# Storage Through sqlx: SQLite for One Node, PostgreSQL for Clusters

STATE

committed

STATUS

Committed

TYPE

Architectural Specification

AUTHORS

Vikrant Rathore

CREATED

2026-10-15

UPDATED

2026-10-16

SUMMARY

Replaces Zaxonlite with storage through sqlx, an embedded SQLite file for a single node and one shared PostgreSQL database for a cluster, while storage stays optional and one storage thread owns all SQL off the request path.

DISCUSSION

No external discussion URL assigned

LABELS

storage, rust, cluster

**Record status**

The catalog defines the review process and the meaning of each state. This numbered record forms part of the catalog.

## Contents

1. Abstract ..... 1
2. Motivation and scope ..... 2
3. Specification ..... 2
4. Rationale and alternatives ..... 6
5. Security and operational considerations ..... 6
6. Backwards compatibility ..... 7
7. Acceptance criteria and implementation status ..... 7
8. Open questions ..... 7
9. Review history ..... 7
10. References ..... 9

## 1. Abstract

The Rust port removes Zaxonlite. Persistent state (dynamic policies, IP reputation, incident forensics and the console's tables) is stored through `sqlx`. A single node uses an embedded SQLite file in its data directory. A cluster points every node at one PostgreSQL database, whose availability is PostgreSQL's concern. Storage stays optional, as in SID 0005: without `--data-dir` or `--database-url` the daemon is the in-memory firewall of SID 0002, and a build without the storage features contains no external crate. The request path never waits on the database; one storage thread owns all SQL, as before.

## 2. Motivation and scope

SID 0005 chose Zaxonlite, a replicated SQLite built on Multi-Paxos, so that every node could read and write the same SQL state. It worked, but it made a firewall depend on a young consensus system: 48,000 lines of Zig, its own journal format (changed incompatibly in 0.7.0), OpenSSL for mutual TLS, and a shutdown path that could block on consensus. Sibuna stores little: a few hundred policy rows, a reputation table, and forensic incidents that operators also export to external systems. That state does not need a bespoke replicated engine. It needs a reliable SQL store, local for one node and shared for many.

This record covers the persistent layer of SID 0005 and the console storage of SID 0007. It does not change what is stored or when; it changes where and how.

## 3. Specification

### 3.1. Backends and selection

**Table 23** Storage configurations

| Configuration | Option | Store |
|---|---|---|
| In-memory (default without options) | none | nothing persists; policies come from `--policy-file` |
| Single node | `--data-dir <path>` | SQLite file `<path>/sibuna.db` in WAL mode with `synchronous=NORMAL` and a busy timeout |
| Cluster | `--database-url postgres://...` and `--cluster-node <id>` | one PostgreSQL database shared by every node |

`--cluster-node` keeps its meaning from the Zig daemon: it binds challenge keys to the issuing node and occupies the top bits of incident identifiers. With PostgreSQL it is required and must be unique per node. The Zaxonlite options `--cluster-listen`, `--cluster-peer`, `--cluster-secret-file` and the cluster TLS options are removed; giving one stops startup with a diagnostic that names this record. The PostgreSQL connection string may carry its own TLS parameters when the build enables the TLS feature.

### 3.2. The storage thread

One storage thread owns a single-threaded asynchronous runtime and the connection pool. Request threads never await: they push incident records into the bounded MPSC ring (1 slot of 512 per record, dropping and counting when full) and read the published engine. Every `--storage-poll-ms` (default 500) the thread runs one tick:

1. Drain up to 32 queued incidents and write them in one transaction.
2. Read the policy revision. When it changed, or a reputation entry expired, rebuild the engine from the policy file, the `policies` and `policy_inspection` tables and `ip_reputation`, and publish it.
3. Run the console's due work (SID 0007), when the console is enabled.

A rebuild reads the revision before and after loading; if a writer committed in between, the candidate is discarded and the old engine stays live, exactly as SID 0005 required.

Other callers, the daemon and the console, hand the thread jobs and wait for their answers. At most 32 jobs wait in the job queue, the capacity of the Zig console mailbox. A job submitted to a full queue is refused at once with `Busy`, and the console answers `503`; the caller never waits for room. The thread runs at most 16 jobs before it checks whether its tick is due, as the Zig storage thread ran at most 16 console requests per tick. A due tick therefore waits for at most 16 jobs and the one already running, however many callers keep the queue full, and a burst of console requests can neither grow memory nor starve incident batches and policy publication.

