect is used; the base driver has no runtime dependency outside the Python standard library and its bundled native extension.

The location is load-bearing for release. The repository split publishes `zaxonlite/` verbatim as the public zaxonlite repository, so a package under `zaxonlite/languages/` ships there automatically; a directory at the monorepo root would be excluded. The package is never nested under `zaxonlite/src/cli_ui`, which a second split publishes separately, and `zaxonlite/build.zig.zon` deliberately keeps `languages` out of its `.paths` list so the Zig package hash consumed by embedders does not change when only SDK files change.

## 7.2. Project tooling and code style

The project is initiated and maintained with uv. `uv init --package` creates the `src` layout, `pyproject.toml` and the committed `uv.lock` are uv-managed, the development loop is `uv sync` and `uv run`, and `uv build` produces local artifacts. Release wheels are still built by `cibuildwheel`, which invokes the same PEP 517 backend, so the uv workflow and the release pipeline share one build definition.

Ruff is the single formatter and linter. `ruff format` and `ruff check` are configured in `pyproject.toml`, installed through a uv development dependency group, and enforced in CI: an unformatted or lint-dirty tree fails the build. The public Python surface follows the standard library's conventions — PEP 8 naming, the `sqlite3` module's idioms for the DB-API surface, imperative stdlib-style docstrings, and complete type annotations — so a `sqlite3` user reads `zxlite` code without relearning style. The C extension follows PEP 7.

## 7.3. Three compatibility gates

| Gate | User promise | Required native behavior |
|---|---|---|
| Gate A: typed autocommit | A useful sqlite-shaped DB-API subset for one-operation replicated writes and materialized reads. | Typed result handles, prepared positional parameters, changes, generated row ID, reliable exception mapping, and explicit autocommit-only policy. |
| Gate B: remote cluster | A redundant multi-seed connection with authenticated failover, exactly-once autocommit writes, concurrent consistency-aware reads, and an explicit Python-hosted backend lifecycle. | External client C ABI, typed client RPC, endpoint health, leader routing, replicated write sessions, first-in-first-out writer-gate admission, a queued-versus-ambiguous timeout discriminator, and a bounded pool of physical connections; wrap the existing cluster C facade and wire its missing Unix and development-PSK options. |
| Gate C: local transactions | Rollback, read-your-writes, savepoints, generated keys, `RETURNING`, and supported SQLAlchemy Core/ORM operation. | Connection-scoped local transaction, writer-connection reads, result rows before commit, safe WAL capture at commit, rollback without replication, and host-managed savepoints. |

Gate A is independently useful for scripts, services that already use one-statement transactions, migrations expressed as one atomic batch, and direct zaxonlite maintenance. It includes typed search because search uses the existing bounded read path and does not require a live transaction. Gate A is not marketed as general SQLAlchemy support.

Gate B is intentionally autocommit-only. It exposes the existing multi-endpoint reference client without claiming that remote calls form a live SQLite transaction. Gate C is intentionally local. It extends the embedded node so Python can hold a live transaction between calls without inventing distributed transaction semantics. Distributed live transactions require a separate ZDS.

# 8. Detailed Design

## 8.1. Additive typed C result ABI

The existing JSON query API remains supported. New functions return opaque materialized result handles:

```
typedef void zaxonlite_result;

typedef struct zaxonlite_exec_result {
    int64_t changes;
    int64_t last_insert_rowid;
    bool has_last_insert_rowid;
    bool replayed;
} zaxonlite_exec_result;

int zaxonlite_query_prepared_result(
    zaxonlite *handle,
    const char *sql,
    const zaxonlite_value *values,
    size_t value_count,
    zaxonlite_result **out_result);

int zaxonlite_exec_prepared_result(
    zaxonlite *handle,
    const char *sql,
    const zaxonlite_value *values,
    size_t value_count,
    zaxonlite_exec_result *out_result,
    zaxonlite_result **out_returning);

size_t zaxonlite_result_column_count(const zaxonlite_result *result);
size_t zaxonlite_result_row_count(const zaxonlite_result *result);
const char *zaxonlite_result_column_name(
    const zaxonlite_result *result,
    size_t column);
int zaxonlite_result_value(
    const zaxonlite_result *result,
    size_t row,
    size_t column,
    zaxonlite_value *out_value);
void zaxonlite_result_close(zaxonlite_result *result);
```

The exact C spelling may follow repository naming conventions, but the contract is fixed:

- a result owns copied column names and cell bytes;
- `zaxonlite_result_value` borrows text/blob bytes until result close;

- integer and real values preserve SQLite's runtime storage class;
- a zero-length text or blob is distinct from NULL;
- all count and index operations are bounds-checked;
- fallible functions clear output values before work;
- result close accepts null;
- `last_insert_rowid` is present only for a successful INSERT or REPLACE whose SQLite statement semantics update it;
- a DML statement with `RETURNING` returns typed rows and completes before the write is acknowledged.

The internal Zig `QueryResult` changes from nullable byte strings to a tagged value union. JSON encoding remains a presentation adapter over that typed form. Existing JSON consumers continue to receive strings and null unless a separate format-version decision changes their contract.

## 8.2. Native CPython extension

The C extension is a functional Limited-API shim: native handles travel as capsules with destructor backstops, every wrapper releases the GIL around its C ABI call, and ownership, locking, and the write lane live in the Python layer where they are testable. The extension does not read CPython object layouts or use private `_Py` symbols.

Local calls do not run on the calling thread. Each local connection owns one dedicated native worker thread with a 32 MiB stack, because node open and leader bootstrap need more stack than CPython threads carry; the worker also enforces one-call-at-a-time by construction. Every result is converted to Python objects and the native result closed before the call returns, so no pointer outlives its owner.

Each potentially blocking call follows this sequence:

1. validate and convert Python arguments while holding the GIL;
2. enter the connection's ordered lane (the write-queue contract) and hand the call to the connection's native worker;
3. release the GIL;
4. call the C ABI;
5. reacquire the GIL;
6. leave the lane;
7. convert the result or raise the mapped Python exception.

The remote path calls the native pool directly with the GIL released: slot reservation, the ticketed write lane, and cancellation state live inside the native pool. The close path marks the logical connection closing, refuses new reservations, waits for checked-out slots, and then destroys them. Finalization never calls into a handle already detached by explicit `close()`.

Python-to-SQLite binding is:

| Python input | SQLite value | Rule |
|---|---|---|
| `None` | NULL | No bytes pointer and zero length. |
| `bool`, `int` | INTEGER | Reject values outside signed 64-bit range with `OverflowError`. |
| `float` | REAL | Pass IEEE-754 binary64; SQLite decides subsequent SQL behavior. |

| Python input | SQLite value | Rule |
|---|---|---|
| `str` | TEXT | Encode UTF-8 once; embedded NUL is preserved by explicit length. |
| `bytes`, `bytearray`, contiguous `memoryview` | BLOB | Borrow a pinned contiguous buffer for the duration of the call. |
| other | none | Raise `ProgrammingError` until an explicit adapter API is added. |

SQLite-to-Python conversion creates `None`, arbitrary-precision Python integers from signed 64-bit values, Python floats, UTF-8 strings, and bytes. Invalid UTF-8 returned as SQLite TEXT raises `OperationalError`; it is not silently reclassified as BLOB. A later `text_factory` extension may offer different policy.

## 8.3. Public DB-API module

The base module exports:

```
apilevel = "2.0"
threadsafety = 2
paramstyle = "qmark"

def connect(
    target,
    *,
    timeout=5.0,
    isolation_level=None,
    check_same_thread=None,
    autocommit=True,
    tls_ca=None,
    tls_cert=None,
    tls_key=None,
    auth_file=None,
    allow_psk_only_loopback=False,
    read_level="linearizable",
    read_policy=None,
    freshness_ms=None,
    pool_size=None,
    connect_timeout_ms=None,
    operation_timeout_ms=None,
    expected_database_id=None,
): ...
```

Every remote option may arrive through the DSN or the keyword, never both: specifying one option in both places rejects the call before any network activity.

`target` is a path-like local zaxonlite data directory or a `zxlite://` connection string; `unix:/path/to/socket` names a served single-node backend on supported POSIX platforms. Gate A accepts only `isolation_level=None` and `autocommit=True`. Any other combination raises `NotSupportedError` at connect time. Gate B adds remote cluster targets but remains autocommit-only. Gate C adds `isolation_level="DEFERRED"` for local targets while retaining explicit autocommit.

When `check_same_thread` is omitted, local connections use `True` and remote pooled connections use `False`. Setting it to `True` on a remote connection restores creator-thread enforcement. Setting

it to `False` on a local connection permits sharing but still serializes all calls through the one local native handle.

`timeout` keeps `sqlite3`'s name but bounds a queue, not a lock. In `sqlite3` it configures the busy handler that spins on SQLite's file lock. In `zxlite` it bounds how long a write waits for its turn on the connection's write lane while earlier writes drain. Expiry raises `OperationalError` with the stable category `write_queue_timeout`; the statement was never admitted, no session sequence was consumed, and an immediate retry is safe. No `zxlite` error message claims the database is locked merely because another zxlite write is ahead of it: that contention waits at the write lane and server writer gate. A genuine `SQLITE_BUSY` or `SQLITE_LOCKED` from a different native engine path remains an `OperationalError` with its real extended category.

The exception hierarchy is:

```
Exception
  Warning
  Error
    InterfaceError
    DatabaseError
      DataError
      OperationalError
      IntegrityError
      InternalError
      ProgrammingError
      NotSupportedError
```

Native return codes provide the first classification:

| Code | Python exception | Examples |
|---|---|---|
| 1 | `DatabaseError` subclass | SQLite constraint maps to `IntegrityError`; busy, interrupt, and other SQL errors — including syntax — map to `OperationalError`, matching `sqlite3`; session errors map to `OperationalError` with the `session` category; parameter and statement misuse arrives as code 2. |
| 2 | `ProgrammingError` | Closed object, bad argument, write on read path, unsupported transaction control, or parameter mismatch. |
| 3 | `DatabaseError` | Integrity verification failed; maintenance API may expose a structured report later. |
| 4 | `OperationalError` | Locked directory, storage failure, corrupt state, or unavailable node. |

The native ABI must expose stable extended error categories rather than requiring Python to parse human-readable error strings. The existing integer codes remain; an additive error-category accessor supplies SQL constraint, busy, interrupt, misuse, storage, integrity, and availability classification.

