Zaxon Working Group
Request for Discussion and Implementation Record

ZDS 0009
Engineering Discussion

Multimodal Search in zaxonlite

STATE
committed

CREATED
2026-07-28

LAST UPDATED
2026-07-29

INTENDED STATUS
Implemented

AUTHORS
paxos-zig project

DISCUSSION

FTS5, sqlite-vec, memory-mapped vector scans, RRF, and DBSF

LABELS
zaxonlite, sqlite, search, multimodal, vectors

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 . . . 2
2. Introduction . . . 3
3. Terminology and Scope . . . 3
4. Problem Statement . . . 3
5. Goals and Non-Goals . . . 4
   5.1. Goals . . . 4
   5.2. Non-Goals . . . 4
6. Invariants . . . 4
7. Design Overview . . . 5
   7.1. System architecture . . . 5
   7.2. Search pipeline . . . 5
8. Detailed Design . . . 6
   8.1. Module and package boundaries . . . 6
   8.2. Build and connection lifecycle . . . 7
   8.3. SQLite-managed memory mapping . . . 8
   8.4. SQLite-compatible EBNF function contract . . . 8

8.5. RRF contract 9
8.6. DBSF contract 10
8.7. Zig implementation 11
8.8. SIMD feasibility and CPU dispatch 13
8.9. Vector storage and query 15
8.10. Typed search operation 16
8.11. Hybrid query sequence 17
8.12. Ingestion and replication flow 17
8.13. Fusion selection flow 18
9. Failure Semantics 18
10. Security Considerations 18
11. Replication and Compatibility 19
12. Operational Considerations 19
13. Verification and Acceptance 20
13.1. Unit tests 20
13.2. SQLite integration tests 20
13.3. Replication tests 20
13.4. Benchmark fixtures 21
13.5. Performance gates 21
14. Delivery Plan 22
15. Alternatives Considered 22
15.1. Standalone search package in the first release 22
15.2. Raw float32 exhaustive scan 22
15.3. HNSW 23
15.4. DiskANN 23
15.5. Direct operating-system mmap 23
15.6. RRF or DBSF in application code 23
15.7. A custom hybrid-search SQL statement 23
16. Resolved Design Decisions 24
17. References 25

1. Abstract

zaxonlite currently replicates ordinary SQLite schemas and data but does not build SQLite with an explicit full-text-search contract, does not ship vector search, and has no native score-fusion functions.

This record proposes a SQL-first multimodal search layer. SQLite FTS5 supplies lexical retrieval. A statically linked and pinned sqlite-vec supplies vector storage and similarity search. Zig implements weighted reciprocal rank fusion (RRF), distribution-based score fusion (DBSF), and sample standard deviation as allocation-free SQLite functions.

The vector path deliberately avoids a separate graph index. sqlite-vec scans chunked bit vectors through SQLite, optionally using SQLite-managed memory-mapped reads. It retains a bounded coarse candidate set and reranks only those candidates against float32 vectors with a portable Zig SIMD kernel. This preserves the existing WAL-page replication model, reduces the coarse vector representation by 32 times, and keeps query heap memory independent of corpus size.

2. Introduction

Text, images, audio, and video can all be retrieved through embeddings, but a database does not become multimodal merely by accepting BLOBs. It needs:

- a lexical index for exact terms, identifiers, and phrases;
- vector indexes for semantic similarity within one embedding space;
- deterministic ways to combine incomparable result lists;
- storage and query behavior that remains inside the existing replication, durability, and security boundaries.

The current architecture already provides the most important property. Application writes execute once on the leader. zaxonlite captures the committed WAL pages and replicates page images, not SQL recipes. FTS5 and sqlite-vec virtual tables persist their state in ordinary SQLite shadow tables. Their pages can therefore travel in the same transaction payload as the application row that caused the index update.

No model runtime is placed inside SQLite. The application supplies embeddings for writes and queries. Large media remains in application-owned object or content-addressed storage; SQLite stores references, extracted text, metadata, and embeddings.

3. Terminology and Scope

- **item**: an application row representing a document, image, audio object, video, or chunk
- **modality**: the input domain of an embedding space, such as text or image
- **embedding space**: one model revision, dimension, normalization rule, and distance metric
- **lexical retriever**: an FTS5 query returning ranked item IDs
- **dense retriever**: an exact float32 vector query
- **coarse retriever**: a bit-vector Hamming scan used to select rerank candidates
- **fusion**: combining independently ranked retriever results
- **RRF**: reciprocal rank fusion, which depends only on rank
- **DBSF**: distribution-based score fusion, which normalizes each score distribution before summing
- **oversampling**: asking the coarse retriever for more than the final `k`
- **SQLite-managed mmap**: read-only page access requested through SQLite's `mmap_size` interface, never a direct mapping created by zaxonlite

This proposal covers storage and retrieval primitives. It does not define an embedding model, model server, media decoder, object store, or managed application schema. It does not introduce custom SQL statements. All examples are valid SQLite SQL composed from virtual tables, common table expressions, window functions, and ordinary scalar functions.

4. Problem Statement

Four gaps prevent zaxonlite from serving as a multimodal search substrate.

First, `zaxonlite/build.zig` compiles the pinned SQLite amalgamation without an explicit `SQLITE_ENABLE_FTS5` contract. The amalgamation may contain FTS5, but the product does not assert or test that capability.

Second, `SQLITE_OMIT_LOAD_EXTENSION` intentionally removes runtime extension loading. That is the correct security boundary, but it means sqlite-vec must be pinned, statically linked, and registered on every connection.

Third, the result scales differ. FTS5 `bm25()` and vector distance are both lower-is-better, but their magnitudes are unrelated. Adding raw values gives one retriever accidental control over the answer.

Fourth, a naive float32 scan reads four bytes per dimension per item. Loading an in-memory graph would exchange scan cost for persistent and resident graph overhead, and a sidecar index would fall outside the replicated SQLite image. The search path needs a small and reproducible memory footprint.

5. Goals and Non-Goals

5.1. Goals

- Make FTS5 and sqlite-vec capabilities explicit, pinned, and testable.
- Keep runtime extension loading compiled out.
- Register every search extension on every SQLite connection.
- Replicate base rows, FTS state, and vector state atomically as WAL pages.
- Provide weighted RRF and DBSF as Zig SQLite functions.
- Use SQL syntax accepted by the pinned SQLite parser.
- Keep vector query heap memory bounded by the candidate count, not item count.
- Reduce coarse scan bytes using one-bit quantization and chunked access.
- Use SQLite-managed mmap as a bounded performance profile.
- Preserve exact float32 reranking for final vector results.
- Use portable SIMD for full-vector distance where the target guarantees it.
- Retain a tested scalar fallback without illegal-instruction risk.
- Treat Apple Silicon, generic ARM, x86, wasm, and non-SIMD targets as first-class build configurations rather than assuming Intel or AMD.
- Define deterministic tie-breaking and error behavior.
- Work on leaders, followers, standbys, read replicas, and restored snapshots.

5.2. Non-Goals

- No model inference inside SQLite.
- No raw media fetching or decoding.
- No user-defined loadable extensions.
- No new network query language.
- No unpinned ANN index format.
- No promise of constant-time vector search.
- No automatic relevance tuning across applications.
- No replacement for application-level access control.