### 3.3. Schema and migrations

**Migrations.** The schema is written once per dialect and applied by numbered migrations compiled into the binary. The table `sibuna_schema(component, version, applied_at)` records every applied migration of every component. The core schema of this record is the component `core`; the console adds its own component with its own numbering, so a node without the console neither applies nor refuses the console's tables. A node refuses to start against a database whose version of a component is newer than it knows, with a diagnostic naming both versions. All pending migrations run in one transaction; under PostgreSQL a transaction-

scoped advisory lock serialises nodes that start together. The core schema is one baseline migration, version 1, that creates everything below. No Rust deployment predates it, so there is no older core schema to upgrade.

**Typing rules.** The schema states in SQL what the Rust types state in Rust, so the database refuses a row that the storage layer could not represent:

- Every SQLite table is `STRICT`. A value of the wrong storage class is refused instead of being kept as written.
- Timestamps are integers holding Unix seconds. Identifiers, counters and sequence numbers are integers.
- Raw bytes are `BLOB` on SQLite and `BYTEA` on PostgreSQL. `TEXT` holds only textual values: names, patterns, addresses, methods and categories.
- A column is `NOT NULL` exactly when its Rust field is not an `Option`.
- An enumerated column has a `CHECK` that lists exactly the names its Rust enum writes with `as_str` and reads with `parse`. The stored spelling is the lowercase name; a direct write of `'DENY'` is refused, because one spelling per value keeps equality queries exact.
- An integer whose Rust type is narrower than 64 bits, or whose meaning is bounded, has a `CHECK` with that range.

**Tables.** The SQLite text below is normative; the PostgreSQL text differs only as the table of dialect differences says.

```
CREATE TABLE sibuna_schema (
    component TEXT NOT NULL,
    version INTEGER NOT NULL CHECK (version >= 1),
    applied_at INTEGER NOT NULL CHECK (applied_at >= 0),
    PRIMARY KEY (component, version)) STRICT;

CREATE TABLE policy_revision (
    id INTEGER PRIMARY KEY CHECK (id = 1),
    revision INTEGER NOT NULL CHECK (revision >= 0)) STRICT;
INSERT INTO policy_revision (id, revision) VALUES (1, 0);

CREATE TABLE policies (
    id TEXT NOT NULL PRIMARY KEY,
    name TEXT NOT NULL,
    priority INTEGER NOT NULL DEFAULT 100,
    path_pattern TEXT,
    ua_pattern TEXT,
    action TEXT NOT NULL CHECK (action IN ('allow', 'deny', 'challenge', 'weigh')),
    difficulty INTEGER CHECK (difficulty BETWEEN 0 AND 4294967295),
    algorithm TEXT CHECK (algorithm IN ('hashcash', 'posw')),
    header_matchers TEXT CHECK (CASE WHEN json_valid(header_matchers)
        THEN json_type(header_matchers) = 'object' ELSE header_matchers IS NULL END),
    cidr_matchers TEXT CHECK (CASE WHEN json_valid(cidr_matchers)
        THEN json_type(cidr_matchers) = 'array' ELSE cidr_matchers IS NULL END),
    weight INTEGER NOT NULL DEFAULT 0 CHECK (weight BETWEEN -2147483648 AND 2147483647),
    limit_rate INTEGER CHECK (limit_rate BETWEEN 1 AND 1000000),
    limit_window_seconds INTEGER CHECK (limit_window_seconds BETWEEN 1 AND 86400),
    limit_ban_seconds INTEGER NOT NULL DEFAULT 0
        CHECK (limit_ban_seconds BETWEEN 0 AND 86400),
    enabled INTEGER NOT NULL DEFAULT 1 CHECK (enabled IN (0, 1)),
    created_at INTEGER NOT NULL CHECK (created_at >= 0),
    updated_at INTEGER NOT NULL CHECK (updated_at >= 0),
    CHECK ((limit_rate IS NULL) = (limit_window_seconds IS NULL)),
    CHECK (limit_rate IS NOT NULL OR limit_ban_seconds = 0)) STRICT;
CREATE INDEX policies_enabled ON policies (priority, name, id) WHERE enabled;

CREATE TABLE policy_inspection (
    id INTEGER PRIMARY KEY CHECK (id = 1),
    path_traversal TEXT NOT NULL CHECK (path_traversal IN ('disabled', 'audit', 'enforce')),
    sqli TEXT NOT NULL CHECK (sqli IN ('disabled', 'audit', 'enforce')),
    xss TEXT NOT NULL CHECK (xss IN ('disabled', 'audit', 'enforce')),
    rce TEXT NOT NULL CHECK (rce IN ('disabled', 'audit', 'enforce'))) STRICT;

CREATE TABLE ip_reputation (
    ip_or_cidr TEXT NOT NULL PRIMARY KEY,
    reputation_score INTEGER NOT NULL CHECK (reputation_score BETWEEN -100 AND 100),
    banned_until INTEGER CHECK (banned_until >= 0),
```

```
    trigger_rule TEXT,
    hits INTEGER NOT NULL DEFAULT 1 CHECK (hits >= 0),
    last_seen INTEGER NOT NULL CHECK (last_seen >= 0)) STRICT;
CREATE INDEX ip_reputation_decisive ON ip_reputation (banned_until)
    WHERE reputation_score <= -50 OR reputation_score >= 50;

CREATE TABLE incident_receipts (
    node_id INTEGER PRIMARY KEY CHECK (node_id BETWEEN 1 AND 8388607),
    next_sequence INTEGER NOT NULL CHECK (next_sequence BETWEEN 1 AND 1099511627776)) STRICT;

CREATE TABLE security_incidents (
    id INTEGER PRIMARY KEY,
    node_id INTEGER NOT NULL
        REFERENCES incident_receipts (node_id) DEFERRABLE INITIALLY DEFERRED,
    client_ip TEXT NOT NULL,
    user_agent BLOB NOT NULL CHECK (length(user_agent) <= 200),
    method TEXT NOT NULL,
    path BLOB NOT NULL CHECK (length(path) <= 512),
    violation_category TEXT NOT NULL,
    offending_payload BLOB NOT NULL CHECK (length(offending_payload) <= 512),
    campaign_id INTEGER NOT NULL,
    recorded_at INTEGER NOT NULL CHECK (recorded_at >= 0),
    embedding BLOB NOT NULL CHECK (length(embedding) = 256),
    CHECK ((id >> 40) = node_id AND (id & 1099511627775) >= 1)) STRICT;
CREATE INDEX security_incidents_recent ON security_incidents (recorded_at, id);
CREATE VIRTUAL TABLE incidents_fts USING fts5(path, offending_payload, tokenize = 'ascii');
```