`Connection` provides:

- `cursor(factory=Cursor)`;
- shortcut `execute`, `executemany`, and `executescript`;

- zxlite-specific `query(sql, parameters=(), *, read_level=None, freshness_ms=None)` for a per-call remote consistency override;
- `commit`, `rollback`, `close`, and context-manager methods;
- `row_factory`, `total_changes`, and `in_transaction`;
- zaxonlite-specific `snapshot`, `backup`, `integrity_check`, `open_session`, `execute_idempotent`, `expire_sessions`, and `search`;
- remote-specific `resolve_pending()` returning the resolved change count and replay flag, and `status_json()` as the diagnostics passthrough;
- read-only `zaxonlite_version` and `sqlite_version`.

`Cursor` provides:

- `execute(sql, parameters=())`;
- `executemany(sql, iterable_of_parameters)`;
- `executescript(sql_script)`;
- `fetchone`, `fetchmany`, `fetchall`, and iteration;
- `close`, `setinputsizes`, and `setoutputsize`;
- `description`, `rowcount`, `lastrowid`, `arraysize`, `connection`, and `row_factory`.

`description` is one seven-tuple per result column with the name followed by six `None` values. Cursors are materialized in every mode: native execution finishes before `execute()` returns, and fetch methods traverse copied rows. This makes ownership simple and matches the current node query architecture. Query row, byte, and VM-step budgets must be configurable before untrusted network inputs are routed through the local SDK.

Gate A supports positional qmark parameters. `execute()` rejects more than one SQL statement; `executescript()` is the explicit multi-statement path. The implementation uses SQLite preparation metadata to detect a trailing statement rather than parsing SQL in Python.

Named dict binding ships with Gate A, ahead of the original Gate C schedule: the C ABI exposes each prepared parameter's SQLite name and index so `:name`, `@name`, and `$name` are resolved by SQLite itself. Python never rewrites SQL with a regular expression.

## 8.4. Hosting a backend from Python

DB-API connection ownership and server ownership stay deliberately separate. `zxlite.connect("/data/db")` opens the existing non-serving `Node` C handle. `zxlite.start_server(...)` opens the existing transport-owning `zaxonlite_cluster` C handle and returns a `Server` object:

```
from zxlite import Member, start_server

with start_server(
    directory="/tmp/example-node",
    node_id=1,
    members=[Member(1, "unix:/tmp/example.sock")],
) as server:
    with zxlite.connect(server.endpoint) as db:
        db.execute("create table item(id integer primary key, value text)")
```

`Server.close()` is idempotent at the Python layer, asks the member to stop, joins its native background thread, and releases the directory lock. `Server.endpoint`, `node_id`, `members`, and `closed` are read-only. `Server.call()` is not public: ordinary use must exercise the typed DB-API and search client path rather than depending on the legacy JSON RPC. Server open, close, and shutdown-join operations release the GIL.

This surface is grounded in the current implementation:

| Capability | Existing native evidence | SDK work |
|---|---|---|
| Transport-owning member | `Embedded.open` starts `server.serve` on a background thread; `zaxonlite_cluster_open` exposes it through the C ABI. | Wrap the opaque cluster handle in `Server` with Python ownership and GIL rules. |
| Multi-process TCP cluster | `Embedded.open` accepts one identical runtime member registry per process; the role-cluster test opens real TCP members and the cluster integration test spawns three `zaxon serve` processes. | Document the spawn-safe Python test pattern and validate identical registries, unique IDs, directories, and endpoints. |
| Unix-domain service | `client.Endpoint.parse`, `client.Connection`, and `server.serve` support `unix:<path>`; the CLI verifies owner-only mode and refuses peers. | Add `listen_unix` to `Embedded.OpenOptions` and the cluster C options, startup probe, and shutdown wake path. Current `Embedded.open` does not yet forward this field. |
| Loopback PSK development cluster | The server already has a loopback-only PSK option; CLI `--dev-psk` requires a secret and numeric loopback for the listener and every peer. | Forward that option through `Embedded` and versioned C options. Keep failpoint-only insecure TCP out of release Python APIs. |

The public definitions are intentionally small:

```
@dataclass(frozen=True)
class Member:
    id: int
    endpoint: str
    role: Literal[
        "data_voter", "witness", "standby", "read_replica", "gateway"
    ] = "data_voter"

def start_server(
    *,
    directory,
    node_id,
    members,
    cluster_id=None,
    auth_file=None,
    tls_ca=None,
    tls_cert=None,
    tls_key=None,
    startup_timeout=10.0,
    allow_psk_only_loopback=False,
) -> Server: ...
```

All member endpoints are copied before `start_server()` returns. For a TCP cluster every process receives the same member sequence and `cluster_id`; each receives its own `node_id`, directory, and node certificate when TLS is used. The SDK validates the shape before native startup, while zaxonlite remains authoritative for voter counts, campaigner presence, roles, database-identity derivation, transport policy, directory locking, and recovery.

Unix sockets are a local single-node transport, not a way to connect three Paxos members. They omit TLS and PSK because filesystem ownership and the socket mode are the authorization boundary. The server refuses a pre-existing socket path rather than deleting it and removes its own path on orderly shutdown. Windows rejects this mode until the native implementation can apply an equivalent owner-only DACL.

For a three-process test, the parent chooses three unused numeric loopback ports, creates one protected PSK provider file containing at least 32 bytes, and starts three fresh Python interpreters with the same three `Member` records, cluster ID, and `auth_file`. Each child calls `start_server(..., allow_psk_only_loopback=True)` and remains alive until directed to close. The parent connects with all three seeds, the same `auth_file`, and the same explicit development flag. Tests use `subprocess` or the `spawn` `multiprocessing` context; they do not `fork` a process after a native server thread or client pool exists.

`allow_psk_only_loopback` is rejected unless `auth_file` is supplied and every configured endpoint is numeric `127.0.0.1` or `::1`. It provides authentication and frame integrity but no confidentiality or per-node identity, so it is never a production transport. The SDK does not allocate ports, spawn children, kill processes, remove stale socket paths, or create a CA. Those remain explicit test-harness or deployment responsibilities.

## 8.5. Remote connection strings

The same `connect()` entry point distinguishes a local data directory from a remote cluster DSN:

```python
# Local embedded node.
db = zxlite.connect("/var/lib/example")

# Client connection to a Python-hosted Unix server.
db = zxlite.connect("unix:/run/zxlite/example.sock")

# Redundant remote cluster seeds.
db = zxlite.connect(
    "zxlite://db1.example:9901,db2.example:9901,db3.example:9901/"
    "?read_level=linearizable&pool_size=8",
    tls_ca="/run/secrets/cluster-ca.pem",
    tls_cert="/run/secrets/client.pem",
    tls_key="/run/secrets/client-key.pem",
)

# Three-process loopback test cluster; never a production transport.
db = zxlite.connect(
    "zxlite://127.0.0.1:9901,127.0.0.1:9902,127.0.0.1:9903/",
    auth_file="/tmp/zxlite-test/cluster.psk",
    allow_psk_only_loopback=True,
)
```

The remote grammar is:

```
remote-dsn       ::= authority-dsn | query-seed-dsn ;
authority-dsn    ::= "zxlite://" seed ( "," seed )* [ "/" ]
                     [ "?" remote-option ( "&" remote-option )* ] ;
query-seed-dsn   ::= "zxlite:///?" query-option
                     ( "&" query-option )* ;
query-option     ::= "seed=" encoded-endpoint | remote-option ;
seed             ::= dns-host ":" port
                   | ipv4-address ":" port
                   | "[" ipv6-address "]" ":" port ;
unix-endpoint    ::= "unix:" absolute-path ;
encoded-endpoint ::= percent-encoded ( seed | unix-endpoint ) ;
remote-option    ::= "read_level=" read-level
                   | "read_policy=" read-policy
                   | "freshness_ms=" unsigned-integer
                   | "pool_size=" unsigned-integer
                   | "connect_timeout_ms=" unsigned-integer
                   | "operation_timeout_ms=" unsigned-integer
                   | "expected_database_id=" hex-u128 ;
read-level       ::= "any" | "leader" | "linearizable" ;
read-policy      ::= "least_in_flight" | "round_robin" ;
```

The authority form is the preferred direct `zxlite.connect()` spelling. `unix:<absolute-path>` is also accepted directly. The repeated `seed=` form exists because SQLAlchemy's URL model represents repeated query keys without relying on a comma-separated host extension; the dialect reads it through `URL.normalized_query`. It can encode one Unix endpoint for a local served node. Unix endpoints cannot appear in the authority form, mix with TCP seeds, or name a multi-member registry. After percent decoding, a TCP form contains one to 36 unique seeds. A missing port, empty host, userinfo, fragment, path other than `/`, unknown option, duplicate singleton option, invalid percent encoding, or value outside its bound rejects the DSN before network activity. Only `seed` may repeat. IPv6 literals require brackets before percent encoding. Unix paths must be absolute, contain no NUL, and fit the platform's native socket-path limit.

The DSN never contains a password, private-key bytes, or secret. `auth_file` is a separate provider path and may be composed inside TLS in production. Production TCP requires all three `tls_ca`, `tls_cert`, and `tls_key` arguments. PSK-only TCP additionally requires `allow_psk_only_loopback=True` and passes the same native loopback checks as the server. Provider paths are copied into native configuration but excluded from `repr`, logs, exceptions, and pool status. Unauthenticated plaintext and the failpoint-gated `allow_insecure_test_tcp` remain unavailable from the published Python surface.

`read_level` defaults to `linearizable`. `freshness_ms` is accepted only with `read_level=any`. `pool_size` defaults to `min(32, max(4, 2 * seed_count))` and is bounded to `1..=64`. `connect_timeout_ms` defaults to 5000 and `operation_timeout_ms` to 10000.