6. Invariants

1. The authoritative state remains the Paxos journal and transaction payloads.
2. Search indexes are materialized SQLite state and are reconstructible from snapshots plus committed payloads.
3. A media row and all of its index changes become visible in one committed transaction.
4. Followers never execute indexing SQL during apply; they place decided pages exactly as they do today.
5. Every process that opens an image registers the same FTS5, sqlite-vec, and Zig function versions before preparing application SQL.
6. Runtime `load_extension`, `readfile`, and `writefile` remain absent.

7. One vector table contains one fixed embedding dimension and metric.
8. Final ordering always has a stable item-ID tie-break.
9. Query memory has an explicit bound derived from `candidate_count`. The typed `search` operation is the enforced path: it validates the count against the 4096 ceiling before any SQL exists. For raw application SQL the `k = ?` value is consumed inside the vec0 virtual table where the host cannot observe it, so the ceiling is a documented contract there, backed by the server row, byte, and VM-step budgets; the VM-step budget under-counts virtual-table scan work, so rows, bytes, and the statement deadline are the operative raw-SQL bounds.
10. zaxonlite never directly memory-maps a database file behind SQLite.
11. Offline page apply never overlaps an open connection or live SQLite mmap.
12. A mixed binary whose search feature manifest is incompatible fails closed.

7. Design Overview

7.1. System architecture

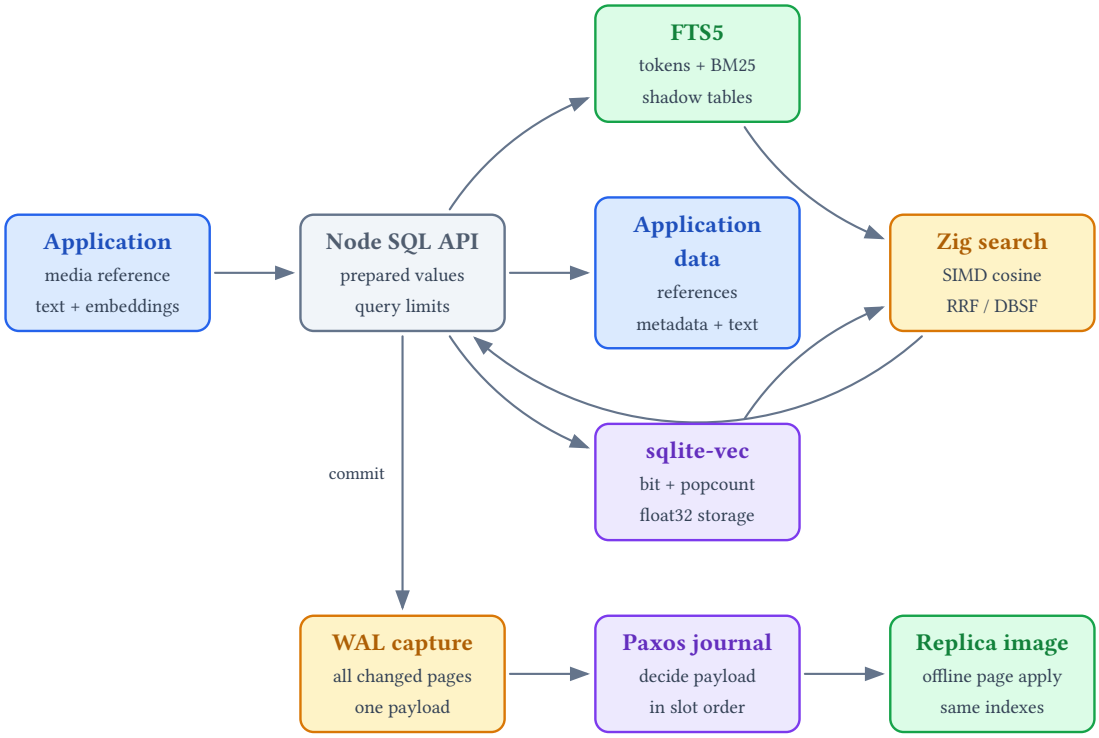

Figure 1: Multimodal indexes stay inside the replicated SQLite image.

The write path and read path deliberately meet at SQLite rather than at a second storage service. A leader executes virtual-table maintenance. The captured WAL carries the outcome. A replica needs sqlite-vec when it opens the materialized image for queries, but it does not need sqlite-vec to apply decided pages offline.

7.2. Search pipeline

The default vector algorithm is a two-stage exact-rerank pipeline.

1. Normalize the caller's float32 query vector.
2. Quantize it to one bit per dimension.
3. Scan sqlite-vec bit-vector chunks with Hamming distance.
4. Retain `candidate_count` coarse matches.
5. Read only those candidates' float32 vectors.
6. Compute full-precision cosine distance with the Zig SIMD kernel.

7. Keep the best final `k`.

For final `k`, the default candidate count is:

$$C = \min(\max(8k, 64), 4096)$$

Applications may request a different value, but the host and network query budgets impose the hard maximum of 4096. The oversampling factor is a quality/performance parameter, not part of the index format.

For a dimension `d`, float32 needs `4d` bytes per stored vector. The coarse bit representation needs `d / 8` bytes, a 32-times reduction. Full float32 vectors remain on disk for exact reranking, but the scan reads the compact representation and the exact stage touches only `C` vectors.

8. Detailed Design

8.1. Module and package boundaries

The implementation is divided by dependency direction, not merely by file size. Numeric search code must remain usable without SQLite, while SQLite C types and callbacks stay inside one subsystem.

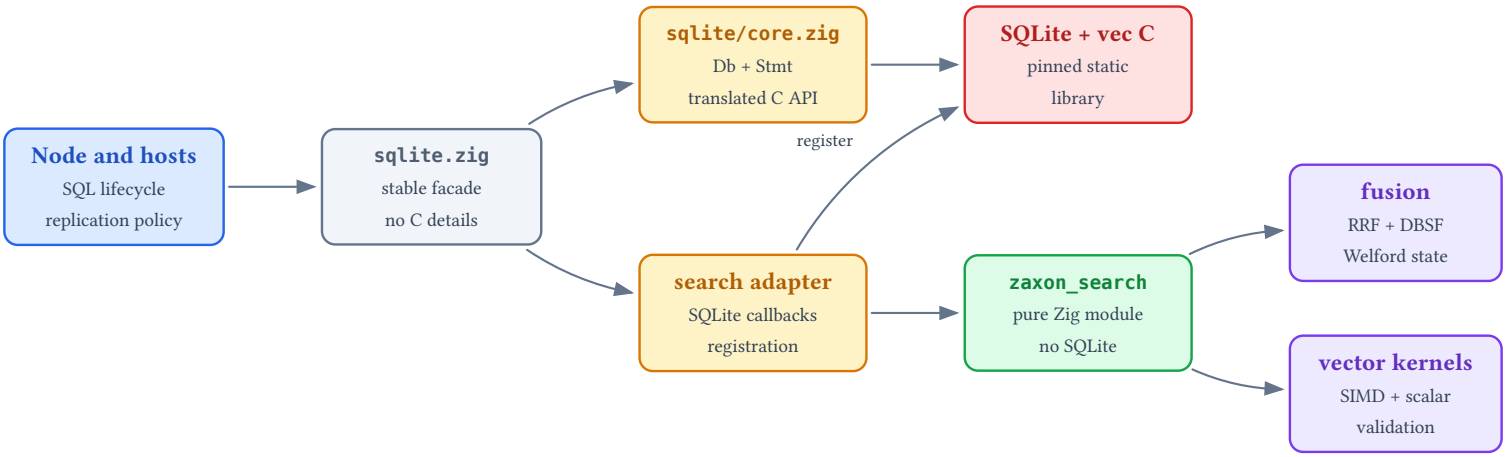

Figure 2: Only the SQLite subsystem sees translated C declarations. Pure fusion and vector kernels have no database or consensus dependency.