For each of `policies`, `ip_reputation` and `policy_inspection` and each of `INSERT`, `UPDATE` and `DELETE`, an `AFTER` trigger runs `UPDATE policy_revision SET revision = revision + 1 WHERE id = 1`.

**Table 24** Dialect differences

| Concern | SQLite | PostgreSQL |
|---|---|---|
| column typing | `STRICT` tables | typed columns |
| 64-bit integers | `INTEGER` (`INTEGER PRIMARY KEY` for keys) | `BIGINT` |
| 32-bit integers (`weight`, `reputation_score`, the `limit_*` columns) | `INTEGER` with the range `CHECK` | `INTEGER` with the same `CHECK` |
| `policies.enabled` | `INTEGER` holding 0 or 1 | `BOOLEAN NOT NULL DEFAULT TRUE` |
| `policies.id` and `policies.name` | `TEXT` (`BINARY` collation) | `TEXT COLLATE "C"` |
| JSON matchers | `TEXT` checked with `json_valid` and `json_type` | `JSONB` checked with `jsonb_typeof`; queries read `CAST(... AS TEXT)` |
| byte strings | `BLOB`, bounded with `length` | `BYTEA`, bounded with `octet_length` |
| larger of two values | `MAX(a, b)` | `GREATEST(a, b)` |
| upsert | `ON CONFLICT ... DO UPDATE` | `ON CONFLICT ... DO UPDATE` |
| full-text search | FTS5 table `incidents_fts` with the `ascii` tokenizer, holding the canonical tokens of SID 0013 joined by spaces | column `search tsvector NOT NULL`, written by the storage layer from the canonical tokens of SID 0013, and a GIN index |
| revision triggers | nine row triggers, as above | one PL/pgSQL function `sibuna_policy_revision` and an `AFTER INSERT OR UPDATE OR DELETE OR TRUNCATE` statement trigger per table |
| change notification | polling only | polling, plus `NOTIFY sibuna_policy` so nodes tick early |

**Indexes.** Every query of the storage layer is served by a key or an index. The schema version is read by the key of `sibuna_schema`; the revision and the inspection override by their single-row keys; the enabled rules in evaluation order by `policies_enabled`; the decisive unexpired reputation by `ip_reputation_decisive`; bans and receipts by their keys; the most recent embeddings for the campaign

window by `security_incidents_recent`; one node's identifier range by the incident key; and searches by `incidents_fts` or the GIN index.

