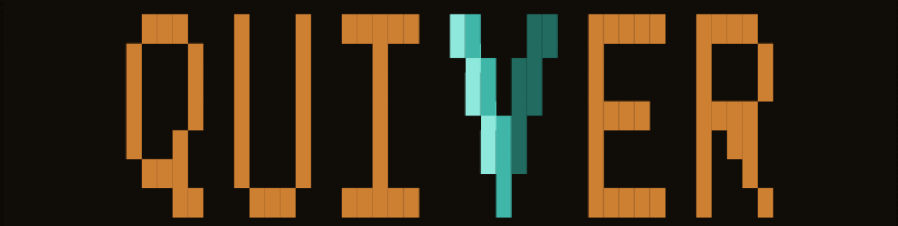

Quiver, Explained

From "what is a vector?" to a production vector database — a complete, plain-English tour of how a secure, memory-frugal ANN engine actually works.

VECTOR SEARCH · RUST ENGINE · SECURITY-FIRST · MEMORY-FRUGAL

```
quiver@cockpit:~$ ./read --audience=everyone --depth=full
```

The security-first vector database
Open-source · AGPL-3.0 · single static binary

Authored for engineers & the curious
github.com/achref-soua/quiver

00 FRONT MATTER

How to read this guide & what's inside

One document, written for two readers at once — the newcomer and the engineer.

This guide explains a real, production-grade vector database — **Quiver** — from absolute first principles all the way down to its on-disk format and cryptography. It is deliberately written so that two very different readers get value from the *same* pages.

TWO READING PATHS

If you have never touched AI or databases: read straight through. Every concept is introduced with a real-world analogy and a concrete example before any jargon appears. You can skip the dashed "Under the hood" boxes and still understand everything.

If you build software: the same sections carry the depth — algorithms, data structures, trade-offs, and the real numbers, with the design *decisions* explained, not just stated. The dashed boxes go all the way down.

We keep one running example throughout: a **movie-recommendation search** — "find me films that *feel like* Blade Runner." By the end you will know precisely what happens, step by step, when that query runs.

Contents

PART 1 FOUNDATIONS

The Big Idea: Turning Meaning into Numbers

Before any database, one question: how do you make a computer understand that two different things mean *the same thing?*

1.1 The problem with keywords

Imagine you run a movie site. A user types: *"a moody sci-fi about androids and identity."* A traditional database searches with **keywords** — it looks for the literal words "moody," "sci-fi," "androids." But Blade Runner's description might say "a replicant hunter in a neon dystopia." Zero shared keywords — yet it is a perfect match. Keyword search is **blind to meaning**.

What we actually want is **search by meaning, not by spelling**. That is the entire reason vector databases exist.

1.2 What is an "embedding"? — *the single most important idea*

> THE CORE IDEA
>
> You can turn any piece of content — a sentence, an image, a song, a product — into a list of numbers that captures its *meaning*. That list of numbers is called an **embedding** (or a **vector**).

A vector is just an ordered list of numbers, for example:

EMBEDDING

```
# a movie, as a vector of ~768 numbers
Blade Runner  →  [ 0.91, -0.20, 0.74, 0.05, 0.61, -0.42, … ]
```

Each number is a coordinate along some invisible "axis of meaning." Loosely, you can think of these axes as *"how sci-fi is it? how romantic? how violent? how hopeful?"* — except a real AI model discovers thousands of subtle axes on its own, far richer than words we would name.

The magic property: things that *mean* similar things get *nearby* numbers. Blade Runner and Ghost in the Shell end up close together; Blade Runner and Paddington 2 end up far apart. Meaning becomes **geometry**.

THE MENTAL MODEL

Picture a giant map. Every movie is a pin. Similar movies cluster into neighbourhoods — the "gritty sci-fi" district, the "feel-good comedy" district. Searching "find films like Blade Runner" becomes: **drop a pin where the query lands, and look at its nearest neighbours.** A real map has 2 dimensions; an embedding map has hundreds or thousands. The idea is identical.