The proposed source layout is:

```
zaxonlite/src/
  sqlite.zig                stable facade imported by the product
  sqlite/
    core.zig                narrow SQLite C wrapper, Db, and Stmt
    search_extension.zig    registration and SQLite callback glue
  search/
    root.zig                zaxon_search module exports
    fusion.zig              RRF, DBSF, and Welford statistics
    vector.zig              scalar/SIMD distance kernels and validation
```

`build.zig` creates `zaxon_search` as an internal Zig module and imports it into the zaxonlite root module. It is not a separate package, repository, or semantic-versioned public artifact in the first release. A second independent consumer is the promotion criterion: if another package needs the pure algorithms, `search/` can be extracted without moving SQLite or Paxos code because its dependency boundary is already clean.

Component	May import	Must not import	Owns
`zaxon_search`	Zig standard math, builtin target information	SQLite C API, Paxos, filesystem, network, allocator	Formulas, numeric contracts, scalar and SIMD kernels
`sqlite/core.zig`	translated SQLite C module, search adapter	Paxos, node, transport	Connection, statement, binding, result, and open lifecycle
`sqlite/search_extension.zig`	translated SQLite C module, `zaxon_search`	node, Paxos, filesystem, network	C callbacks, SQLite value conversion, function registration
`sqlite.zig`	SQLite subsystem files	translated C declarations directly	Stable import path and re-exported narrow types
node and hosts	`sqlite.zig`, application policy	search internals, sqlite-vec symbols	Transactions, query limits, replication, and status

The dependency graph is acyclic. `sqlite/core.zig` opens the raw connection and calls `search_extension.register(handle)` before returning it. `search_extension.zig` accepts the raw SQLite handle and does not import `core.zig`, so registration does not create a module cycle.

The existing rule that only `sqlite.zig` touches the C header becomes a directory boundary: only files under `src/sqlite/` may import the translated `c` module. Product code continues to import only `sqlite.zig`.

Each module owns its tests:

- `fusion.zig` uses table-driven formula and statistical tests with no SQLite;
- `vector.zig` compares scalar and SIMD kernels across the target matrix;
- `search_extension.zig` tests SQLite arity, type conversion, errors, and registration against an in-memory connection;
- `sqlite/core.zig` retains connection and statement lifecycle tests;
- node and cluster tests own WAL capture, restart, and replica behavior.

8.2. Build and connection lifecycle

The SQLite build gains an explicit FTS5 flag. The sqlite-vec amalgamation is a pinned Zig package dependency compiled into the same static SQLite library with its supported static-link mode and filesystem helpers omitted. The pin is sqlite-vec v0.1.9 (upstream is pre-1.0; the pin names the exact release, not a stability promise), release zip SHA-256 `b87cdda12112657ba5ab8842f0088a4090982eaf41f22b2bd6d495b81765a8c9`, compiled with `SQLITE_CORE`, `SQLITE_VEC_STATIC`, and `SQLITE_VEC_OMIT_FS` (which removes the `vec_npy_file`/`vec_npy_each` filesystem readers). Architecture-specific sqlite-vec flags are enabled only when the resolved Zig target guarantees the corresponding instructions.

`sqlite.Db.open` becomes the single registration boundary:

1. open the SQLite connection;
2. register sqlite-vec directly with `sqlite3_vec_init`;
3. register the Zig RRF, DBSF, standard-deviation, and SIMD-distance callbacks;
4. configure the connection's mmap limit before preparing statements;
5. return the usable connection or close it and return an error.

Direct registration is preferred over `sqlite3_auto_extension`. It makes initialization failure local to `Db.open`, avoids process-global mutable registration state, and covers all call sites already routed through `sqlite.Db.open`: live writers, short read leases, WAL tests, restored-image checks, and backup validation.

Both regular and benchmark SQLite libraries use the same extension sources and compile flags. A shared build helper constructs the SQLite library, `zaxon_search` module, translated-C import, and

zaxonlite module for the requested optimization mode. This prevents the benchmark graph from silently using different SIMD or extension flags. Runtime extension loading remains omitted.

8.3. SQLite-managed memory mapping

zaxonlite does not call the operating system's `mmap` API on `current.db`. It asks SQLite to use its VFS `xFetch`/`xUnfetch` path. SQLite then owns mapping lifetime, statement safety, WAL visibility, and fallback behavior.

The proposed configuration is:

Profile	Limit	Behavior
default	`0`	Disable mmap on every target. sqlite-vec continues chunked `xRead` scans without exposing mapped-I/O faults to the process.
opt-in 64-bit performance	`256 MiB`	Map at most the first 256 MiB after explicit operator configuration. Pages are faulted on demand; this is an address-space ceiling, not a heap allocation.
custom opt-in	`1..1 GiB`	Embedding hosts may select a smaller or larger nonzero cap before any statement runs. Values above 1 GiB are rejected by zaxonlite even if SQLite permits them.

The build sets a nonzero `SQLITE_MAX_MMAP_SIZE` on supported 64-bit targets and leaves `SQLITE_DEFAULT_MMAP_SIZE` at zero. zaxonlite explicitly sets zero on each connection unless the operator opts into a nonzero profile. A platform where SQLite does not support mapped I/O silently uses normal reads; startup reads the effective value back and exposes it in node status.

Application SQL may read `PRAGMA mmap_size` but may not write it. The authorizer adds `mmap_size` to the connection-owned write-denied pragmas. Changing the limit while a statement is active can be a no-op in SQLite and would make query memory policy unpredictable.

Mapped I/O primarily benefits reads by avoiding an extra page-cache copy. It does not replace chunking or quantization, and it is not a durability mechanism. Operators may disable it because an I/O fault delivered through a mapped page cannot be converted into an ordinary SQLite error. zaxonlite's trusted local-image model, startup integrity validation, and no-open-handle offline apply rule reduce but do not eliminate that operational risk.

8.4. SQLite-compatible EBNF function contract

SQLite uses the term syntax diagram for its complete SQL grammar. The grammar below is an additive expression grammar: each production expands to an ordinary SQLite function call and can appear wherever SQLite accepts an expression. Keywords and function names are ASCII case-insensitive. Whitespace and comments follow SQLite's tokenizer.