**Choices.** Each departure from the key/value and text columns of SID 0005 has one reason:

- `policy_revision` is a table of one row because the revision is one typed counter, and a key/value table cannot state its type or its presence.
- `incident_receipts` holds one row per node because a receipt is a typed pair (node, next sequence) whose ranges SQL can check, where a key named `incident_cursor_<node>` had to be parsed.
- Rule limits are the three columns `limit_rate`, `limit_window_seconds` and `limit_ban_seconds` because they are bounded integers whose ranges SQL states exactly, so an invalid quota is refused when it is written, not when a node rebuilds.
- Header and CIDR matchers stay JSON documents because they are short lists in the policy file's own syntax whose elements, patterns and prefixes, only the policy crate can validate, and child tables would add two more triggered tables and a join for at most four headers and eight prefixes.
- PostgreSQL stores the matchers as `JSONB` because it validates the document on input; it does not keep white space, key order or a repeated key, none of which carries meaning in a matcher object, although a repeated header name that the node would refuse is collapsed there to its last value.
- Addresses stay `TEXT` in both dialects because PostgreSQL's `inet` has no SQLite counterpart and would need a statement and a decoder per dialect, while the engine's CIDR parser remains the one validator of addresses.
- `policies.id` and `policies.name` use the `"C"` collation on PostgreSQL so that rules with equal priority are ordered by bytes on both backends.
- `security_incidents.node_id` references the node's receipt, deferred to commit, because the batch writes its incidents before it advances the receipt in the same transaction; every stored incident therefore belongs to a node with a receipt.
- `campaign_id` has no foreign key because a campaign is named by its first incident, which retention may delete while later members remain.
- `security_incidents.node_id` has no index of its own: identifiers carry the node in their top bits, so a node's rows are a key range, and only the deletion of a receipt, which Sibuna never performs, would scan.

**Rust rows.** Each table that the storage layer reads has one row struct with one decode function, and each enumerated column one Rust enum whose `as_str` binds and whose `parse` decodes: `policies` is `PolicyRow` (with `PolicyAction`, `ChallengeAlgorithm` and `RuleLimits`), `policy_inspection` is `InspectionRow` (with `InspectionMode`), `ip_reputation` is `ReputationRow`, `incident_receipts` is `IncidentReceipt` and `security_incidents` is `IncidentHit`. The enums mirror `Action`, `Algorithm` and `Mode` of the policy crate, and a test checks that every stored name parses there.

Every statement binds its values as parameters. The Zig layer built SQL text with quoted literals; parameters remove the quoting code and its failure modes.

### 3.4. Policy revision

Operators may write the `policies`, `ip_reputation` and `policy_inspection` tables directly. Triggers on insert, update and delete increment `policy_revision.revision` in the writer's transaction, so a direct write is seen by every node on its next tick, as in SID 0005. A direct write that breaks a constraint of the schema is refused by the database and never reaches a node. Under PostgreSQL the trigger also issues `NOTIFY sibuna_policy`; a node that receives it ticks at once, and polling remains the guarantee.

### 3.5. Incidents and campaigns

An incident identifier is `(node_id << 40) | sequence`. Each node keeps a durable receipt, its row of `incident_receipts`, whose `next_sequence` is the sequence number after its last committed batch. The receipt is advanced in the same transaction as the batch, and never moves backwards (`MAX` or `GREATEST` of the stored and the new value). Every insert of a batch ending before sequence `end` is guarded by `COALESCE((SELECT next_sequence FROM incident_receipts WHERE node_id = $node), 0) < $end`, so replaying a batch after an ambiguous failure writes nothing twice. At startup a node continues at the larger of its receipt and one past its largest stored sequence. These rules are unchanged from SID 0005.