`expected_database_id` is optional. When supplied, it must match the authenticated status probe from every endpoint. When omitted, the first successfully authenticated seed establishes the database identity for the logical connection. Every later physical connection runs a status probe before joining the pool and must report the pinned identity. The probe runs once per pool slot when it first connects; endpoint identity across later reconnects is bound by mutual TLS in production, and the development PSK mode is already confined to numeric loopback.

Opening succeeds when at least one seed authenticates, reports the expected database identity, and accepts a client RPC. The remaining slots connect lazily. Failure to reach a quorum does not make `connect()` lie: connection creation can succeed while a later write or linearizable read correctly fails because no leader or quorum is available.

## 8.6. External client C ABI and typed RPC

`zaxonlite_cluster_open` already starts an in-process cluster member and is the native basis of Python `Server`; it is not the right primitive for a Python DB-API connection to existing servers. Before wrapping it, the existing `Embedded` and C option paths gain two narrowly additive behaviors:

- when this member's address parses as `unix:<path>`, require a one-member non-gateway registry and forward the path to `ServeOptions.listen_unix`; startup probing and shutdown wake-up use `UnixAddress`. This replaces the current behavior, which silently leaves the socket path in the member's host field with port zero and fails later inside `server.serve`;
- add `allow_psk_only_loopback` beside the C ABI's existing `allow_insecure_test_tcp`, forward it to the server's existing field, and keep the insecure failpoint field private to native tests.

No new server engine is introduced. The implementation continues to use `Embedded.open`, `server.serve`, the facade's background thread, and `Embedded.close`. The public C struct has no size/version member, so fields must not be appended while retaining the old entry point: an already-compiled caller would pass a shorter object. Add `zaxonlite_cluster_options_v2` with a leading `struct_size` and `zaxonlite_cluster_open_v2`; retain the original function and layout unchanged. Version 2 adds an `auth_file_path` mutually exclusive with the legacy raw secret and the loopback-PSK flag. It loads the provider through the existing `configuration.loadSecret` path, including the native regular-file, symlink, size, minimum-length, and permission checks. The Python extension uses v2 and exposes only the provider path.

An additive external-client ABI wraps `client.ClusterConnection` without opening a data directory or listener:

```
typedef void zaxonlite_remote;

typedef struct zaxonlite_remote_options {
```

```
    const char *const *seeds;
    size_t seed_count;
    const char *tls_ca_path;
    const char *tls_cert_path;
    const char *tls_key_path;
    const char *auth_file_path;
    bool allow_psk_only_loopback;
    size_t pool_size;
    uint64_t connect_timeout_ms;
    uint64_t operation_timeout_ms;
    bool has_expected_database_id;
    uint8_t expected_database_id[16];
    uint64_t write_admission_timeout_ms;
} zaxonlite_remote_options;

int zaxonlite_remote_open(
    const zaxonlite_remote_options *options,
    zaxonlite_remote **out_remote);
void zaxonlite_remote_close(zaxonlite_remote *remote);
int zaxonlite_remote_search(
    zaxonlite_remote *remote,
    const zaxonlite_search_options *options,
    int level,
    uint64_t freshness_ms,
    zaxonlite_result **out_result);
```

The remote execute, query, and search functions use the same `zaxonlite_value`, typed result, search options, and structured write result defined for local connections. This requires an additive `typed-v1` client RPC representation:

- request parameters carry explicit null, integer, real, UTF-8 text, or base64 blob tags;
- the server binds them with the existing prepared-value path;
- response cells carry the same tags instead of converting every non-null value to a JSON string;
- legacy requests that omit `format:"typed-v1"` retain their existing JSON string/null response;
- row, byte, VM-step, SQL-length, embedding, and candidate limits remain server-enforced.

This is a client RPC format addition, not a Paxos, journal, snapshot, or WAL payload-format change. A server that does not advertise `typed-v1` causes remote `zxlite.connect()` to fail with a version error instead of silently losing value types.

Remote autocommit writes use a replicated session. The logical connection opens one session at the leader, serializes its writes, assigns monotonically increasing sequences, and retries the same request with the same session and sequence after a redirect or ambiguous connection loss. The replicated session result retains the change count today, and a fresh (non-replayed) write also reports the optional last inserted row ID; a replayed result carries only the change count until the session-table format extension lands, so callers must not rely on the row ID surviving replay. Remote `RETURNING` stays unsupported until its bounded typed result can also be retained and replayed.

Remote `executemany()` uses one bounded `typed-v1` batch request and one replicated transaction, with one parameter vector per statement execution. The batch has one session and sequence and is replayed as one operation. Python never implements remote atomicity by looping over rows or by issuing independent writes.

If the operation deadline expires while a write remains ambiguous, the connection retains the exact pending request and enters `write_pending`. Reads may continue. A different write raises `OperationalError` until `resolve_pending()` reconnects and retries the same session/sequence to a definitive replay or success response. The SDK never advances the sequence or substitutes a new SQL statement while a prior result is unresolved.

This retry guarantee is scoped to a live replicated session. An unknown, closed, or expired session is never recreated under the same or a replacement identity. If future ordered session expiry causes resolution to return `SessionExpired`, the SDK raises a distinct `OperationalError` carrying the session and sequence as diagnostic operation identity and marks the outcome unknown; it still does not execute SQL again. Explicit `close()` refuses while `write_pending` remains unresolved so normal application code cannot silently discard that state. Interpreter finalization may abandon the local object but does not send a session-close command and emits `ResourceWarning`.

## 8.7. Multi-threaded remote read pool

One `client.ClusterConnection` owns one socket and mutable retry state. It remains single-call-at-a-time. Concurrency comes from a bounded pool of independent instances, never concurrent mutation of one instance.

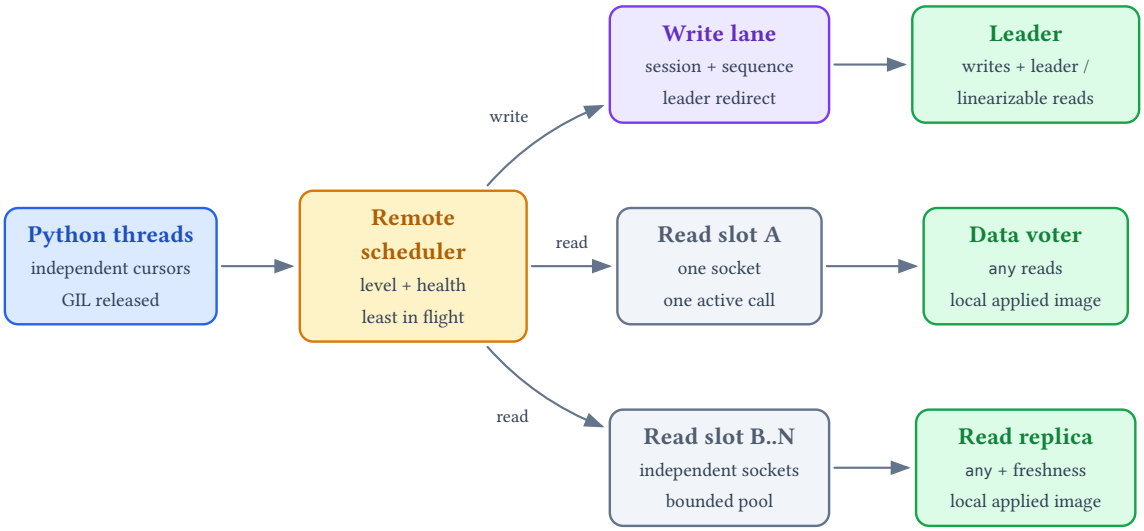


Figure 2: Concurrency uses independent physical clients. Consistency level, not thread count, determines which cluster members may answer.

The scheduler applies these rules:

| Read level | Eligible target | Scheduling and guarantee |
|---|---|---|
| `linearizable` | leader only | Default. Follow the authenticated leader hint and complete the quorum read fence. Multiple pool slots may overlap requests, but work is not distributed to followers. |
| `leader` | leader only | Follow the authenticated leader hint and read the leader's applied image without the linearizable fence. |
| `any` | healthy read-serving member | Use least-in-flight or round-robin across data voters, standbys, and read replicas. The response is local and may be stale; |

| Read level | Eligible target | Scheduling and guarantee |
|---|---|---|
| | | `freshness_ms` lets a learner reject a result outside the requested bound. |

The first authenticated `members` response seeds the topology cache. Only roles whose advertised capabilities serve reads are eligible for direct `any` routing; witnesses are never selected. Gateways remain valid configured seeds and can provide availability before topology discovery, but a successfully authenticated storage-member endpoint is preferred for explicit per-member distribution.

Topology is advisory and refreshed after configuration changes, repeated role errors, or a bounded interval. Every newly learned endpoint must be authenticated by the cluster CA, must present the advertised node ID, and must report the pinned database identity before it becomes eligible. Unauthenticated leader or topology advertisements never expand the seed set.

Each slot tracks consecutive connection failures, `stale` responses, current in-flight count, and a monotonic retry deadline. Transport failures trigger bounded exponential backoff with jitter; successful probes return a slot to service. Endpoints are not permanently evicted. A read may retry another eligible slot because the server enforces that the statement is read-only. Retries stop at the operation deadline and never change `read_level` or remove `freshness_ms`.

The first Gate B release implements this section in a staged form. The pool exists with identity pinning, typed-v1 enforcement, one ticketed write lane, and read rotation across configured seeds, but the topology cache, role-based eligibility, jittered per-endpoint backoff, and the client-side status surface are not built yet: an ineligible member rejects a routed read server-side and the client simply retries the next slot with a flat pause. Because every physical connection serves one call at a time, a slot's in-flight count is zero or one, so `least_in_flight` and `round_robin` coincide by construction today; both spellings are accepted and recorded. The full scheduler above is the contract this pool grows into before it is recommended for large clusters.

One cursor is never used concurrently. Separate cursors on one remote `Connection` may execute in parallel, and their materialized result ownership is independent. Local connections remain serialized because they own one node handle and, in Gate C, may own one live transaction.