This is the only grammar defined by this record. Application tables, keys, relationships, triggers, and embedding-space catalogs remain application policy. Identifiers such as `media_item_fts` and `vector_reranked` in later queries are illustrative inputs to the extension functions, not prescribed relations.

```
search-extension-expression ::= rrf-call
                              | dbsf-call
```

```
                              | stddev-call
                              | cosine-call ;

rrf-call          ::= "rrf" "(" rank-expression
                      [ "," k-expression
                        [ "," weight-expression ] ] ")" ;

dbsf-call         ::= "dbsf" "(" score-expression
                      "," mean-expression
                      "," stddev-expression
                      [ "," weight-expression ] ")" ;

stddev-call       ::= "stddev_samp" "(" score-expression ")" ;

cosine-call       ::= "zaxon_vec_distance_cosine" "("
                      vector-expression "," vector-expression ")" ;

rank-expression   ::= sqlite-expression ;
k-expression      ::= sqlite-expression ;
weight-expression ::= sqlite-expression ;
score-expression  ::= sqlite-expression ;
mean-expression   ::= sqlite-expression ;
stddev-expression ::= sqlite-expression ;
vector-expression ::= sqlite-expression ;

sqlite-expression ::= literal-value
                    | bind-parameter
                    | column-name
                    | unary-expression
                    | binary-expression
                    | function-call
                    | cast-expression
                    | case-expression
                    | "(" sqlite-expression ")" ;

bind-parameter    ::= "?" [ decimal-digits ]
                    | ":" parameter-name
                    | "@" parameter-name
                    | "$" parameter-name ;
```

`sqlite-expression` is intentionally a reference to SQLite's existing `expr` production rather than a competing SQL parser. No statement grammar is added. Arity overloading registers `rrf` with one, two, and three arguments, and `dbsf` with three and four arguments. `zaxon_vec_distance_cosine` has exactly two arguments.

8.5. RRF contract

For one retriever result at one-based rank `r`, rank constant `k`, and positive weight `w`:

$$\mathrm{RRF}(r, k, w) = \frac{w}{k + r}$$

Defaults are `k = 60` and `w = 1`. `rank` must be an integer greater than zero, `k` must be finite and greater than zero, and `weight` must be finite and nonnegative. Null input produces null. Invalid input returns a SQLite constraint error. An item absent from a retriever contributes zero because it has no row in that branch.

RRF depends only on ordering. It is the default fusion when retriever scores are uncalibrated, candidate counts differ greatly, or one branch has unstable score magnitude.

Example:

```
WITH
lexical AS (
  SELECT rowid AS item_id,
         row_number() OVER (ORDER BY bm25(media_item_fts), rowid) AS rank
  FROM media_item_fts
  WHERE media_item_fts MATCH :text_query
  ORDER BY bm25(media_item_fts), rowid
  LIMIT :lexical_candidates
),
semantic AS (
  SELECT item_id,
         row_number() OVER (ORDER BY exact_distance, item_id) AS rank
  FROM vector_reranked
),
contributions AS (
  SELECT item_id, rrf(rank, 60, :text_weight) AS score FROM lexical
  UNION ALL
  SELECT item_id, rrf(rank, 60, :vector_weight) AS score FROM semantic
)
SELECT item_id, sum(score) AS fused_score
FROM contributions
GROUP BY item_id
ORDER BY fused_score DESC, item_id
LIMIT :k;
```

8.6. DBSF contract

Each retriever first orients scores so higher is better:

- lexical score is `-bm25(media_item_fts)`;
- vector score is `-exact_distance`.

For a score `s`, sample mean `mu`, sample standard deviation `sigma`, and nonnegative weight `w`:

$$\mathrm{DBSF}(s, \mu, \sigma, w) = w\left(0.5 + \frac{s - \mu}{6\sigma}\right)$$

When the set has one score or all scores are equal, the normalized score is `0.5w`. Values are deliberately not clipped. Null input produces null. Non-finite values, negative standard deviation, or negative weight return a SQLite error.

`stddev_samp(x)` uses Welford's online algorithm in a fixed SQLite aggregate context. It ignores null rows, returns null for an empty set, and returns zero for a singleton so `dbsf` applies the neutral rule.

Example:

```
WITH
lexical_raw AS (
  SELECT rowid AS item_id, -bm25(media_item_fts) AS score
  FROM media_item_fts
  WHERE media_item_fts MATCH :text_query
  ORDER BY bm25(media_item_fts), rowid
  LIMIT :lexical_candidates
```

```
),
lexical AS (
  SELECT item_id,
         dbsf(
           score,
           avg(score) OVER (),
           stddev_samp(score) OVER (),
           :text_weight
         ) AS score
  FROM lexical_raw
),
semantic_raw AS (
  SELECT item_id, -exact_distance AS score
  FROM vector_reranked
),
semantic AS (
  SELECT item_id,
         dbsf(
           score,
           avg(score) OVER (),
           stddev_samp(score) OVER (),
           :vector_weight
         ) AS score
  FROM semantic_raw
),
contributions AS (
  SELECT * FROM lexical
  UNION ALL
  SELECT * FROM semantic
)
SELECT item_id, sum(score) AS fused_score
FROM contributions
GROUP BY item_id
ORDER BY fused_score DESC, item_id
LIMIT :k;
```

DBSF is selected when the magnitude within each retriever carries useful relevance information. Candidate limits should be comparable because DBSF estimates its distribution from returned candidates, not the whole corpus.

8.7. Zig implementation

The C-facing registration remains in `sqlite.zig`, preserving the rule that no other module imports the translated SQLite header. Pure formulas, vector kernels, and their unit tests live in a domain-neutral search module.

The SQLite callbacks have these properties:

- `callconv(.c)` and no unwinding across the C boundary;
- `SQLITE_DETERMINISTIC | SQLITE_INNOCUOUS | SQLITE_UTF8`;
- no allocation for `rrf` or `dbsf`;
- one 24-byte aggregate state for `stddev_samp`: count, mean, and `M2`;
- no allocation for `zaxon_vec_distance_cosine`;
- checked conversion from SQLite numeric values;
- explicit finite-number validation;
- `sqlite3_result_error` on contract violations;

- scalar work bounded to constant time.

The standard-deviation window implementation supplies step, inverse, value, and final callbacks. The inverse update clamps only tiny negative `M2` values caused by floating-point cancellation; a material negative value is an internal error.

8.8. SIMD feasibility and CPU dispatch

SIMD belongs in the distance kernels, not in the fusion formulas.

RRF performs one division per candidate row. DBSF performs a handful of floating-point operations per row. SQLite invokes each scalar callback with one row, so there is no contiguous batch for those functions to vectorize. `stddev_samp` also has a loop-carried Welford dependency. These functions stay allocation-free scalar `f64`; reducing SQLite callback and grouping work is more valuable than forcing SIMD into them.

The hot vector work has two distinct implementations:

Stage	Kernel	Parallelism	Decision
coarse	sqlite-vec Hamming	64 dimensions per `u64` XOR and popcount	Keep the upstream chunked kernel. It is word-parallel and may compile to a hardware popcount, but is not described as SIMD.
rerank	Zig cosine	four-way unrolled 128-bit vectors, 16 float32 dimensions per iteration	Normative SIMD kernel with scalar tail and scalar fallback.
optional L2	sqlite-vec L2	upstream AVX or NEON path	Enable only for a target that guarantees the instruction set.

The stable sqlite-vec source does not provide runtime CPU dispatch. Its AVX path executes AVX whenever it is compiled in, and its current cosine loop is scalar. Enabling `SQLITE_VEC_ENABLE_AVX` in a generic x86-64 binary could therefore cause an illegal instruction on an older CPU and would not accelerate cosine. It stays off for the portable x86-64 artifact. A CPU-specific build may enable it only when the Zig resolved target includes AVX.