Zaxonlite linked sqlite-vec and assigned campaigns with a nearest-neighbour query over every stored embedding. The Rust port stores the 64-dimension embedding of SID 0006 as 256 bytes (little-endian `f32`) in `security_incidents.embedding` and searches in the storage thread: it keeps the most recent 16,384 embeddings of all nodes in memory (4 MiB), refreshed incrementally from the database on each tick, and assigns an incident to the campaign of its nearest neighbour when the cosine distance is at most 0.35. The distance threshold, the embedding and the campaign identifiers are unchanged. Clustering against older incidents than the window is a deliberate omission: campaigns are recent by nature, and the bound keeps memory and work fixed.

A request carries bytes, not text, and an investigator needs the exact bytes of an attack. The columns `user_agent`, `path` and `offending_payload` therefore hold raw bytes (`BLOB` in SQLite, `BYTEA` in PostgreSQL), exactly as captured within the per-field bounds of the incident record (200, 512 and 512 bytes). The columns `client_ip`, `method` and `violation_category` stay text, with NUL bytes removed and invalid UTF-8 sequences replaced by U+FFFD. Full-text search indexes the canonical tokens of the path and the payload, which Rust computes from their bytes (SID 0013, “Search”). On SQLite the tokens, joined by single spaces, are kept in the FTS5 table `incidents_fts`, whose `ascii` tokenizer splits them exactly at the spaces; the table stores its own copy of the text instead of reading external content, because its external content would be the raw bytes; a row leaves the index with `DELETE FROM incidents_fts WHERE rowid = ...`. On PostgreSQL the storage layer writes the tokens as the `tsvector` column `search` (path tokens with weight A, payload tokens after them with weight B). A query is parsed in Rust and translated for each backend, so both return the same matches. Readers, the console among them, receive the raw bytes and choose how to display them.

### 3.6. Reputation and bans

`ip_reputation` keeps its columns and upsert rules: a score lies between -100 and 100, a honeypot hit sets the score to -100 and extends `banned_until` with the greater of the two values, and a rebuild loads the unexpired decisive entries (score at most -50 or at least 50) into the engine’s CIDR trie and schedules the next rebuild at the earliest expiry.

### 3.7. Console storage

The console’s tables of SID 0007 move unchanged in meaning. Where SID 0007 relied on SQLite triggers to enforce audit and integrity invariants (for example “a token update writes an audit row”, “a retention delete cannot remove an incident under investigation”), the Rust console performs the same statements explicitly inside one transaction in a single function per invariant, and tests assert the invariant against both backends. Triggers remain only for the policy revision, because operators may write those tables directly.

## 4. Rationale and alternatives

**Why sqlx.** It supports SQLite and PostgreSQL behind one API, is maintained, is widely used, and its SQLite driver bundles SQLite 3.51 with FTS5 compiled in. We use it without macros, without its migration framework and without TLS by default (`default-features = false` with `runtime-tokio`, `sqlite-bundled` and `postgres`), which keeps the build free of a database at compile time. The cost is about one hundred transitive crates (`tokio`, the `futures` family, the `url` and ICU stack, `libsqlite3-sys` and its C build). They are confined to `sibuna-storage` and to the storage thread, and both backends are cargo features.

**Why not keep Zaxonlite through its C ABI.** It would keep the cluster design of SID 0005 but would require the Zig toolchain to build the Rust daemon and keep the consensus failure modes we are removing.

**Why not a hand-written store.** The console’s queries (time windows, rankings, audit filters, full-text search) are naturally SQL. Writing a query engine is not Sibuna’s job.

**Why PostgreSQL for clusters.** Sibuna nodes need shared, consistent, durable state, and PostgreSQL provides exactly that with mature replication and failover (streaming replication, Patroni or a managed service). Sibuna’s write rate is small (batched incidents, rare policy changes), so one primary is not a bottleneck.

## 5. Security and operational considerations

The data plane does not depend on the database being reachable: if it is down, the last published engine keeps serving, incidents queue in the bounded ring and are dropped with a counter once it is full, and the storage thread retries with backoff. This matches the Zig daemon’s behaviour when a cluster lost quorum.

Connection strings are secrets: `--database-url-file` reads one from an owner-only file, and the banner never prints it. A cluster's consistency is now PostgreSQL's: a node that loses the database keeps its last policy until it reconnects.

## 6. Backwards compatibility

A Zaxonlite data directory is not read. Operators export policies (`sibuna console policies export`) from the Zig daemon and import them into the Rust daemon. Forensic history is not migrated automatically. The removed cluster options fail with a diagnostic rather than being ignored.