Who makes these numbers? An **embedding model** — a neural network like OpenAI's `text-embedding-3`, Cohere, or an open model. You feed it text or an image; it returns the vector.

⌥ UNDER THE HOOD

Quiver is deliberately **model-agnostic**: *you* produce the embeddings with whatever model you like, and Quiver stores and searches them. It never bundles a model — a conscious choice, because embedding models change monthly and tying a database to one would age it instantly. Quiver's job starts the moment you have vectors.

1.3 "Similar" means "close" — but close *how*?

If meaning is geometry, then **similarity is distance**. Two questions follow: how do we measure distance, and which measure is right? There are three standard answers, and Quiver supports all three.

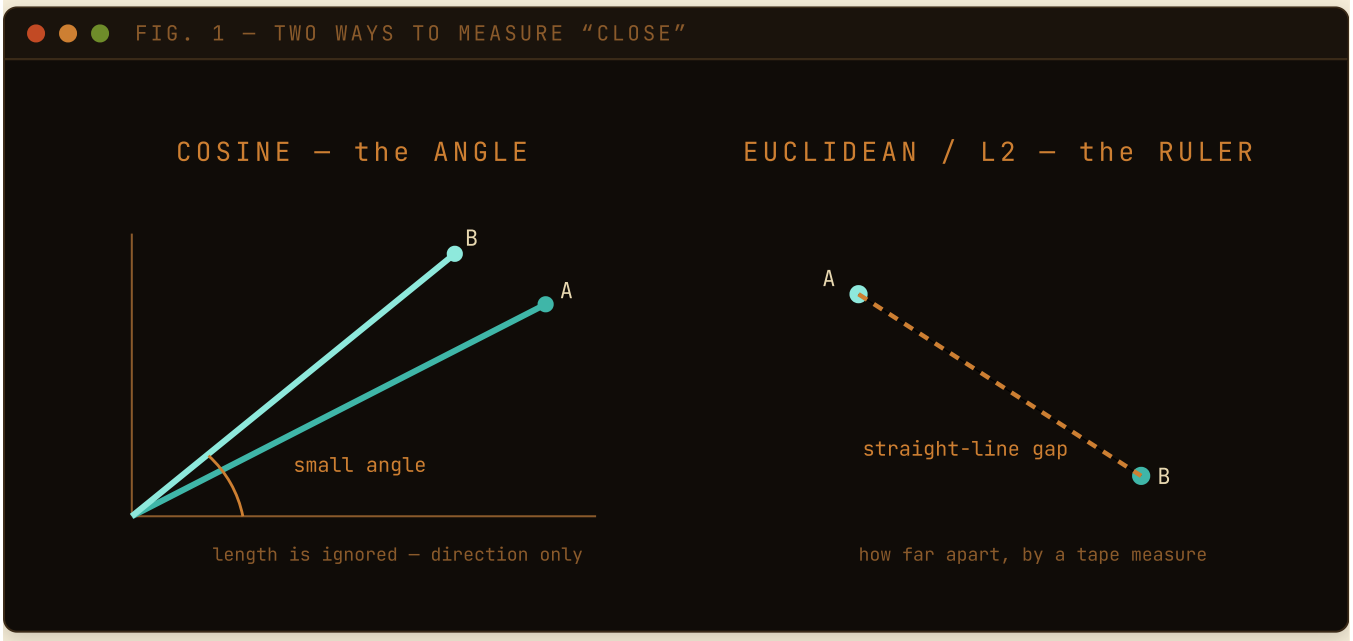

FIG. 1 — Cosine asks "do the arrows point the same way?"; L2 asks "how far apart are the points?".

Table 1 — the three similarity metrics Quiver supports

METRIC	PLAIN-ENGLISH MEANING	BEST FOR	"CLOSER" MEANS
Cosine	Do the two arrows point the *same direction*? (ignores length)	Text / semantic search — the common default	Higher (max 1.0)
Euclidean / L2	How far apart are the points, by a ruler?	Image embeddings, spatial data	Smaller distance
Dot product	Direction *and* magnitude together	Recommenders, "maximum inner product"	Higher

WHY COSINE WINS FOR TEXT

A long document and a short tweet about the same topic point the *same direction* but have very different *lengths*. Cosine ignores length and asks only "same topic?" — exactly what you want. That is why it is the default for semantic search.