The Zig cosine kernel uses a conservative 128-bit vector width. Backend selection comes from the resolved Zig target, never from a CPU vendor string:

Target	Backend	Initial mmap	Rule
macOS AArch64 / Apple M-series	NEON 128-bit	`0`	Advanced SIMD is mandatory in AArch64. The 256 MiB mapped-I/O profile is operator opt-in. No Apple AMX or Accelerate dependency is required.
Linux/Windows AArch64	NEON 128-bit	`0`	Same kernel, SQL behavior, and explicit mmap opt-in as Apple Silicon.
ARMv7 / 32-bit ARM	NEON if target feature; otherwise scalar	`0`	Never emit NEON for an artifact whose target does not guarantee it.
x86-64	SSE 128-bit	`0`	Portable artifact keeps sqlite-vec AVX disabled; tuned builds and mmap remain explicit opt-ins.
x86 / 32-bit	SSE2 if target feature; otherwise scalar	`0`	Conserve address space and retain the scalar fallback.
wasm32	`simd128` if target feature; otherwise scalar	`0`	No assumption of an operating-system mmap VFS.
RISC-V, PowerPC64LE, other little-endian targets	scalar initially	`0`	Add RVV, VSX, or an mmap profile only after compiler, disassembly, VFS, and conformance gates.

The first vector format supports little-endian targets. sqlite-vec stores float vector payloads as raw BLOB bytes and interprets them with native float loads, so a big-endian binary cannot safely query an image produced by a little-endian leader. The build fails closed on big-endian targets until the extension defines a canonical cross-endian vector representation. This still covers Apple M-series, mainstream AArch64, ARMv7 little-endian, x86, wasm, and mainstream RISC-V deployments.

Four-way unrolling uses twelve accumulation vectors: four each for dot product, squared magnitude of `a`, and squared magnitude of `b`. This reduces dependency-chain stalls while processing 16 dimensions per iteration. The kernel then performs fixed horizontal reductions and a scalar tail:

$$\text{cosine-distance}(a, b) = 1 - \frac{\text{sum}(a_i b_i)}{\sqrt{\text{sum}(a_i^2)\ \text{sum}(b_i^2)}}$$

The implementation uses explicit Zig `@Vector(4, f32)` operations rather than depending on compiler auto-vectorization. It does not enable global fast-math. Both inputs must be float32 BLOBs with the same nonzero length divisible by four. Non-finite elements, a zero magnitude, or malformed BLOBs return a SQLite error.

SIMD changes floating-point reduction order. The function therefore promises full float32-vector reranking, not bit-identical distances across CPU architectures. Stable final ordering uses item

ID after distance. Tests compare scalar and SIMD distances within a documented tolerance and include adversarial nearly tied vectors.

Apple M-series builds use the same explicit reduction topology as other 128-bit backends. Fused multiply-add contraction is disabled for this kernel, so Apple NEON and x86 SSE do not silently use different arithmetic graphs. The implementation does not promise identical platform `sqrt` rounding in the last bit; callers needing a canonical audit result can select the scalar reference kernel.

`zaxon_search_debug()` reports `simd=neon128`, `simd=sse128`, `simd=wasm128`, or `simd=scalar`, plus the sqlite-vec build flags returned by `vec_debug()`. Release verification also inspects target object code: supported AArch64 builds must contain vector multiply/add instructions and supported x86-64 builds must contain packed float multiply/add instructions in the Zig cosine kernel. A benchmark alone is not accepted as proof of SIMD.

8.9. Vector storage and query

The application writes both representations:

```
INSERT INTO media_text_vec(item_id, embedding, embedding_coarse)
VALUES (
  :item_id,
  vec_f32(:normalized_embedding),
  vec_quantize_binary(:normalized_embedding)
);
```

The coarse query and exact rerank are:

```
WITH coarse AS (
  SELECT item_id, embedding
  FROM media_text_vec
  WHERE embedding_coarse MATCH vec_quantize_binary(:query_embedding)
    AND k = :candidate_count
),
vector_reranked AS (
  SELECT item_id,
         zaxon_vec_distance_cosine(
           embedding,
           :query_embedding
         ) AS exact_distance
  FROM coarse
  ORDER BY exact_distance, item_id
  LIMIT :k
)
SELECT * FROM vector_reranked;
```

For dimension `d`, corpus size `N`, and candidate count `C`, the work is:

- coarse scan time: `O(N * d / word_bits)` Hamming operations;
- rerank time: `O(C * d / SIMD_lanes)` vector operations plus scalar tails;
- query heap: `O(C + d)`;
- coarse index bytes: approximately `N * d / 8`, excluding SQLite metadata;
- exact vector bytes: `N * 4d`, stored on disk and read for candidates.

The typed search-result surface returns item IDs, fused or per-retriever scores, and application-selected metadata. It does not return raw embedding BLOBs unless a caller uses ordinary SQL to request them explicitly. For `k = 100` and `d = 1536`, float32 embeddings alone would add `100 * 1536 * 4 = 614400` bytes, or 600 KiB, to one response before framing. Keeping vectors out of the default result also prevents the search helper from coupling its API to an embedding element format.

The single-table layout above — one vec0 table holding both the float32 and the bit column, with KNN `MATCH` on the bit column — is verified against the pinned v0.1.9 source and covered by conformance tests. Raw BLOBs bind as float32 by default; bit vectors bind through `vec_bit()` or `vec_quantize_binary()`.

sqlite-vec's shadow tables already divide vectors into chunks and scan one chunk at a time. SQLite-managed mmap allows those chunks to be faulted from the OS page cache without first copying every page into SQLite heap. If mmap is disabled, the same algorithm uses bounded SQLite page-cache reads.

Binary quantization can reduce recall. The implementation therefore never returns Hamming ordering as the final semantic order. Every release benchmark records recall after exact rerank for every exercised embedding space. The checked representative fixture exercises the text- and image-query paths but makes no neural-model quality claim. Text/image quality is qualified only by the optional GME/Qwen harness; audio remains a storage and query compatibility path until an audio model is separately qualified.

8.10. Typed search operation

The network host exposes one typed `search` operation beside `query`. It is the enforced path for the candidate ceiling: the host validates table identifiers (ASCII identifier characters only, never the `__zaxon_` or `sqlite_` namespaces), `k` and `candidate_count` in `1..=4096` (default `min(max(8k, 64), 4096)`), finite nonnegative weights, and the embedding shape (float32, dimension divisible by eight) before any SQL exists. An optional metadata projection validates one application table, its item-ID column, and at most 16 selected column identifiers. The host then builds exactly the canonical statements this record documents — lexical-only, coarse-plus-rerank vector-only, or the RRF/DBSF hybrid — with every user value bound, never spliced. The built statement runs through the standard read path, so read levels, leases, guards, and query budgets apply unchanged, and the result carries item IDs, scores, and requested metadata, never implicit embedding BLOBs. Raw SQL remains fully supported for everything else; the typed operation is a convenience and an enforcement point, not a new query language.