## 8.8. Write queueing without database locks

Python's `sqlite3` exposes SQLite file-lock contention directly: a second writer hits `SQLITE_BUSY`, the driver raises `database is locked`, and every application invents its own busy timeout and retry loop. A server database such as PostgreSQL hides that contention: many clients submit writes, the server orders and executes them, and each client simply receives its result. zaxonlite is already on the server-database side of this divide, and the SDK must present it that way.

| System | Two concurrent writers | Application obligation |
|---|---|---|
| `sqlite3` on one file | The second writer receives `SQLITE_BUSY`; the driver raises `database is locked` after the busy timeout expires. | Configure busy timeouts and wrap writes in retry loops. |
| PostgreSQL | Both writes are accepted; the server orders and executes them and answers each client. | None for plain autocommit writes. |

| System | Two concurrent writers | Application obligation |
|---|---|---|
| zxlite | Both writes are accepted; the connection write lane and the server writer gate queue them in order, Paxos commits each in sequence, and each caller receives its own typed result. | None for plain autocommit writes; bounded waits surface as typed timeouts, never lock errors. |

Three layers make this true, and none of them is an SQLite lock:

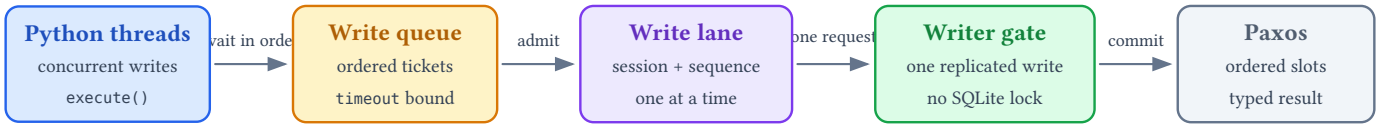


Figure 3: Concurrent replicated writes queue through ordered admission instead of competing for SQLite's writer lock. Other SQLite busy failures retain their real category.

First, the SDK queues. Every logical connection has one write lane. A thread that submits a write while the lane is busy waits in arrival order on an ordered ticket, not on a bare mutex, so a sustained stream of writers cannot starve one caller. The wait is bounded by the connection `timeout`; expiry raises `OperationalError` with the stable category `write_queue_timeout`. A write rejected at this layer was never admitted: it consumed no session sequence and never reached SQL, so an immediate plain retry is safe.

Second, the server queues. The node admits one replicated write at a time behind `Server.runWrite`; a contending connection blocks on `writer_cond` and is answered when its turn completes, bounded by the server operation deadline, currently ten seconds. The admitted request then reaches the leader's one live writer and `Node.writeRequest` opens `BEGIN IMMEDIATE`. The gate wait is a host condition wait, not an SQLite busy-handler spin. Read leases are separate; while an admitted write awaits its consensus decision the condition wait releases the server mutex and reads can continue. An unrelated SQLite maintenance or engine failure may still return `SqliteBusy` and is reported honestly.

Third, Paxos orders. Each admitted write commits speculatively in SQLite, its captured WAL frames become one transaction payload, and the payload is appended as the next replicated slot. Because payload N+1 is a physical WAL-frame delta over payload N's applied image, a dependent payload cannot be built before its predecessor is chosen. The consensus core can pipeline independent slots, but replicated SQLite transactions are never independent, so zaxonlite runs one replicated write at a time by design. Grouping several queued transactions into one slot — ZDS 0002's bounded writer queue and group commit — is server-side throughput work that changes no contract in this record; the SDK treats it as transparent.

Two consequences must be stated honestly. `pool_size` scales reads only: every write on one logical connection travels the single write lane, and the server runs one replicated write at a time regardless of how many sockets a client opens. And graceful queueing is admission control, not parallelism: under sustained write pressure, latency grows with queue depth until waits reach their bounds and surface as typed timeouts.

Today the server reports one `timeout` error for three different waits: the writer-gate wait before any SQL executes, the epoch-rollover wait, and the post-execution consensus wait whose outcome is genuinely unknown. Gate B therefore requires two additive server behaviors. The writer gate becomes a first-in-first-out admission queue, delivering ZDS 0002's bounded writer queue, so pooled clients and separate processes cannot be starved past the deadline by luckier waiters. And

the timeout error gains a discriminator stating whether the statement was never admitted — safe to retry plainly — or had been admitted, in which case the SDK applies the existing ambiguous-write rule and resolves only through the same replicated session and sequence. Until a contacted server advertises the discriminator, the SDK conservatively treats every remote write timeout as ambiguous — and the first release stays conservative even when the discriminator is advertised, because an expired deadline can span attempts with mixed outcomes; resolution through the session replay path is always safe.

A local connection follows the same shape with shorter plumbing. All calls on one local handle serialize through the connection's ordered lane, and `timeout` bounds a write's wait for that lane. The native `WriteInFlight` fail-fast remains a misuse detector for foreign embedders; a correct SDK build never triggers it, and no code path converts internal contention into an SQLite busy error.

## 8.9. Typed search API

Raw search SQL works through ordinary `execute()` from Gate A onward. The bundled SQLite build registers FTS5, sqlite-vec, `rrf`, `dbsf`, `stddev_samp`, and `zaxon_vec_distance_cosine` on every connection. Python TEXT and BLOB parameters preserve the query text and embedding bytes, and the typed result ABI preserves item IDs, scores, and metadata values.

Raw SQL alone is not sufficient for the enforced search contract from `ZDS 0009`. In particular, a host cannot reliably inspect the vec0 `k` constraint inside arbitrary SQL. The native Python package therefore also exposes the existing typed `Node.search` path through an additive C request:

```
typedef enum zaxonlite_search_fusion {
    ZAXONLITE_SEARCH_RRF = 0,
    ZAXONLITE_SEARCH_DBSF = 1
} zaxonlite_search_fusion;

typedef struct zaxonlite_search_options {
    const char *fts_table;
    const char *vec_table;
    const void *text;
    size_t text_length;
    const void *embedding;
    size_t embedding_length;
    uint32_t k;
    uint32_t candidate_count;
    bool has_candidate_count;
    zaxonlite_search_fusion fusion;
    double text_weight;
    double vector_weight;
    const char *metadata_table;
    const char *metadata_id_column;
    const char *const *metadata_columns;
    size_t metadata_column_count;
} zaxonlite_search_options;

int zaxonlite_search(
    zaxonlite *handle,
    const zaxonlite_search_options *options,
    zaxonlite_result **out_result);
```

Optional strings use null pointers. Text uses a pointer and explicit length so an FTS query is not constrained by C-string termination. Embeddings are raw little-endian float32 bytes. All pointer/count pairs follow the C ABI's existing safe-empty-output and early-validation rules.

The C adapter borrows request memory for the call, converts the request to `search_api.Request`, and calls `Node.search`. The Zig implementation remains the single authority for:

- table and column identifier validation;
- rejection of `__zaxon_` and `sqlite_` namespaces;
- lexical-only, vector-only, and hybrid branch selection;
- `k` and `candidate_count` validation in `1..=4096`;
- the default `min(max(8k, 64), 4096)` candidate rule;
- nonnegative finite fusion weights;
- nonempty little-endian float32 embeddings whose dimension is divisible by eight;
- RRF or DBSF canonical SQL planning;
- a maximum of 16 selected metadata columns;
- omission of implicit embedding BLOBs from results.

The public Python method is:

```
cursor = connection.search(
    *,
    fts_table=None,
    vec_table=None,
    text=None,
    embedding=None,
    k=10,
    candidate_count=None,
    fusion="rrf",
    text_weight=1.0,
    vector_weight=1.0,
    metadata_table=None,
    metadata_id_column=None,
    metadata_columns=(),
    read_level=None,
    freshness_ms=None,
)
```

It returns a normal materialized `Cursor`, so `description`, row factories, iteration, and all fetch methods behave exactly as for `execute()`. Lexical-only search requires `fts_table` and `text`. Vector-only search requires `vec_table` and `embedding`. Hybrid search supplies both branches. The Gate B typed remote search RPC sends the request to the server's `search` operation with `format:"typed-v1"`. The server applies the same native planner and returns typed result cells through the read pool. Omitted read options inherit the DSN defaults; explicit options apply to this call only. A local connection rejects non-null `read_level` or `freshness_ms`.

`embedding` accepts `bytes`, `bytearray`, or a contiguous `memoryview`. Bytes-like input is interpreted as raw little-endian float32. The base package does not depend on NumPy; NumPy callers pass `numpy.asarray(vector, dtype="<f4").tobytes()`. A typed memoryview with native float format is accepted only on little-endian hosts and is borrowed for the duration of the native call. Big-endian hosts fail closed, matching ZDS 0009's first vector-format contract.

Invalid Python option types raise `ProgrammingError`. Validly typed requests rejected by zaxonlite's search validator map to `ProgrammingError` with a stable native search error category. SQLite execution failures map through the normal database exception hierarchy.

SQLAlchemy applications have two supported paths:

- compose the documented search SQL with SQLAlchemy `text()` and bound parameters when the query must participate in a larger SQL expression;
- obtain the dialect's DB-API driver connection and call its typed `search()` method when the enforced candidate cap and canonical planner are required.

The first SQLAlchemy release does not invent a custom ORM query language. A future SQLAlchemy executable search construct may be added only if it can retain native validation rather than duplicating the ZDS 0009 planner.

## 8.10. Gate A autocommit behavior

Every write `execute()` invokes one complete zaxonlite write transaction. Every `executemany()` builds one bounded native transaction batch and commits the batch atomically. `executescript()` uses the native SQL-batch write path and inherits its size and statement limits.

`Connection.commit()` is a no-op only because the connection was explicitly opened with `autocommit=True`. `Connection.rollback()` is also a no-op in that mode, matching the fact that no transaction is open. The SDK never accepts a non-autocommit configuration and then silently performs these no-ops.

`Cursor.rowcount` is the change count for completed DML and `-1` for reads or statements whose category is not known to the driver. `lastrowid` changes only after a qualifying successful `execute()`; `executemany()` and `executescript()` leave it unchanged. A read cursor has `description` even when it has zero rows.

SQL text containing application `BEGIN`, `COMMIT`, `ROLLBACK`, `SAVEPOINT`, or `RELEASE` continues to be rejected by the zaxonlite guard. Transaction control is a connection operation, never an escape hatch around WAL capture.

## 8.11. Gate C local live transactions

Gate C introduces a native local-transaction handle. It is restricted to a single embedded node because that node cannot lose leadership to another voter while a Python caller holds a transaction.