⌥ UNDER THE HOOD

Quiver normalises cosine internally: it scales every vector to length 1, after which **cosine reduces to a plain dot product**. So the engine only needs fast code for two operations (dot product and squared-L2) and cosine comes for free. Internally every metric is converted to a uniform *"smaller is closer"* orientation (similarities are negated), so all the search machinery uses one comparison rule and never confuses direction.

1.4 The scaling wall: why we can't just compare everything

Naively, finding nearest neighbours is easy: compare the query to *every* stored vector, sort, take the top few. This is **brute force** (also "exact" or "flat" search). It is also a trap. With 1,000 movies it is instant. With **100 million** documents of 768 numbers each, every single query does ~76 *billion* multiplications. Per query. Hopeless for a live website.

So vector databases make a bargain:

APPROXIMATE NEAREST NEIGHBOUR (ANN)

Give up *guaranteeing* the perfect top-10, in exchange for being **100–1000× faster** while still being right ~95–99% of the time.

That trade — a tiny, controllable sacrifice in accuracy for an enormous speed-up — is the heart of every vector database. The accuracy you keep is called **recall**, and it is the number to watch.

RECALL, DEFINED ONCE

Recall@10 = "of the 10 truly nearest items, how many did the fast approximate search actually find?" Recall of 0.96 means it found 9.6 of the true top 10 on average. Brute force always has recall 1.0 but is slow; ANN trades a sliver of recall for speed. **Every benchmark in this guide is reported at a fixed recall**, because speed without recall is meaningless — you can be infinitely fast by returning garbage.

PART 2 POSITIONING

What Quiver Is, and Why It's Different

Now that the concepts are clear, here is the product — and the narrow, deliberate edge it competes on.

Quiver is an open-source vector database written in Rust. You give it vectors (plus optional metadata like `{"year":1982,"genre":"sci-fi"}`), and it answers "find the *k* most similar" queries in milliseconds — over a network API, an embeddable in-process library, or as a tool that AI agents can call directly.

There are already excellent vector databases — Pinecone, Milvus, Qdrant, Weaviate, FAISS, pgvector. Quiver does not try to beat them at raw scale. It competes on a **narrow, deliberate edge** — three things, executed well.

2.1 The wedge

1 SECURITY-FIRST, BY DEFAULT
Encryption is on out of the box — every durable byte sealed on disk. You can even encrypt vectors so the server itself never sees them. Only audited, industry-standard cryptography.

2 MEMORY FRUGALITY
Serve hundreds of millions of vectors from a laptop's RAM budget, using disk-resident indexes plus compression (~32× less memory). The headline metric is RAM-at-a-fixed-recall.

3 DEVELOPER EXPERIENCE
A single static binary. Embedded *or* server mode. A retro terminal "cockpit." Python & TypeScript SDKs. An MCP server so AI agents can drive it.

FIG. 2 — Quiver's defensible edge is the **combination** of three things, each executed well.

2.2 And — just as important — what Quiver honestly says it does *not* do

A trustworthy system is clear about its limits. Quiver states plainly:

- It is **single-node first**. It will not out-scale a distributed Milvus cluster. (Async read-replicas exist as a clearly-labelled stretch feature.)

- Client-side encryption protects **payloads, not vectors** — unless you opt into a special experimental mode (Part 6), with documented leakage.
- There is **no fully homomorphic "search on encrypted data with zero leakage"** — because no such scheme is fast enough to be practical today, and it will not pretend otherwise.

> WHY THIS HONESTY MATTERS
>
> A database is infrastructure you bet your data on. The project's README carries a striking rule: *"Every performance/memory claim is backed by a reproducible benchmark on documented hardware — until those numbers are recorded, the table stays empty rather than guess."* That discipline — never fabricating a number — is itself a feature.

PART 3 ARCHITECTURE

The Architecture, Top to Bottom

A large codebase stays understandable only when each piece has one job and the dependencies point one way.