8.11. Hybrid query sequence

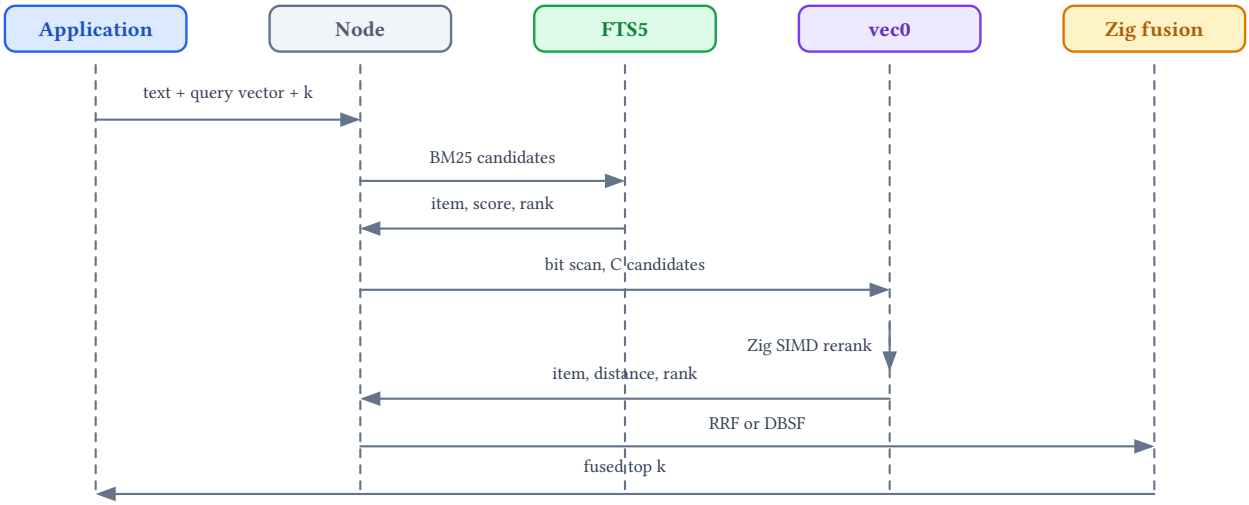

Figure 3: One SQLite statement obtains candidates, reranks vectors, and fuses results. The diagram separates actors to make the execution stages visible.

SQLite may execute the CTEs according to its query plan; the sequence defines logical data dependencies, not a promise of concurrent branch execution.

8.12. Ingestion and replication flow

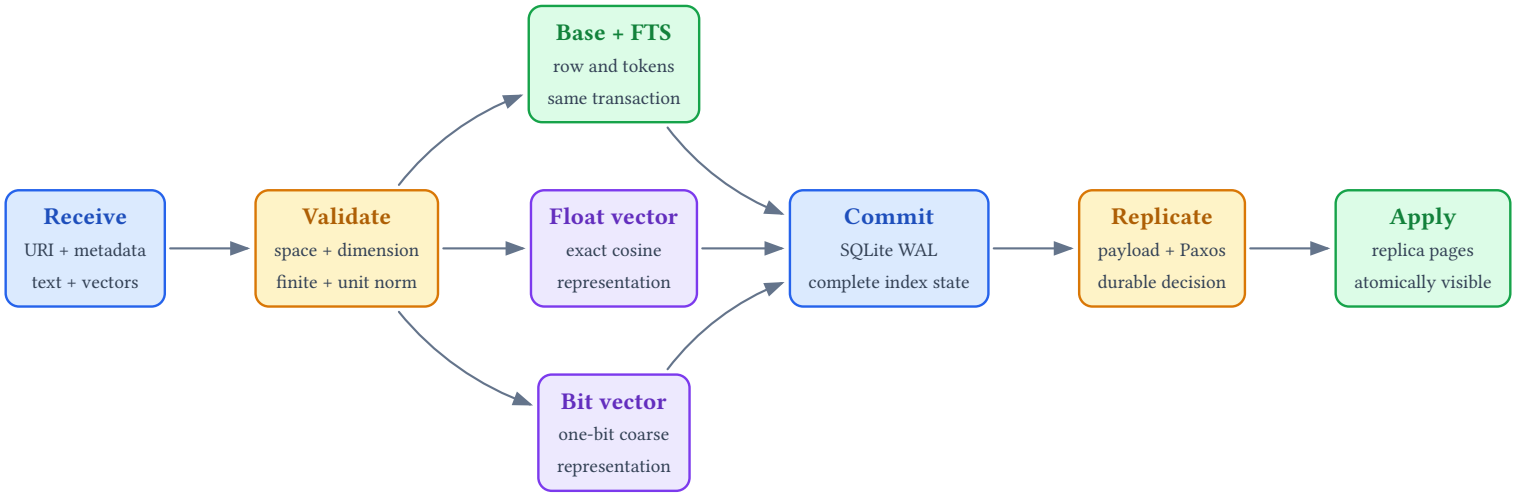

Figure 4: All logical and physical search representations commit in one captured SQLite transition.

Validation occurs before `BEGIN IMMEDIATE` where possible. The actual insert still relies on vec0 dimension checks and SQL constraints. If any base, FTS, float, or bit write fails, zaxonlite rolls back the entire transaction.

8.13. Fusion selection flow

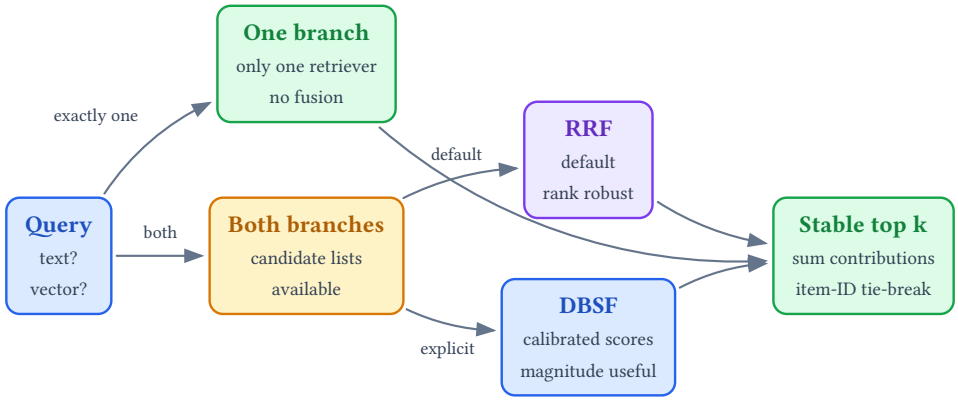

Figure 5: RRF is the safe default. DBSF is an explicit choice for meaningful score distributions.

9. Failure Semantics

Condition	Result	Reason
sqlite-vec registration fails	`Db.open` fails	A connection must never serve a schema whose virtual-table module is missing.
dimension or vector type mismatch	statement error; transaction rolls back	Mixed embedding spaces cannot produce meaningful distance.
query vector contains NaN or infinity	statement error	Non-finite values break ordering and distribution statistics.
`candidate_count > 4096`	query rejected	Heap and rerank work remain bounded.
mmap unsupported	normal SQLite reads	`PRAGMA mmap_size` is advisory and may be a no-op.
mapped-page I/O fault	process or query may fault	Documented SQLite mmap risk; operators can select the constrained profile.
one fusion branch is empty	return the nonempty branch	Absence contributes zero; no synthetic distribution is created.
both branches empty	empty result	An empty query is not an error.
constant DBSF scores	every row gets `0.5 * weight`	Avoid division by zero and preserve a neutral contribution.

10. Security Considerations

sqlite-vec is statically linked from a pinned source archive. Runtime extension loading remains omitted, and sqlite-vec filesystem helpers are not compiled. The extension version and source hash become part of the build provenance.

The database is still an application-principal database, not a hostile multi-tenant SQL sandbox. Existing authorizer rules continue to protect the capture contract. The new rule denies application writes to `mmap_size`.

SQLite recommends disabling mmap in especially security-sensitive applications because a mapped I/O fault cannot be contained as an SQLite error. The runtime default is therefore zero on every target. Enabling a nonzero profile is an explicit operator acceptance of this process-crash risk; it does not change the cryptographic, authentication, or consensus trust model.