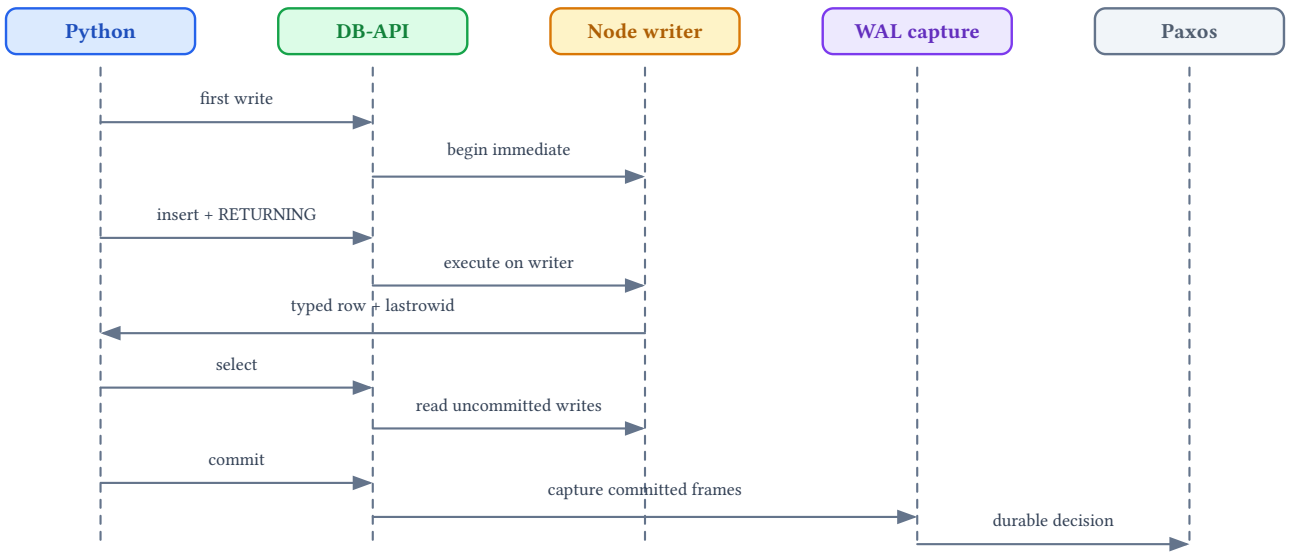


Figure 4: A local live transaction returns statement results immediately, but acknowledges commit only after the captured transition is decided.

The native implementation refactors the current write path into these states:

```
idle -> sqlite-open -> executing -> committing -> idle
                       |    |
                       |    +-> rolling-back -> idle
                       +-> failed -> rolling-back -> idle
```

On first transactional write, the node:

1. verifies the single-member capability and epoch capacity;
2. opens or reuses the guarded writer connection;
3. verifies no other captured write is in flight;
4. records the current WAL frame and total-change baselines;
5. executes `BEGIN IMMEDIATE` in internal guard scope;
6. returns a transaction handle bound to the node and connection.

Application statements execute under application guard scope. Read statements prepare and step on the same writer connection so they observe uncommitted writes. Their rows are copied before returning. Write statements may return rows. The host records changes and last row ID after each completed statement.

Savepoints are host operations. SQLAlchemy dialect hooks and DB-API transaction control call native savepoint functions that execute the necessary SQL under internal scope after validating an SDK-generated name. Arbitrary application transaction-control SQL remains denied.

Commit:

1. performs the batch marker and internal metadata work;
2. commits SQLite;
3. verifies the WAL hook and capture contract;
4. reads the frames since the recorded baseline;
5. builds one bounded payload and appends its descriptor through Paxos;
6. acknowledges only when the local single-member slot is committed and applied;
7. marks the node for resync if SQLite committed but durable logging failed.

Rollback executes SQLite rollback, discards the frame baseline, clears transaction state, and publishes no payload. Closing a connection with an open transaction attempts rollback. A rollback failure makes the handle unavailable and prevents reuse.

While a live transaction exists, no other operation may use that node handle. The Python mutex serializes calls from the owning connection; native state also rejects misuse so non-Python embedders cannot violate the rule.

## 8.12. SQLAlchemy dialect

Gate C registers:

```
sqlalchemy.dialects
  zxlite = zxlite.sqlalchemy:ZxLiteDialect
```

Local connection URLs distinguish relative and absolute data directories:

```
create_engine("zxlite:///relative/data-directory")
create_engine("zxlite:////var/lib/zxlite/data-directory")
```

A remote SQLAlchemy URL uses repeated, percent-encoded `seed` values so it round-trips through SQLAlchemy's standard URL parser:

```
create_engine(
    "zxlite:///?seed=db1.example%3A9901&seed=db2.example%3A9901"
    "&seed=db3.example%3A9901&read_level=any&freshness_ms=250&pool_size=8",
    connect_args={
        "tls_ca": "/run/secrets/cluster-ca.pem",
        "tls_cert": "/run/secrets/client.pem",
        "tls_key": "/run/secrets/client-key.pem",
    },
    isolation_level="AUTOCOMMIT",
)
```

The dialect reuses SQLAlchemy's SQLite SQL compiler, type compiler, identifier preparer, and schema reflection queries. It overrides:

- DB-API module loading and connect argument translation;
- database path interpretation;
- begin, commit, rollback, savepoint, rollback-to, and release hooks;
- supported isolation and autocommit values;
- server and SQLite version discovery;
- `RETURNING`, generated-key, and sane-rowcount capability flags;
- connection-pool defaults appropriate to a directory-locked embedded node;
- pysqlite-specific `on_connect` handlers, including automatic Python UDF registration.

The default pool is `NullPool`: one checked-out DB-API connection owns the zaxonlite directory lock, and closing it releases that ownership. A `QueuePool` is allowed only when configured so the process maintains one underlying connection per data directory. The dialect detects a second open as `OperationalError`, not as a separate SQLite connection to the same file.

For a remote DSN the dialect defaults to `StaticPool`: one thread-safe logical `zxlite.Connection` already owns its bounded native socket pool, so a second SQLAlchemy pool would multiply sockets without improving scheduling. Remote SQLAlchemy requires `isolation_level="AUTOCOMMIT"` and supports Core autocommit operations. ORM unit-of-work rollback and nested transactions are supported only by Gate C local connections.

An optional dependency extra installs the tested SQLAlchemy range:

```
zxlite[sqlalchemy]
```

Importing the base DB-API package never imports SQLAlchemy. The first dialect release supports the maintained SQLAlchemy 2.0 and 2.1 lines. SQLAlchemy 1.x is out of scope.

## 8.13. Unsupported sqlite3 facilities

| sqlite3 facility | SDK status | Reason |
|---|---|---|
| `create_function`, aggregates, collations | unsupported | zaxonlite opens multiple internal SQLite connections; Python callbacks would need registration, lifetime, GIL, and replication rules on every connection. |
| `enable_load_extension`, `load_extension` | unsupported | Runtime extensions are compiled out as a product security boundary. |
| `ATTACH` and `DETACH` | unsupported | Attached database WAL pages are outside the replicated image. |
| `backup` to another connection | replaced | `Connection.backup(path)` calls zaxonlite's consistent logical backup API; it does not accept another DB-API connection. |
| `serialize`, `deserialize`, `blobopen` | unsupported | They expose or mutate materialized SQLite state outside the replicated transaction path. |

| sqlite3 facility | SDK status | Reason |
|---|---|---|
| `set_authorizer`, progress/trace callbacks | unsupported | zaxonlite owns the authorizer and query-budget policy. |
| `interrupt` | deferred | Needs a safe cross-thread native interrupt handle and lifecycle design. |
| `text_factory`, adapters, converters | deferred | The first contract uses SQLite's five native types without implicit conversion registries. |

The package README contains a compatibility matrix organized by API and behavior, not a blanket percentage. Unsupported methods exist only where the DB-API requires them; otherwise attribute absence is preferred to a method that fails late.

## 8.14. Build and wheel production

`zaxonlite/languages/python/pyproject.toml` is uv-managed and uses setuptools as its PEP 517 backend with a focused `build_ext` subclass:

1. locate the zaxonlite package root and resolve its `paxos` dependency, which `build.zig.zon` declares at the parent path: a monorepo checkout satisfies it directly, and a standalone zaxonlite checkout uses the same sibling layout as the published repository's CI;
2. invoke the pinned Zig 0.16.0 build for the static zaxonlite C library with `-Doptimize=ReleaseSafe -Dtls=true`;
3. consume the exact `libsqlite3` archive installed by that same Zig product graph, never an archive guessed from cache timestamps;
4. compile the CPython shim with `Py_LIMITED_API=0x030C0000`;
5. link the shim, zaxonlite, pinned SQLite, sqlite-vec, pinned OpenSSL 3, the platform C runtime, and required platform system libraries into `_zxlite`;
6. mark the extension `py_limited_api=True`;
7. copy it under `src/zxlite` with the platform's `abi3` extension suffix.

TLS is enabled because Gate B is a production remote client and zaxonlite requires mutual TLS for production TCP. Release builds pin OpenSSL 3, link or bundle it according to the platform wheel policy, and record its exact version in provenance. The wheel dependency inspection fails on an unaccounted system OpenSSL. An OpenSSL security update triggers a new zxlite wheel release even when the Python API is unchanged.

The initial release matrix is:

| System | Architecture | Wheel platform | Runtime floor |
|---|---|---|---|
| Linux | x86_64, AArch64 | `manylinux2014` | glibc 2.17 |
| macOS | x86_64 | `macosx_10_15_x86_64` | macOS 10.15 |
| macOS | AArch64 | `macosx_11_0_arm64` | macOS 11 |
| Windows | AMD64 | `win_amd64` | Windows 10 1809 / Server 2019 |

Each artifact is tagged `cp312-abi3`. CI imports and exercises the same wheel on CPython 3.12, 3.13, and 3.14. A later CPython minor is added to the test matrix before the project claims it, even though the Stable ABI should permit loading the artifact.