3.1 The whole system on one page

Before the parts, the *whole*. Everything Quiver does fits in five layers, and exactly two journeys cross them: a **write** travelling down to durable disk, and a **query** travelling down to the index and back. Keep this map in mind as you read — every later section zooms into one box on it.

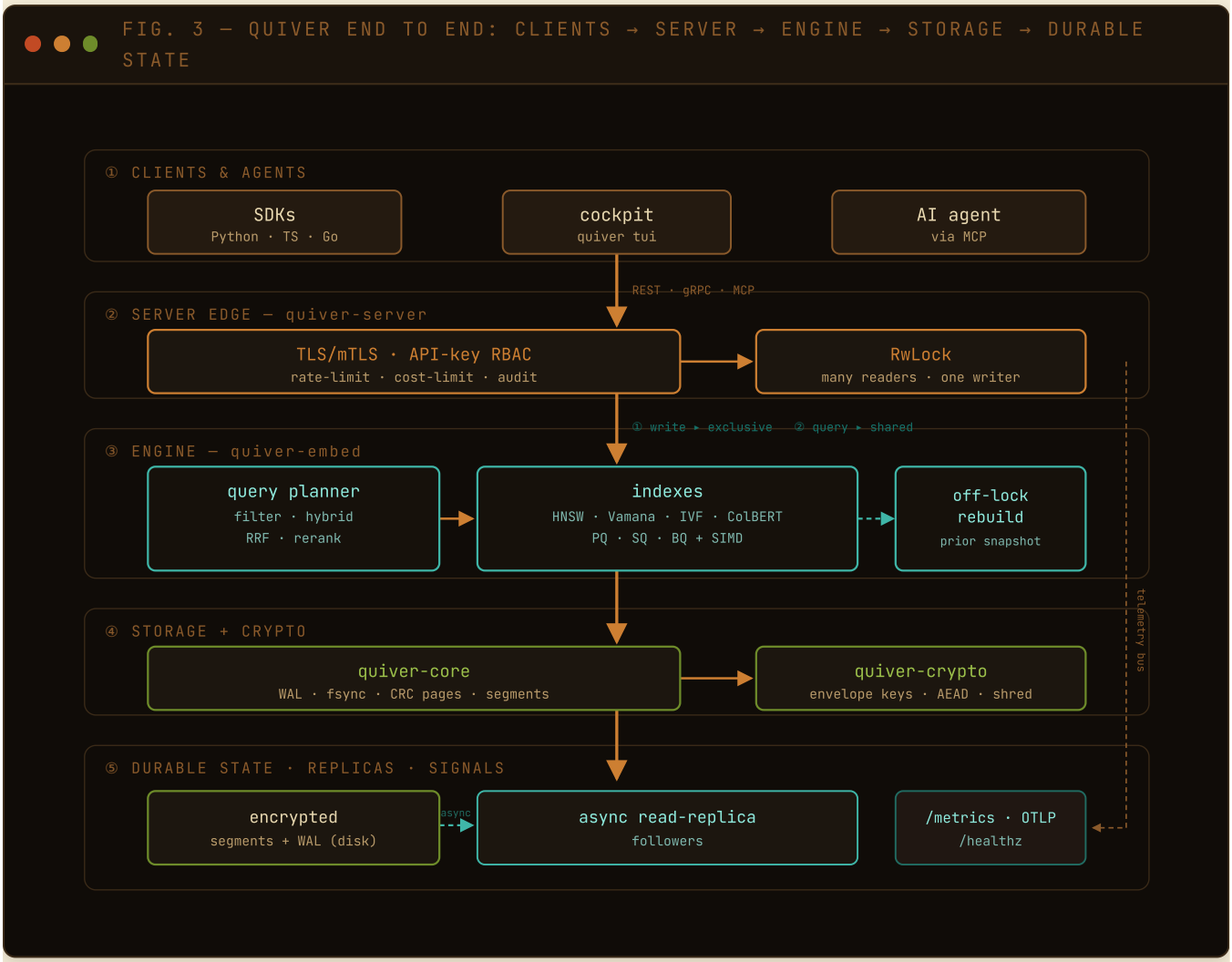

FIG. 3 — The five layers, and the two journeys that cross them. A **write** takes the exclusive lock, is appended to the WAL and fsync'd through the crypto layer to disk, *then* acknowledged; a **query** takes the shared lock (so many run at once), plans its filter, searches the index, and re-ranks against full-precision vectors. Followers replicate asynchronously; every layer emits metrics and traces.