## 7. Acceptance criteria and implementation status

1. The storage tests of the Zig persistent layer pass against SQLite, and against PostgreSQL when `SIBUNA_TEST_DATABASE_URL` names a test database.
2. A three-node test with one local PostgreSQL shows a policy inserted on one node enforced by all three within one tick, a honeypot ban on one node enforced by the others, and incident identifiers that never collide.
3. Stopping PostgreSQL does not stop request service; restarting it resumes storage without a daemon restart.
4. The console's storage invariants hold on both backends.
5. On both backends the schema refuses rows outside the ranges and names of the Rust types, and a direct operator insert into `policies` advances the revision and reaches a running node.

Status: all five criteria are met by the tests of `sibuna-storage` and `sibuna-console-storage` on both backends. Committed on 2026-10-16; released in 1.0.0-beta.1. The release executables are built without PostgreSQL support; a cluster builds from source with `--features postgres-tls`.

## 8. Open questions

- Closed 2026-10-16: TLS to PostgreSQL is offered only behind the `postgres-tls` feature (see the review history, "transport security").

## 9. Review history

- 2026-10-15: drafted with the Rust port (SID 0010).
- 2026-10-15 (Rust port): the implementation in `sibuna-storage` differs from this specification as follows. The schema table is `sibuna_schema(component, version, applied_at)`: the `component` column lets the console add its own numbered migrations without making nodes that lack the console refuse the database. Incident text is stored as UTF-8 with NUL bytes dropped and invalid sequences replaced by U+FFFD, because PostgreSQL text holds neither; Zig stored raw bytes. PostgreSQL full-text search uses a generated `tsvector` with the `simple` configuration (no stemming, like the default FTS5 tokenizer) and `websearch_to_tsquery`, so a query string never fails to parse there, while SQLite keeps the FTS5 `MATCH` syntax. Under PostgreSQL a batch with honeypot bans locks the revision row first, so two nodes that ban the same addresses in different orders cannot deadlock, and the revision triggers also cover `TRUNCATE`. Startup retries only transient connection errors, for a bounded time. The console's evidence, head and country sidecars are not written yet; the console port adds them with its own migrations.
- 2026-10-15 (raw incident bytes): `user_agent`, `path` and `offending_payload` are stored as raw bytes again, as the Zig daemon stored them, because the UTF-8 conversion of the previous item lost forensic detail. Full-text search indexes a lossy UTF-8 rendering of the path and the payload kept beside the bytes: in `incidents_fts` on SQLite, which now holds its own text and is no longer an external-content table, because external content would hand FTS5 the bytes and a delete could not reproduce the indexed text; and in the text column `search_text` that feeds the generated `tsvector` on PostgreSQL. Readers receive the raw bytes. Core migration 2 upgrades existing databases in place.
- 2026-10-15 (typed schema): the core schema becomes one baseline migration, version 1, that folds the previous migrations 1 and 2, since no Rust deployment exists to upgrade; the numbered migrations and the `component` column stay for the console. The key/value table `sibuna_meta` is replaced by the single-row table `policy_revision` and the table `incident_receipts`; the receipt guard keeps its semantics. SQLite

tables are `STRICT`; enumerated and bounded columns have `CHECK` constraints that mirror the Rust types; columns are `NOT NULL` where the Rust field is not optional; rule limits become three typed columns instead of the JSON document `limit_config`; matchers stay JSON, validated on SQLite and stored as `JSONB` on PostgreSQL; incidents reference their node's receipt; and every query of the storage layer has a key or an index. The section on the schema now specifies all of it.