`cibuildwheel` drives platform builds, wheel repair, and tests in fresh environments. Linux uses `auditwheel`, macOS uses `delocate`, and Windows uses `delvewheel` inspection. The release fails if

the wheel depends on an unexpected zaxonlite, SQLite, sqlite-vec, OpenSSL, Zig, or non-platform shared library.

The initial PyPI release is wheel-only. A source distribution is added only after a release staging tool can copy the exact `paxos`, `zaxonlite`, Python SDK, manifests, dependency pins, and licenses into an isolated source tree. The sdist must build without reaching outside its unpacked directory. Symlinks to monorepo parents and build-time downloads of mutable branches are rejected.

### 8.15. Versioning and provenance

The Python SDK version follows the zaxonlite product version while the native ABI reports its own version. Import checks both:

```
Python distribution version == Python package version
native zaxonlite major/minor  == SDK expected native major/minor
```

Patch differences are allowed only when the C ABI compatibility policy says they are compatible. Wheels record:

- Git commit and clean/dirty state;
- Zig version;
- zaxonlite and paxos versions;
- SQLite numeric version;
- sqlite-vec version and source pin;
- target triple, optimization mode, TLS-enabled flag, and OpenSSL version;
- CPython Limited API floor;
- wheel repair tool and version.

Release builds require a clean tagged commit. PyPI upload uses trusted publishing from a protected GitHub environment. The build job cannot upload; the publish job downloads immutable wheel artifacts, verifies hashes and provenance, and then publishes first to TestPyPI or PyPI as selected.

## 9. Failure Semantics

| Condition | Python result | Native consequence |
|---|---|---|
| data directory already open | `OperationalError` | No second connection is returned; the first owner is unchanged. |
| closed connection or cursor | `ProgrammingError` | No native call occurs. |
| wrong creator thread | `ProgrammingError` | No native call occurs unless `check_same_thread=False`. |
| unsupported parameter type | `ProgrammingError` | Statement does not execute. |
| integer outside signed 64-bit | `OverflowError` | Statement does not execute. |
| SQL constraint | `IntegrityError` | Current statement or transaction rolls back according to its gate. |
| busy, interrupt, or I/O error | `OperationalError` | Handle remains usable only when native error state permits it. |

| Condition | Python result | Native consequence |
|---|---|---|
| some remote seeds are down | connect or operation succeeds through another healthy seed | Failed endpoints enter bounded backoff and remain eligible for later probes. |
| all remote seeds are down | `OperationalError` after the configured deadline | No consistency downgrade or unbounded retry occurs. |
| remote endpoint has another database identity | `InterfaceError`; endpoint quarantined | The endpoint never enters the read pool or receives application SQL. |
| `any` read is stale on one replica | retry another eligible member | The same freshness bound and read level are retained. |
| linearizable read has no leader or quorum | `OperationalError` | The SDK never retries it as an `any` read. |
| remote write response is ambiguous | retry the same session and sequence | A deadline leaves the connection in `write_pending`; a different write is refused until resolution. |
| two writes contend on one connection | the second waits in arrival order; both succeed | SQLite is never contended; the server admits one replicated write at a time behind its writer gate. |
| write-queue wait exceeds `timeout` | `OperationalError` with category `write_queue_timeout` | The statement was never admitted; no sequence was consumed; a plain retry is safe. |
| remote write times out after admission | `OperationalError`; treated as ambiguous | Resolution replays the same session and sequence; the SDK never substitutes a different statement. |
| unix target on Windows | `NotSupportedError` | No native call occurs; the platform cannot apply an owner-only socket mode. |
| server socket path already exists | `OperationalError` | The server refuses the path and never deletes it; stale paths are an explicit operator responsibility. |
| dev PSK with a non-loopback endpoint | `ProgrammingError` before network activity | Native startup validation also rejects it; no listener or socket is created. |
| server startup probe timeout | `OperationalError` | The native thread is joined and the directory lock released before the error returns. |
| second `Server` handle on one directory | `OperationalError` | The first owner is unchanged; directory locking stays authoritative. |
| connect to an absent unix socket | `OperationalError` | No retry loop is implied; the caller owns server lifecycle ordering. |
| invalid UTF-8 TEXT result | `OperationalError` | Result is closed; no partially converted cursor is returned. |
| Gate A rollback request | no-op only in explicit autocommit | Every prior write is already replicated and remains committed. |
| Gate C close with open transaction | rollback then close | No payload is proposed; rollback failure poisons the handle. |

| Condition | Python result | Native consequence |
|---|---|---|
| SQLite commit succeeds, logging fails | `OperationalError` | Node marks image for resync and rejects unsafe reuse. |
| Python exception during row factory | propagate exception | Cursor retains or discards its materialized row according to documented fetch semantics; native result ownership remains balanced. |

## 10. Security Considerations

The native extension runs inside the Python process and has the same filesystem authority as that process. A wheel is executable code; release provenance, pinned inputs, protected publishing, and artifact inspection are part of the security boundary.

The package statically links the existing guarded SQLite build. Runtime extension loading stays omitted. Python cannot replace zaxonlite's authorizer, register an arbitrary callback, attach a database, or obtain `current.db`.

Production remote connections use mutual TLS. A topology or leader advertisement can add an endpoint only after the server certificate binds the advertised node ID under the configured cluster CA. A status probe then binds the endpoint to the logical connection's database ID. DNS, an unauthenticated JSON response, or possession of the shared PSK alone is insufficient to expand the trusted endpoint set.

A Unix-domain server's authorization boundary is the filesystem. The listener is created owner-only, a pre-existing path is refused rather than deleted, and the connection carries neither certificate nor PSK identity. Unix and PSK principals remain anonymous to the server, so privileged membership and enrollment operations are structurally unreachable over those transports; they require an authenticated mTLS identity.

The development PSK provides HMAC-based mutual authentication and frame integrity with replay rejection. It provides no confidentiality and no per-node identity, which is why native validation confines it to numeric loopback endpoints behind an explicit opt-in and why it never appears in a production deployment guide. Secret providers are validated as regular, non-symlink, owner-only files holding between 32 and 4096 bytes; the SDK passes provider paths through to that native validation and never loads secret bytes into Python objects.

All Python lengths, row/column indexes, parameter counts, and buffer sizes are validated before casting to C types. Contiguous buffers are pinned for the entire GIL-released native call. Text and blob output is copied before its native result closes.

The extension never parses native error messages to make a security decision. Stable native error categories drive exception mapping; the message is diagnostic text only.

The GIL is not a connection or pool-slot lock. Native calls release the GIL, so explicit synchronization, slot reservations, cancellation state, and close-state transitions prevent use-after-close, concurrent physical-client use, and result-owner races.

TestPyPI and PyPI publication use separate environments. The publishing job has no arbitrary pull-request execution and no long-lived API token. Every wheel includes the MIT license plus required notices for bundled third-party code.

# 11. Replication and Compatibility

The typed local result ABI changes no journal, WAL payload, snapshot, or consensus wire format. The `typed-v1` client RPC is additive and negotiated; legacy request and response bodies remain byte-for-byte compatible.

Gate C changes the duration of the writer transaction, not the committed artifact. A completed local transaction still produces one captured WAL transition and one transaction payload. Its protocol payload remains subject to the existing statement, input-byte, and maximum-payload bounds.

Holding a writer transaction longer can delay other work and rollover. The local Python connection owns the node handle, so this is a process-local availability tradeoff rather than a cross-node consistency change. Cluster transactions remain out of scope.

The Python API follows semantic versioning:

- removing a supported method, changing a default, or changing exception classification is a breaking change;
- adding an optional method or accepted input type is additive;
- expanding the wheel matrix is additive;
- changing a native dependency without changing the Python API is still release-noted because SQLite behavior may change.

`sqlite3` compatibility is stated feature by feature. A local program can usually port by changing `import sqlite3 as db` to `import zxlite as db` only when every used feature is marked supported and its database argument is changed from an SQLite file to a zaxonlite data directory. A remote program supplies a multi-seed DSN and accepts the explicitly documented autocommit-only transaction contract.

# 12. Operational Considerations

Opening a connection initializes a zaxonlite node and can perform recovery. It is heavier than opening a routine cursor and should be pooled at the application level only within the one-directory/one-owner rule.

Opening a remote connection authenticates at least one seed and initializes a bounded native socket pool. Applications and SQLAlchemy reuse that logical connection instead of creating one pool per thread. The status surface reports seed count, connected and in-flight slots, leader cache, topology generation, pinned database ID, endpoint backoff, read level, read policy, freshness bound, and unresolved write state. It never reports key paths or certificate contents.

Opening a `Server` is heavier than opening either connection kind: it acquires one data-directory lock, initializes recovery and transport state, starts a listener and background thread, and waits for a successful status probe. Applications retain the handle for the backend lifetime. A process must close all server and client handles before interpreter shutdown and must not carry them across `fork`. Unix-server shutdown removes only the socket path that this handle successfully bound; the SDK never removes a pre-existing path.

Materialized query results bound native lifetimes but can consume memory proportional to total result bytes. The SDK documents query budgets and encourages pagination. A streaming cursor requires a different read-lease lifetime and cancellation design and is deferred.

The package logs nothing by default. Native errors become exceptions. Applications opt into standard Python logging for SDK diagnostics. Secrets, SQL parameter values, TLS material, and full SQL text are not included in default error logs or build provenance.

Release telemetry is artifact-level only: build duration, wheel size, dependency inspection, and test results. The installed package performs no network telemetry.

The first wheels are expected to be substantially larger than a pure-Python driver because they contain SQLite, sqlite-vec, Paxos, zaxonlite, and the remote client's OpenSSL dependency. CI records stripped wheel and loaded-library sizes; size is reported, not hidden by downloading a native library after installation.

# 13. Verification and Acceptance

## 13.1. Native ABI tests

- Preserve NULL, signed integer limits, real values, UTF-8 text, embedded NUL, empty text, blobs, and empty blobs.
- Return column metadata for zero-row queries.
- Bounds-check every row and column accessor.
- Close results before and after partial inspection without leaks.
- Return changes, optional last row ID, replay state, and `RETURNING` rows.
- Convert typed lexical, vector, and hybrid search requests to `search_api.Request` without copying or reimplementing the SQL planner.
- Validate null pointer/length pairs, metadata column counts, fusion values, embedding lengths, and safe empty search outputs.
- Keep existing JSON query output and C smoke tests compatible.
- Exercise allocation failure and safe empty outputs.
- Verify stable error categories independently of message text.
- Run address, undefined-behavior, and leak sanitizers where the target toolchain supports them.