3.2 The shape of the system

Quiver is a **Cargo workspace**: a collection of small, focused Rust libraries ("crates"), each with one job, stacked so the low-level engine pieces never depend on the high-level network pieces. The rule the arrows enforce is that **the engine knows nothing about HTTP, gRPC, or terminals** — the network code is a thin shell around the same engine the embeddable library exposes.

FIG. 4 — CRATE DEPENDENCY GRAPH (ACYCLIC BY CONSTRUCTION)

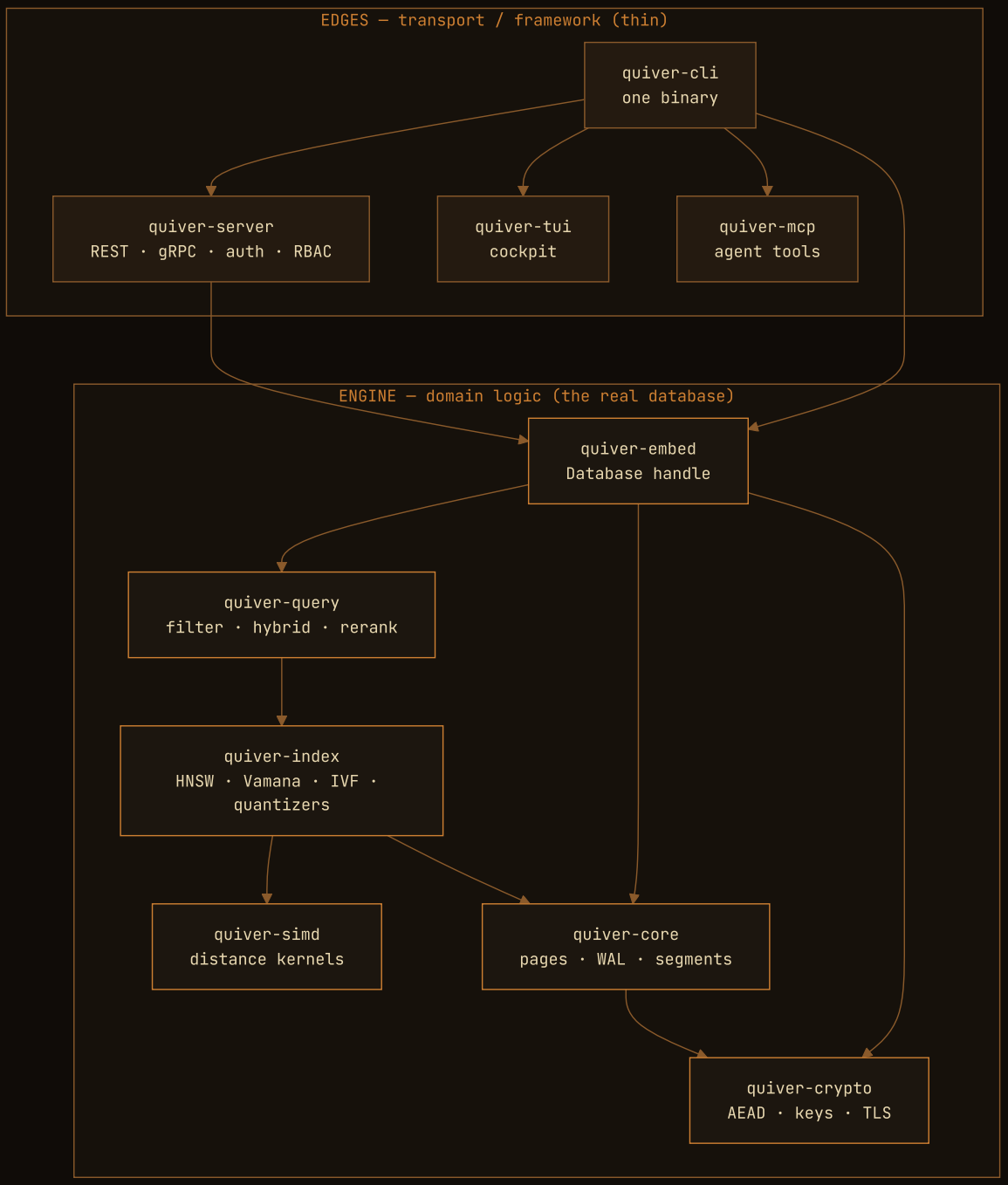

FIG. 4 — The engine (bottom) is reusable and testable in isolation; framework code stays at the edges.

⌥ UNDER THE HOOD — THE CRATE MAP

quiver-simd (~640 lines) — raw arithmetic: distance between two vectors, as fast as the CPU allows. Pure compute, no I/O. **quiver-crypto** (~2,650) — thin, careful wrappers over *audited* cryptography; never a home-grown cipher. **quiver-core** (~6,200) — the storage engine built from scratch: pages, the write-ahead log, segments, the catalog, compaction. *No embedded database (no RocksDB/SQLite/LMDB) is used.* **quiver-index** (~5,700) — the ANN indexes and the quantizers. **quiver-query** (~490) — the query planner: filtering, hybrid search, merge & re-rank. **quiver-embed** (~3,400) — stitches it into one clean `Database` API. The rest are edges. The whole thing is **Rust (edition 2024)**, AGPL-3.0, shipping as **one static binary**; a workspace lint policy even *forbids* `unwrap()` / `expect()`, forcing every error to be handled.

3.3 Two ways to run it

Table 2 — embedded library vs server

MODE	WHAT IT IS	WHEN TO USE
Embedded library	`Database::open(path)` — an in-process handle. No network, no auth surface, but encryption-at-rest still on.	Tests, notebooks, desktop apps, anything local.