- 2026-10-15 (bounded job queue): the first Rust storage thread took jobs from an unbounded channel and ran every queued job before checking its tick, so a burst of console requests could grow memory without bound and delay incident batches and policy publication indefinitely. The job queue now holds at most 32 jobs and refuses further jobs at once with `Busy`, and the thread runs at most 16 jobs between two checks of its tick, restoring the bounds of the Zig mailbox and of its per-tick cap. The section on the storage thread now states both bounds. The mailbox's urgent and background priorities are not kept; jobs run in the order they were queued.
- 2026-10-15 (Rust port, console storage calls): the storage handle gains a non-blocking form of `with_database`. `Storage::submit(job)` queues a job and returns a `Ticket` at once, or refuses with `Busy` when 32 jobs wait; `Ticket::try_take` returns the result without waiting, `Ticket::wait_timeout` waits at most a given time, and `with_database` is `submit` followed by a wait. Dropping a ticket abandons only the reply: the job still runs, and its transaction commits or rolls back whole. Jobs still queued when the thread stops are dropped unrun, and their tickets report `Stopped`, as the Zig mailbox cancelled them. The console's HTTP handlers wait at most 10 seconds; its background tasks poll and hold at most 4 jobs together (SID 0007). Two hooks join `Hooks` for the console. `on_committed` hears about each incident batch once its transaction is confirmed committed, exactly once: a batch whose commit was not acknowledged stays pending and is announced when the retry confirms it, so the console publishes each incident to its live feed once. `on_snapshot` reads the page templates (`console_pages`) in a read transaction of its own during each rebuild, into the new field `PolicySnapshot::pages`; the revision is checked after it as before, and a failure keeps the old engine and retries on the next tick. The daemon registers both hooks only when the console is enabled.
- 2026-10-15 (Rust port, search and snapshots): this record now states the full-text index that `sibuna-storage` builds, which SID 0013 introduced ("Changes to the core storage", item 1): canonical tokens computed in Rust, held by `incidents_fts` with the `ascii` tokenizer on SQLite and by `search tsvector NOT NULL` on PostgreSQL, and queries parsed in Rust and translated, so both backends return the same matches. The `search_text` column, the generated `tsvector`, `websearch_to_tsquery` and the raw FTS5 `MATCH` syntax of the earlier items are gone. Zig passed the query to FTS5 `MATCH` unchanged. The schema text, the table of dialect differences and the section on incidents are corrected accordingly, and the section on incidents now also says how the text columns are converted. The console's evidence, head and country sidecars are written in the batch's transaction by the incident hook (SID 0013). A policy snapshot loads at most 10,000 enabled rules and 1,000,000 unexpired decisive reputation entries; a larger table fails the rebuild, the old engine stays live, and the next tick tries again. Zig skipped reputation rows it could not read; the typed schema cannot hold such rows.
- 2026-10-15 (Rust port, node ids): `--cluster-node` accepts 1 to 8,388,607, the range the schema checks. Zig accepted 0 to `u32::MAX`; node 0 then shared storage node 1 with another node, and a larger id failed only at the first database write.
- 2026-10-15 (Rust port, transport security): the URL chooses the protection of every PostgreSQL connection with libpq's `sslmode` and `sslrootcert`; `PGSSLMODE` and `PGSSLROOTCERT` apply when the URL names neither, and the default is `prefer`. TLS is sqlx's rustls client, built only with the `postgres-tls` feature of `sibuna-storage` and of the daemon. A build without that feature refuses `require`, `verify-ca` and `verify-full` at startup with `StorageError::TlsNotBuilt`, before any connection is made; it never falls back to plaintext. As in libpq, `require` with a root certificate verifies the certificate as `verify-ca` does. A refused certificate is not a transient failure, so startup stops at once instead of retrying. The daemon prints one startup line about the connection: a statement when it is encrypted and verified, and a warning with a hint otherwise. Production clusters use `sslmode=verify-full`. Two differences from libpq remain: `sslrootcert` adds the certificate authority to the system trust store rather than replacing it (`verify-full` still requires the host name to match), and `allow` never tries TLS, while `sslrootcert=system` and `~/.postgresql/root.crt` are not read.

## 10. References

- SID 0005: the Zaxonlite storage architecture this record replaces.
- SID 0007: the console and its storage.
- SID 0010: the Rust port.
- sqlx 0.9 documentation; SQLite FTS5; PostgreSQL text search and `NOTIFY`.

### *Lifecycle event*

```
2026-10-15: prediscussion -> discussion. Specification of the storage
replacement, drafted before the code
```

### *Lifecycle event*

```
2026-10-16: discussion -> accepted. Implemented and released in Sibuna
1.0.0-beta.1; reviewed by the maintainer on 2026-10-16.
```

### *Lifecycle event*

```
2026-10-16: accepted -> committed. Implemented and released in Sibuna
1.0.0-beta.1; reviewed by the maintainer on 2026-10-16. crates/storage tests
on SQLite and PostgreSQL (worker tests, tests/multi_node.rs,
tests/resilience.rs, worker/tests/constraints.rs), crates/console-storage
invariants on both backends; release v1.0.0-beta.1
```