## 13.2. DB-API tests

- Import metadata, module globals, exception hierarchy, and version checks.
- Connection and cursor close idempotence, finalization, context managers, creator-thread checks, and serialized cross-thread use.
- Qmark binding for every supported Python type and parameter-count mismatch.
- Reject multi-statement `execute()` and direct transaction-control SQL.
- `description` on populated and empty results.
- `fetchone`, `fetchmany`, `fetchall`, iteration, arraysize, and exhaustion.
- Default tuple rows, `Row`, connection row factory, and cursor row factory.
- `rowcount`, `lastrowid`, total changes, and no stale metadata after errors.
- Atomic bounded `executemany` and `executescript`.
- Thirty-two threads writing through one shared connection: every write applies exactly once, admission follows arrival order, and no raised error reports `database is locked` merely because a zxlite writer was already ahead in the queue.
- Write-queue timeout: the `write_queue_timeout` category is raised, no session sequence is consumed, and an immediate plain retry succeeds.
- Snapshot, backup, integrity, sessions, idempotent replay, and recovery.
- Raw FTS5, sqlite-vec, RRF, and DBSF SQL with text and embedding bindings.
- Typed lexical-only, vector-only, RRF hybrid, and DBSF hybrid search.

- Default and explicit candidate counts, the 4,096 hard ceiling, invalid identifiers, invalid weights, malformed embeddings, and metadata projection.
- Search cursors with tuple rows, `Row`, custom row factories, zero results, stable item-ID ordering, and no implicit embedding BLOB.
- Compare each supported behavior against CPython 3.12 `sqlite3` using a table-driven compatibility oracle.

### 13.3. Gate B remote-cluster tests

- Parse DNS, IPv4, bracketed IPv6, multiple seeds, every option, and all malformed or duplicate DSN forms.
- Start a single-node backend through Python over a temporary Unix socket, connect through the typed client API, write, search, read, close, and prove owner-only socket mode and orderly path removal on every supported POSIX target.
- Prove Unix server startup rejects a gateway, any peer, a relative or overlong path, Windows, and a pre-existing socket path without deleting it.
- Spawn three fresh Python processes, each hosting one member with the same registry, cluster ID, and PSK provider over distinct numeric loopback endpoints; connect with all three seeds, elect, replicate, stop one member, continue with quorum, restart it, and verify catch-up.
- Reject development PSK without a protected provider, with a non-loopback listener or peer, when mixed with Unix transport, or when the explicit client/server opt-in differs. Prove the release Python API has no `allow_insecure_test_tcp` parameter.
- Reject duplicate node IDs or endpoints, different registries or cluster IDs, a missing campaigner, and reuse of one data directory by two server handles. Close remains idempotent and joins the native thread.
- Round-trip SQLAlchemy's repeated `seed=` URL values through `URL.normalized_query`, including percent-encoded IPv6, and reject mixed authority/query seed forms.
- Connect when the first seed is down, when only a gateway is initially reachable, and after a previously failed seed recovers.
- Fail at the deadline when all seeds are down without leaking sockets or native pool slots.
- Reject a CA-invalid certificate, advertised node-ID mismatch, database-ID mismatch, unauthenticated topology expansion, and a server without `typed-v1`.
- Round-trip every prepared parameter and result type through the typed RPC while retaining legacy string/null responses for legacy requests.
- Prove witnesses are never selected for reads and that stale topology is refreshed after a decided voter replacement or role change.
- Run parallel cursors from at least 32 Python threads; verify no physical client handles two calls concurrently and active calls never exceed `pool_size`.
- Verify least-in-flight and round-robin distribution across data voters and read replicas for `level="any"`.
- Verify `leader` and `linearizable` calls are answered only by the leader, including when follower capacity is idle.
- Verify `freshness_ms` survives every retry, a stale replica rotates to an eligible fresh member, and all-stale members return `OperationalError`.
- Kill a connection before a write, after the request reaches the leader, and before the response; prove one session/sequence changes the database at most once.
- Drive sustained concurrent writers from many logical connections and separate processes: first-in-first-out writer-gate admission, no starvation within the server deadline, every write applied exactly once, and no lock error surfaced.

- Distinguish a queued-timeout response from a fate-unknown timeout; only the latter enters `write_pending` and session replay.
- Verify reads continue to be answered while an admitted write awaits its consensus decision.
- Exercise `write_pending`, `resolve_pending`, leader failover, sequence replay, session expiry with an outcome-unknown error, refusal of a different unresolved write, and proof that no replacement session retries it.
- Verify explicit close refuses an unresolved pending write and finalization neither closes its server session nor executes the statement again.
- Close while reads are active, cancel blocked sockets, join checked-out slots, and reject new operations without use-after-close.
- Run thread and address sanitizers around the native scheduler, slot state, cancellation, topology refresh, and endpoint backoff code where supported.

## 13.4. Gate C transaction tests

- Implicit begin on the first write and explicit autocommit.
- Read-your-writes across insert, update, delete, and DDL.
- Generated row ID and `RETURNING` before commit.
- Commit durability across close and reopen.
- Rollback publishes no payload and restores prior query results.
- Savepoint, rollback-to-savepoint, release, and nested SQLAlchemy transaction.
- Failed statement followed by rollback.
- Connection close, object finalization, and interpreter shutdown with an open transaction.
- Payload-limit failure before commit.
- Crash at SQLite commit, payload sync, journal append, decision, and apply failpoints.
- Compile-time and runtime rejection on multi-member or cluster handles.

## 13.5. SQLAlchemy acceptance

Run SQLAlchemy 2.0 and 2.1 tests against the installed wheel:

- engine construction and disposal;
- metadata `create_all`, `drop_all`, and reflection;
- Core insert, update, delete, select, bound parameters, and compiled cache;
- integer primary-key autoincrement and `RETURNING`;
- ORM add, flush, refresh, commit, rollback, and identity-map refresh;
- unique and foreign-key integrity exception mapping;
- nested transactions and savepoints;
- transactional DDL behavior documented by the dialect;
- `NullPool` reconnect and invalid multi-open configuration;
- Alembic create/upgrade/downgrade smoke tests through an optional test dependency;
- explicit rejection of unsupported Python UDF and attachment features.
- raw search SQL through SQLAlchemy `text()` and typed search through the underlying `zxlite` DB-API connection.
- local URLs use `NullPool`; remote multi-seed URLs use one `StaticPool` logical connection and do not multiply native socket pools.
- remote URLs reject non-autocommit isolation, ORM rollback claims, and nested transactions before SQL reaches the cluster.
- repeated remote `seed=` query values survive SQLAlchemy URL parsing and produce exactly one native logical connection.

The documentation may say “SQLAlchemy supported” only after this suite passes on every release platform.

### 13.6. Wheel and release tests

- Enforce `ruff format --check` and `ruff check` cleanliness on the entire package before any build job runs.
- Build each `cp312-abi3` wheel from a clean tagged checkout.
- Install with no Zig compiler and no system zaxonlite.
- Test the same artifact on CPython 3.12, 3.13, and 3.14.
- Inspect imported symbols to confirm Stable-ABI-only CPython references.
- Inspect shared-library dependencies after wheel repair.
- Run open, write, query, rollback where available, close, reopen, and integrity checks from outside the source tree.
- Verify package/native version agreement and provenance.
- Verify licenses and third-party notices inside the wheel.
- Upload to TestPyPI, install by version from TestPyPI, rerun smoke tests, then permit the protected PyPI publication job.

## 14. Performance Gates

The native driver is not accepted solely because it is functionally correct. Benchmarks record:

- connection open and recovery time;
- prepared insert and select latency through Python versus the C ABI;
- conversion throughput for integers, reals, text, and blobs;
- materialized result memory per row and per byte;
- GIL release by demonstrating progress on another Python thread during a deliberately slow native operation;
- remote read throughput and p95/p99 latency at pool sizes 1, 2, 4, 8, 16, and 32 with matching Python worker counts;
- per-endpoint distribution, open sockets, reconnect rate, and TLS handshake cost under healthy, one-seed-down, leader-failover, and stale-replica cases;
- proof that linearizable throughput changes only through concurrent leader connections and never through follower routing;
- `executemany` throughput and payload size;
- write throughput and queue-wait p50/p95/p99 at 1, 2, 4, 8, 16, and 32 concurrent writer threads, demonstrating ordered queueing with zero lock errors;
- read latency measured while an admitted write awaits its quorum decision;
- Gate C commit and rollback latency;
- import time and loaded native image size.

The first release sets no claim that Python overhead is zero. It must show no per-row native ownership leak, no allocation proportional to query history, and no unexpected serialization beyond the documented local-handle, write-lane, and per-pool-slot rules. Results are stored as raw benchmark artifacts with machine, Python, Zig, and native dependency versions.

## 15. Delivery Plan

### 15.1. Phase 1: typed boundary

1. Change the internal query result to tagged SQLite values.
2. Add opaque typed result and structured write-result C APIs.

3. Add the typed search request C API over the existing `Node.search`.
4. Add stable native error categories and statement metadata.
5. Preserve and extend the C smoke suite and C ABI documentation.

## 15.2. Phase 2: Python autocommit SDK

1. Create `zaxonlite/languages/python` with uv-managed packaging, ruff formatting and lint gates, native extension, DB-API layer, typing, examples, and compatibility matrix.
2. Implement Limited-API connection and result types, GIL release, locking, conversion, and exception mapping.
3. Implement Gate A connections, cursors, fetch behavior, atomic `executemany`, typed search, maintenance, and exactly-once helpers.
4. Add wheel CI for Linux x86_64 first, then Linux AArch64, macOS x86_64 and AArch64, and Windows AMD64.
5. Publish a Gate A prerelease to TestPyPI and retain the explicit autocommit-only label.