Server	`quiver serve` — gRPC + REST with authentication, role-based access, multi-tenant namespaces, audit logging, query cost limits.	Production, shared services.

The same binary does both. The terminal cockpit (`quiver tui`) and the AI-agent server (`quiver mcp`) are just **clients** of the server API, so they work against a local *or* a remote Quiver.

3.4 What happens when you write, and when you search

Let us trace our movie example end to end. First, **writing a movie** (“upsert”). The crucial line is *append → fsync → durable*: the write is acknowledged only *after* it is flushed to disk.

FIG. 5 — The write is acknowledged only **after** the record is physically on disk.

Now **searching** — "films like Blade Runner, but only sci-fi after 1980." Notice the two-phase pattern that recurs *everywhere* in Quiver: a fast approximate pass to get a shortlist, then a slow exact pass on just that shortlist.

FIG. 6 — The two-phase pattern: cheap approximate narrowing, then exact precision on the shortlist.

PART 4 THE ENGINE

The Engine, Block by Block

From the bottom — raw arithmetic — up to the clever data structures that make search fast.

4.1 The speed floor: SIMD distance kernels

Every search, no matter how clever, eventually computes "how far apart are these two vectors?" thousands of times. If that one operation is slow, *everything* is slow. The naïve way multiplies element by element, 768 times. The fast way — **SIMD** ("Single Instruction, Multiple Data") — multiplies *eight numbers at once* with a single CPU instruction. It is the difference between a cashier scanning one item at a time and a scanner that reads eight barcodes in one pass.