FTS query syntax can consume substantial CPU. Vector scans are linear in corpus size. Network hosts must retain `max_vm_steps`, row, byte, statement deadline, and connection concurrency limits. The vector candidate cap is checked independently of SQLite VM steps because most work occurs inside a virtual-table callback.

Media URIs are inert strings. zaxonlite never dereferences them. Extracted text and metadata are ordinary untrusted values and must be rendered with context-appropriate escaping by applications.

11. Replication and Compatibility

FTS5 segment merges and vec0 shadow-table updates may change more pages than the application row alone. That increases payload size but does not alter the safety argument: the commit frame names the final database size, and followers apply every captured page in order.

The feature introduces an on-disk compatibility dependency. A database schema containing `USING vec0` cannot be queried by a binary without that module. Every release therefore records:

- SQLite version and `ENABLE_FTS5`;
- sqlite-vec version and source hash;
- vector element formats in use;
- Zig fusion API version;
- Zig distance-kernel version and selected SIMD backend;
- mmap compile-time maximum.

The internal metadata schema gains a search-feature version. A binary refuses to serve an image whose required feature version is newer than it implements. Rolling upgrades must first deploy a binary that understands the feature to every member, then create FTS5 or vec0 schema. Downgrades across an activated feature version are unsupported unless the application drops the virtual tables while running a compatible binary.

12. Operational Considerations

The node status surface reports FTS5 availability, sqlite-vec version, search feature version, selected SIMD backend, effective mmap limit, and the candidate hard limit.

Operators should measure:

- p50, p95, and p99 FTS, vector, and hybrid latency;
- pages read and major page faults per vector query;
- process resident set size and SQLite heap high-water mark;
- coarse candidate recall and final recall at `k`;
- WAL payload bytes per indexed write;
- checkpoint and snapshot size growth;
- query interruption and candidate-limit rejection counts.

`PRAGMA optimize` and FTS maintenance remain operator-controlled SQL subject to the existing transaction guard. Search-maintenance transactions target at most 16 MiB of captured WAL payload and have a 32 MiB operational soft ceiling. The protocol hard limit remains `command.max_payload_bytes = 64 MiB - 73 bytes`; the operational ceiling does not change the wire format.

FTS5 maintenance uses its bounded merge command. To express a four-segment threshold and roughly ten pages of work per invocation, use separate FTS5 commands:

```
INSERT INTO search_fts(search_fts, rank) VALUES('usermerge', 4);
INSERT INTO search_fts(search_fts, rank) VALUES('merge', 10);
```

The older `merge=10,4` spelling belongs to FTS3/4 and is not valid FTS5 policy. The application repeats bounded merge transactions while work remains and stops or reduces the page budget if the captured payload crosses the 16 MiB target.

A 1536-dimensional float32 vector occupies 6144 bytes; its coarse bit vector adds 192 bytes before SQLite record, page, index, and WAL overhead. Therefore 5000 rows require about 30.2 MiB and 10000 rows about 60.4 MiB of vector payload alone. Bulk vector work is byte-budgeted rather than fixed at 5000 to 10000 rows. The batch controller is application-side policy, not product code: the application starts at 1000 to 2000 rows, observes captured WAL bytes, and adapts without crossing 32 MiB, following the documented pattern in the zaxonlite book. The replication tests prove a 1500-row 1536-dimensional batch stays inside the 16 MiB target. Rebuilds may also build a replacement table and switch schemas in a reviewed migration.

13. Verification and Acceptance

13.1. Unit tests

- RRF defaults, weights, ties, invalid rank, invalid `k`, nulls, and non-finite values.
- DBSF known distributions, sample deviation, singleton, constant set, weights, nulls, and non-finite values.
- Welford step/inverse stability under sliding windows.
- sqlite-vec registration failure closes the connection.
- mmap initializes to zero on every target, an explicit 256 MiB opt-in is set before statements, and the effective size is reported.
- application writes to `mmap_size` are denied.
- scalar and SIMD cosine agree within tolerance for aligned, unaligned, tail-length, zero, non-finite, and nearly tied inputs.
- target feature selection never exposes a SIMD kernel to an unsupported CPU.
- cross-build the kernel for macOS AArch64, Linux AArch64, Windows AArch64, ARMv7 with and without NEON, generic x86-64, wasm32 with and without `simd128`, and RISC-V scalar.
- reject unsupported big-endian vector builds at compile time with a clear diagnostic.

13.2. SQLite integration tests

- `CREATE VIRTUAL TABLE ... USING fts5` and BM25 ranking.
- float32, int8, and bit vec0 construction and KNN queries.
- binary coarse scan followed by exact cosine rerank.
- `zaxon_vec_distance_cosine` accepts sqlite-vec float BLOBs and rejects other vector types or dimensions.
- the EBNF function forms prepare successfully using literals, columns, and all four SQLite bind-parameter forms.
- RRF and DBSF examples execute without SQL rewriting.
- typed search results omit embedding BLOBs and return stable IDs, scores, and selected metadata.
- `load_extension`, `readfile`, and `writefile` remain unavailable.

13.3. Replication tests

- Create base, FTS5, and vec0 schema through a replicated write.
- Insert, update, and delete one multimodal item atomically.

- Verify equal query results on leader, follower, read replica, restart, and restored snapshot.
- Extend the byte-identical WAL spike with FTS segment changes and vec0 shadow-table changes.
- prove bounded FTS5 merge and adaptive vector batches stay within the 16 MiB target and never cross the 32 MiB operational ceiling.
- reject a transaction above the `64 MiB - 73 bytes` protocol hard limit.
- Crash at every payload, journal, commit, apply, and checkpoint failpoint.
- Reject a binary with an older search-feature manifest.

13.4. Benchmark fixtures

The mandatory representative fixture is deterministic, model-free, and generated entirely with the Python standard library. It contains 96 corpus vectors plus 12 text and 12 image query vectors at 512 dimensions. The complete bundle is approximately 244 KiB and needs no model weights, so regeneration and verification are routine on an 8 GiB Apple M1 as well as CI. It exercises NumPy parsing, float and bit vec0 storage, coarse candidate selection, exact reranking, modality-specific query arrays, and recall recording. It is a mechanical regression oracle, not evidence about an embedding model:

```
zaxonlite/benchmarks/data/representative-v1-512/
  manifest.json
  corpus.f32.npy
  text-queries.f32.npy
  image-queries.f32.npy
  relevance.json
```

`generate-representative-fixture.py` deterministically recreates the bundle without NumPy, PyTorch, model weights, or network access. `verify-fixture.py` treats the bundle as required and validates its NumPy shape, little-endian float32 layout, L2 normalization, relevance structure, manifest, and artifact hashes.

Model-quality qualification is a separate, optional tier. The retained `generate-gme-fixture.py` harness loads `Alibaba-NLP/gme-Qwen2-VL-2B-Instruct`, produces 1536-dimensional text and image arrays, and records the exact model revision, prompts, preprocessing, licenses, row IDs, and hashes. Those artifacts live under `benchmarks/data/gme-qwen2-vl-2b-1536/` when explicitly generated. Routine CI does not download or run Qwen. The representative bundle cannot be used to claim text, image, or audio retrieval quality.