## 15.3. Phase 3: redundant remote cluster

1. Add the versioned cluster-open C options, wire single-node Unix serving and loopback-only development PSK through `Embedded.open`, and implement Python `Member`, `Server`, and `start_server`.
2. Add the external-client C ABI over the existing endpoint parser, `ClusterConnection`, mTLS identity checks, redirects, and cancellation.
3. Add negotiated typed prepared-value/result RPC without changing legacy JSON responses or consensus formats.
4. Implement strict multi-seed/Unix target parsing, database-ID pinning, topology discovery, endpoint health, and bounded backoff.
5. Implement one replicated-session write lane with ordered write-queue admission and pending-write resolution.
6. Add first-in-first-out writer-gate admission and the queued-versus- ambiguous timeout discriminator to the server; prove starvation-freedom under sustained write contention.
7. Implement the bounded physical-client pool and consistency-aware read scheduler; keep linearizable as the default.
8. Pass the hosted-server, Unix, development-PSK, failover, multi-thread, identity, consistency, exactly-once, write-queueing, and no-regression gates before enabling remote DSNs in a wheel.

## 15.4. Phase 4: local transaction core

1. Refactor node write capture into begin, statement, query, savepoint, commit, and rollback states without weakening the application guard.
2. Expose the local transaction through an additive C ABI.
3. Add typed `RETURNING`, generated-key, named-parameter metadata, and multi-statement-tail detection.
4. Complete crash, resync, payload-limit, finalization, and one-member capability tests.
5. Enable Gate C DB-API transaction mode only after the native suite passes.

## 15.5. Phase 5: SQLAlchemy and PyPI release

1. Implement the `zxlite://` SQLAlchemy dialect with local `NullPool` and remote `StaticPool` defaults.
2. Run the SQLAlchemy 2.0/2.1 and Alembic acceptance suites on installed wheels.

3. Publish documentation that distinguishes supported sqlite3, DB-API, SQLAlchemy, and zaxonlite-specific behavior.
4. Produce signed provenance, TestPyPI verification, and protected PyPI publication.
5. Add an sdist only after isolated source staging passes its release gates.

# 16. Alternatives Considered

## 16.1. Pure Python `ctypes`

Rejected as the release architecture. It avoids CPython headers but performs conversion and ownership through a dynamic FFI layer, cannot use the Stable ABI wheel tag to describe the bundled native module, and makes GIL release and rich native types awkward. It remains useful as a development spike.

## 16.2. CFFI ABI mode

Rejected for the base package. It adds a runtime dependency and still needs a bundled shared library with platform-specific loading and repair. The narrow CPython extension has a smaller runtime surface.

## 16.3. Write the complete extension in Zig

Rejected for the first release. Zig can call the CPython Stable ABI, but a small C shim uses the Limited API headers as their primary supported form and keeps Python reference-counting conventions reviewable. Product logic remains in Zig.

## 16.4. One wheel per CPython minor

Rejected. The extension does not require CPython object-layout access. Targeting the 3.12 Limited API permits one `abi3` artifact per platform and reduces release and security-patching fan-out.

## 16.5. Shadow the standard-library `sqlite3`

Rejected. Installing a competing top-level `sqlite3` package creates import ambiguity, surprises transitive dependencies, and overstates compatibility. Applications import `zxlite` explicitly.

## 16.6. Use SQLAlchemy's stock pysqlite dialect

Rejected. That dialect applies pysqlite connection and transaction behavior, including optional Python UDF registration. A dedicated dialect can reuse SQL compilation while declaring zaxonlite's actual capabilities.

## 16.7. Claim transactions by buffering statements in Python

Rejected. Deferred statements cannot provide read-your-writes, immediate constraints, generated keys, `RETURNING`, or correct savepoints. The existing native transaction builder remains useful for atomic batches but is not a live DB-API transaction.

## 16.8. Autocommit forever

Rejected as the final SQLAlchemy path. It is a useful first gate, but ORM rollback and unit-of-work semantics require a real transaction boundary.

## 16.9. One locked remote client connection

Rejected as the remote implementation. It would reuse the current `ClusterConnection` directly and provide seed failover, but every Python thread would wait behind one socket. The bounded pool preserves each native client's single-call discipline while permitting independent reads.

### 16.10. Balance linearizable reads across followers

Rejected. A follower cannot complete the current linearizable read fence. Routing a linearizable request as `any` would be a silent consistency regression. Linearizable and leader reads may use concurrent physical connections, but they still execute on the leader.

### 16.11. Python-only endpoint and thread pool

Rejected as the primary scheduler. It would duplicate authenticated redirect, cancellation, health, and database-identity state in Python while the GIL is released around the actual socket operation. The native pool keeps socket ownership and retry state in one reviewed layer; Python exposes configuration and DB-API objects.

### 16.12. Treat ordinary concurrent writers as SQLite busy errors

Rejected. zaxonlite already owns SQLite's single writer behind its own admission gate, so translating an SDK or writer-gate wait into `database is locked` would manufacture the wrong cause. Re-exporting `sqlite3`'s contention contract would force every application to carry busy timeouts and retry loops for a condition the platform can queue and answer. This rejection does not suppress a genuine `SqliteBusy` produced elsewhere by SQLite.

### 16.13. Parallel write execution across pool slots

Rejected. Each replicated transaction payload is a physical WAL-frame delta over its predecessor's applied image, so a dependent payload cannot be built before its predecessor is chosen. The consensus core can pipeline independent slots, but replicated SQLite transactions are never independent. Client-side write parallelism therefore cannot raise write throughput; it would only move queueing into the server unordered. Grouping queued transactions into one slot per ZDS 0002 is the sanctioned throughput path and requires no SDK contract change.

### 16.14. Distributed live DB-API transactions

Deferred to a separate design. A leader can change while a Python application thinks between calls. Correct behavior needs leases, retry identity, abandoned-client cleanup, conflict policy, and availability tradeoffs that are not implied by local transaction support.

## 17. Resolved Design Decisions

| Question | Resolved answer | Constraint or gate |
|---|---|---|
| Q1: import name | `import zxlite`; never shadow `sqlite3`. | The PyPI distribution is `zxlite`; ownership is verified before publication. |
| Q2: Python floor | CPython 3.12 Limited API and `cp312-abi3`. | Test every claimed later CPython minor with the installed wheel. |
| Q3: first semantics | Typed, replicated autocommit DB-API subset. | Only `autocommit=True` and `isolation_level=None`; no general SQLAlchemy claim. |
| Q4: transaction scope | Live DB-API transactions are local embedded and single-member only. | Cluster transaction semantics require a separate ZDS. |
| Q5: SQLAlchemy | A dedicated `zxlite://` dialect after Gate C. | SQLAlchemy 2.0 and 2.1 Core/ORM acceptance on every release platform. |
| Q6: packaging | Static native extension, TLS enabled with pinned OpenSSL 3, wheel-only first release. | manylinux2014 x86_64/AArch64, macOS x86_64/AArch64, Windows AMD64. |
| Q7: result model | Opaque materialized typed results in the C ABI and Python cursors. | Streaming is deferred; query memory is bounded through documented budgets and pagination. |
| Q8: Python callbacks | No arbitrary UDF, aggregate, collation, authorizer, or trace callbacks. | Registration across internal connections and GIL/thread lifetime need a separate design. |

| Question | Resolved answer | Constraint or gate |
|---|---|---|
| Q9: search API | `Connection.search(...)` returns a normal materialized cursor and calls zaxonlite's typed planner; raw search SQL remains available. | The native validator owns identifiers, embedding shape, fusion weights, metadata projection, and the `candidate_count <= 4096` gate. The Python package adds no NumPy or model-runtime dependency. |
| Q10: remote DSN | `zxlite://host:port,host:port/` or repeated SQLAlchemy `seed=` values supply 1 to 36 redundant TCP seeds; `unix:/absolute/path` names one served local node. Credential paths remain separate arguments. | Open after one authenticated seed succeeds; pin its database ID and validate every later endpoint before use. |
| Q11: threaded reads | A bounded native pool uses least-in-flight or round-robin only for explicit `any` reads. | `threadsafety = 2`; distinct cursors may overlap. `leader` and `linearizable` remain leader-only, and the default is linearizable. |
| Q12: remote writes | One serialized write lane uses a replicated session and monotonic sequence numbers across redirects and reconnects. | An ambiguous deadline retains `write_pending`; no different write may advance the sequence until `resolve_pending()` completes. |
| Q13: write concurrency | Concurrent replicated writes queue in arrival order and do not report `database is locked` merely because another zxlite writer is ahead. | The server runs one replicated write at a time. FIFO writer-gate admission and the queued-versus-ambiguous timeout discriminator land with Gate B; genuine native `SqliteBusy` errors remain visible, and slot batching remains ZDS 0002 server work. |
| Q14: Python-hosted backend | `start_server()` returns a synchronous `Server` owning the existing transport facade; clients still connect separately through DB-API. | Unix is single-node and POSIX-only after additive embedded wiring. Multi-member tests use separate processes and explicit loopback-only PSK; the SDK is not a process supervisor. |

These decisions close the implementation choices for the three release gates. PyPI project-name ownership for `zxlite` is an external administrative prerequisite. The distribution name, import namespace, SQLAlchemy dialect name, and connection URL remain aligned as `zxlite`; a different fallback name requires this record to be updated before publication.

## 18. References

- PEP 249: Python Database API Specification v2.0
- Python 3.12 `sqlite3` documentation
- CPython Limited API and Stable ABI
- Python Packaging User Guide: Packaging binary extensions
- Python platform compatibility tags
- cibuildwheel documentation
- SQLAlchemy 2.0 SQLite dialect
- SQLAlchemy 2.1 SQLite dialect
- SQLAlchemy database URLs and repeated query parameters
- SQLite last inserted row ID
- SQLite `RETURNING`
- SQLite result codes: `SQLITE_BUSY`
- SQLite prepared-parameter metadata
- ZDS 0002, *Zaxonlite: Product and Delivery Plan*
- ZDS 0003, *Zaxonlite Security and Trust Plan*
- ZDS 0004, *Zaxonlite Format and Compatibility Contract*
- ZDS 0006, *Windows Durability and the Supported Platform Floor*
- ZDS 0009, *Multimodal Search in zaxonlite*