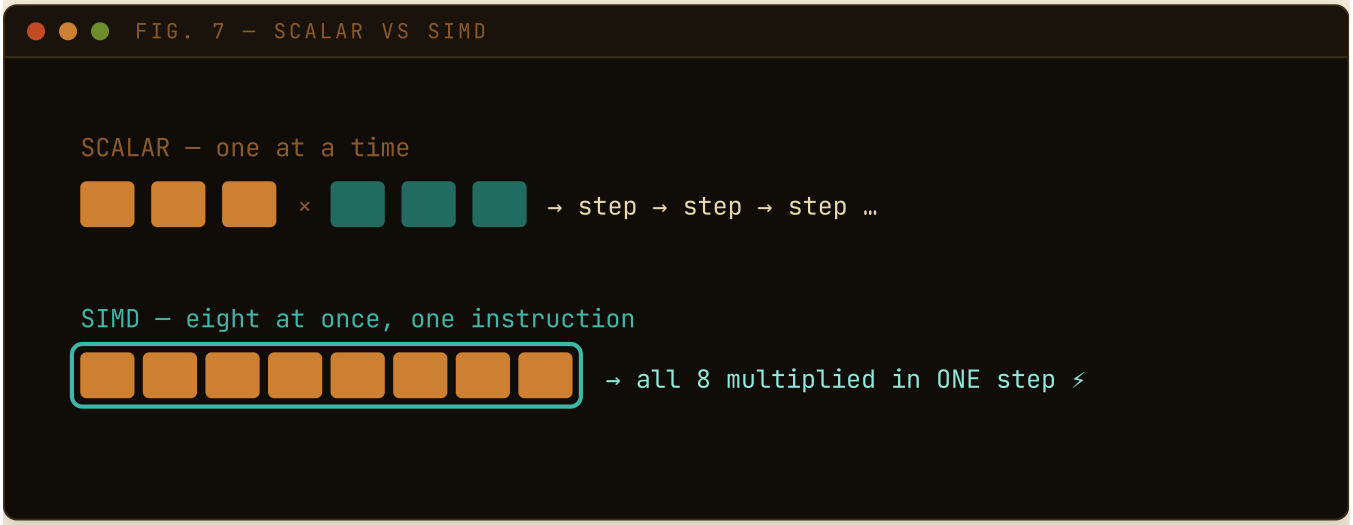

FIG. 7 — The foundational speed-up every index is built on top of.

⌥ UNDER THE HOOD

`quiver-simd` provides hand-written kernels for cosine, squared-L2, dot product (over 32-bit floats *and* 8-bit integers), and **Hamming distance** (bit-counting, for binary compression — §4.3). At runtime it *detects the CPU's features* and dispatches to the AVX2 / AVX+FMA path if present, falling back to portable scalar code otherwise — so one binary runs fast on a modern server and *correctly* on an old one. Every SIMD path is **differential-tested**: random vectors of awkward lengths (0, 1, 7, 769 — deliberately not multiples of 8, to exercise the leftover "tail") are fed to both versions and asserted equal. Fast code you cannot trust is worthless; this is how it earns the trust.

4.2 The core trick of ANN: navigable graphs

How do you find nearest neighbours among 100 million vectors *without* comparing them all? The most successful answer is a **proximity graph**.

THE IDEA

Build a network where each vector is a node, connected by "friendship" links to a handful of nearby vectors. To search, **start anywhere and keep walking to whichever friend is closer to your query, until you can't get closer.** Like finding a house by repeatedly asking "which of your neighbours lives closest to this address?" — you converge in a few hops, never visiting the whole city.

Quiver implements the two best-known proximity-graph families: **HNSW** and **Vamana (DiskANN)**, plus the cluster-based **IVF**.

HNSW — the "skip-list of maps"

HNSW (Hierarchical Navigable Small World) adds one idea to the proximity graph: **layers**, like a zoomed-out highway map laid over a detailed street map. You enter at the sparse top, take big leaps into the right region, then drop down layer by layer, refining, until the dense bottom layer where every vector lives. Coarse-to-fine, geographically.