13.5. Performance gates

Benchmarks use the recorded fixtures and publish raw result files. The first release must demonstrate:

- query heap growth bounded by `candidate_count`, within measurement noise;
- coarse bit storage no more than one thirty-second of raw float payload, excluding SQLite page and table metadata;
- representative text- and image-query recall measured at oversampling factors 4, 8, and 16 on every run;
- mmap-on and mmap-off results for latency, RSS, and page faults;
- scalar versus SIMD rerank throughput at the intended 1536-dimensional production size, compatibility dimensions 384, 768, and 1024, and one non-multiple-of-four tail case;
- disassembly evidence for packed float instructions in supported release targets;
- Apple M-series benchmark evidence from at least one baseline M1/M2 machine and one later M-series generation;

- no unbounded allocation proportional to corpus row count;
- no regression to ordinary non-search transactions beyond the cost of explicitly maintained application indexes.

The representative fixture must retain `recall@10 = 1.0` at factors 4, 8, and 16; this is a deterministic regression threshold, not a quality target. No universal model-quality threshold is specified because models, corpora, and modalities differ. A Qwen qualification run must record its acceptance threshold before binary coarse search is enabled. If factor 16 cannot meet that threshold, the product uses an int8 coarse table or exact float scan for that embedding space.

14. Delivery Plan

All implementation steps below are complete as of 2026-07-29. The checked representative arrays are present, verified in CI, and exercised by the recorded benchmark. GME/Qwen 2B inference remains an explicitly invoked model-quality qualification harness rather than a release-build dependency.

1. Create the internal `zaxon_search` module with independent `fusion.zig` and `vector.zig` tests; keep its public surface free of SQLite types.
2. Divide the SQLite wrapper into `sqlite/core.zig` and `sqlite/search_extension.zig`, preserving `sqlite.zig` as the only stable product-facing import.
3. Centralize the normal and benchmark build graphs in one helper, then add FTS5 and pinned static sqlite-vec to every SQLite build target.
4. Register search modules, expose a feature manifest on every connection, and add SQLite-owned mmap configuration and authorizer protection.
5. Implement and test allocation-free RRF, DBSF, `stddev_samp`, and portable SIMD cosine distance in Zig.
6. Add import-boundary tests that compile `zaxon_search` without SQLite and reject translated-C imports outside `src/sqlite/`.
7. Add direct SQLite conformance tests for the documented grammar and SQL.
8. Add replicated FTS5/vec0 lifecycle and byte-identical rebuild tests.
9. Add the checked 512-dimensional representative fixture and CI recall; retain the 1536-dimensional GME/Qwen 2B harness for later qualification.
10. Publish ingestion, RRF, DBSF, and vector-rerank examples in the zaxonlite book.
11. Enable the search feature version only after all cluster members run the compatible binary.

15. Alternatives Considered

15.1. Standalone search package in the first release

Rejected for now. A separate repository and semantic-versioned package would create a public compatibility promise before there is a second consumer. The internal `zaxon_search` module provides the useful separation immediately. Its SQLite-free dependency rule makes later extraction mechanical if another product genuinely needs the algorithms.

15.2. Raw float32 exhaustive scan

This is exact and simple, but reads 32 times as many vector payload bytes as a bit scan. It remains the correctness baseline and fallback for embedding spaces that do not tolerate quantization.

15.3. HNSW

HNSW offers fast approximate search but keeps a graph with substantial memory overhead. Dynamic graph updates and a new index format would enlarge the replication and compatibility surface. It is not selected for the memory-bounded first release.

15.4. DiskANN

DiskANN is designed for low-RAM approximate search, but sqlite-vec's DiskANN work is currently prerelease. Depending on an unstable entrypoint would make the on-disk contract and cross-platform behavior hard to freeze. It can be reconsidered after a stable sqlite-vec release stores all required state in ordinary SQLite pages and passes zaxonlite's rebuild oracle.

15.5. Direct operating-system mmap

Rejected. Mapping `current.db` outside SQLite would bypass SQLite's VFS, locking, WAL visibility, and mapping lifetime. It could overlap offline page apply and invalidate zaxonlite's no-open-handle invariant.

15.6. RRF or DBSF in application code

Rejected as the only interface. It requires transferring larger candidate sets, duplicates numeric edge cases across clients, and prevents SQLite from performing grouping and limiting before rows cross the API boundary.

15.7. A custom hybrid-search SQL statement

Rejected. It requires a parser or SQLite grammar fork. Ordinary CTEs and registered functions are valid SQLite SQL, composable with application filters, and visible to `EXPLAIN QUERY PLAN`.

16. Resolved Design Decisions

Question	Resolved answer	Constraint or metric
Q1: model / suite	A required deterministic 512-dimensional `.npy` representative suite; GME/Qwen 2B remains an optional 1536-dimensional quality harness.	The checked suite is under 250 KiB and runs on an 8 GiB M1 without inference or third-party Python packages. It proves mechanics, not model quality. Qwen artifacts must pin revision, configuration, licenses, and hashes when qualified.
Q2: mmap policy	Explicit opt-in. Every target starts with `mmap_size = 0`; supported 64-bit operators may select the 256 MiB profile.	A mapped I/O error cannot be converted to an SQLite error and may terminate the process. mmap-on and mmap-off benchmarks remain required.
Q3: payload ceiling	16 MiB captured-payload target, 32 MiB operational soft ceiling, and unchanged protocol hard limit of `64 MiB - 73 bytes`.	Use FTS5 `usermerge=4` plus bounded `merge=10` commands. Batch vectors by observed bytes, initially 1000 to 2000 rows; 5000 to 10000 is not a safe fixed batch for 1536-dimensional float32 plus coarse vectors.
Q4: result types	IDs, scores, and application-selected metadata by default. Embedding BLOBs require an explicit ordinary SQL projection.	At `k = 100`, 1536-dimensional float32 vectors add exactly 600 KiB before response framing.
Q5: candidate-cap enforcement	The typed `search` operation validates `candidate_count` in `1..=4096` before building SQL; raw application SQL treats the ceiling as a documented contract backed by the row, byte, and VM-step budgets, because the vec0 `k` binding is invisible to the host.	The VM-step budget under-counts virtual-table scans; rows, bytes, and the statement deadline are the operative raw-SQL bounds. Node status reports `candidate_hard_limit`.
Q6: sqlite-vec pin	v0.1.9, static amalgamation, `SQLITE_CORE` + `SQLITE_VEC_STATIC` + `SQLITE_VEC_OMIT_FS`, single-table float+bit layout confirmed against the pinned source.	Release zip SHA-256 `b87cdda12112657ba5ab8842f0088a4090982eaf41f22b2bd6d495b81765a8c9`; the Zig package hash pins the dependency in `build.zig.zon`.

These decisions close the open questions for the first release. Selecting and qualifying an audio embedding model is a later retrieval-quality decision; it does not change the generic SQLite vector storage or fusion contracts.

17. References

- SQLite FTS5
- SQLite memory-mapped I/O
- SQLite `PRAGMA mmap_size`
- SQLite security guidance
- sqlite-vec KNN queries
- sqlite-vec binary quantization and exact rescoring
- sqlite-vec static compilation
- Zig vector types
- Qdrant RRF and DBSF behavior
- GME Qwen2-VL 2B model card
- ZDS 0002, *Zaxonlite: Product and Delivery Plan*
- ZDS 0003, *Zaxonlite Security and Trust Plan*
- ZDS 0004, *Zaxonlite Format and Compatibility Contract*