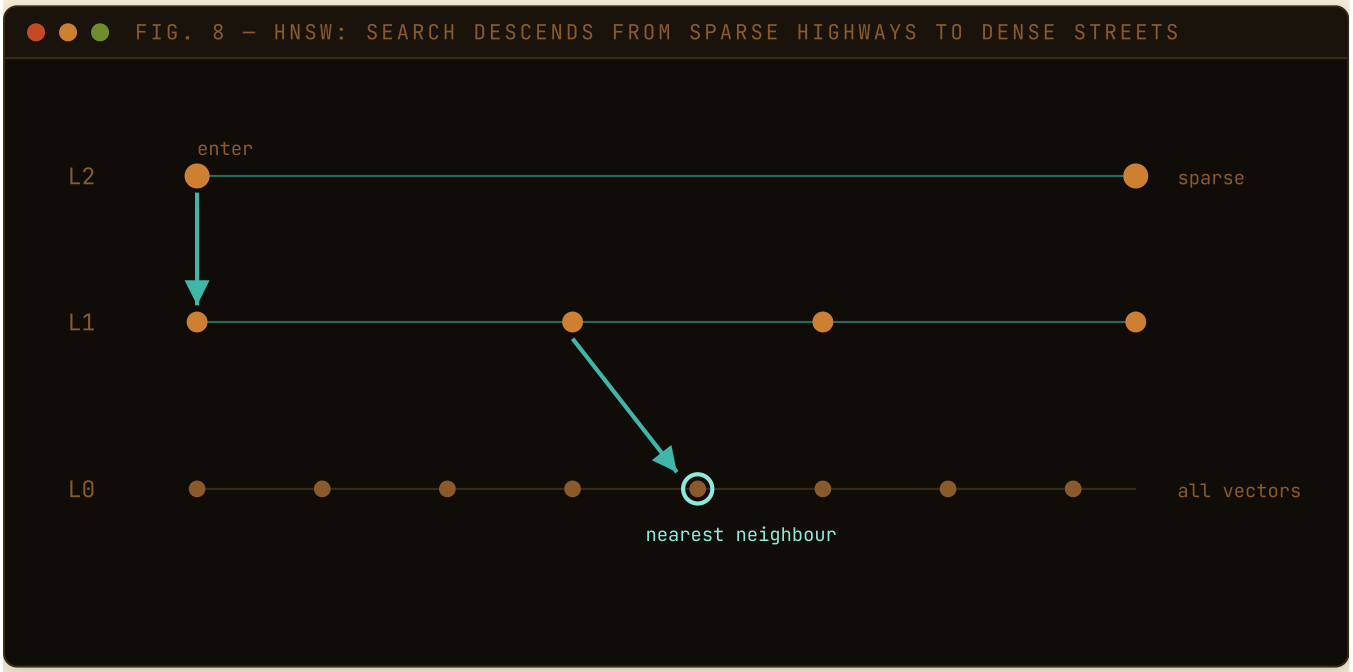

FIG. 8 — Big leaps up top, careful local search at the bottom — few comparisons, high recall.

⌥ UNDER THE HOOD — THE KNOBS

M (default 16): how many friends each node keeps — more friends, better recall, more memory. **efConstruction** (default 200): how hard it searches *while building*. **ef_search**: how hard it searches *at query time* — the size of the candidate "beam," and your live **recall ↔ speed dial** (you will see this exact knob in the benchmarks). Two refinements lift Quiver above a textbook version: (1) the **diversity heuristic** — when choosing friends, prefer ones pointing in *different directions*, so edges span the space instead of clumping, materially improving recall on clustered data; (2) **soft deletes** — a deleted point is tombstoned (kept as a navigation stepping-stone, never returned), and the search widens its beam to compensate, so recall holds even after many deletes.

Vamana / DiskANN — the graph built for *disk*

HNSW assumes the graph lives in RAM. **Vamana** (behind Microsoft's DiskANN) builds a single flat graph engineered so it can live on an **SSD** and still answer queries in very few disk reads — the key to Quiver's memory wedge. Its secret is the **RobustPrune α-slack rule**: when picking a node's neighbours it keeps the closest, then drops any candidate the chosen one is already α× closer to — forcing edges to span *long* distances, so greedy search reaches anywhere in very few hops (very few SSD page reads).

IVF — the "library card catalogue"

IVF (Inverted File) divides and conquers by neighbourhood: cluster all vectors into cells, and at query time scan only the few cells nearest the query.

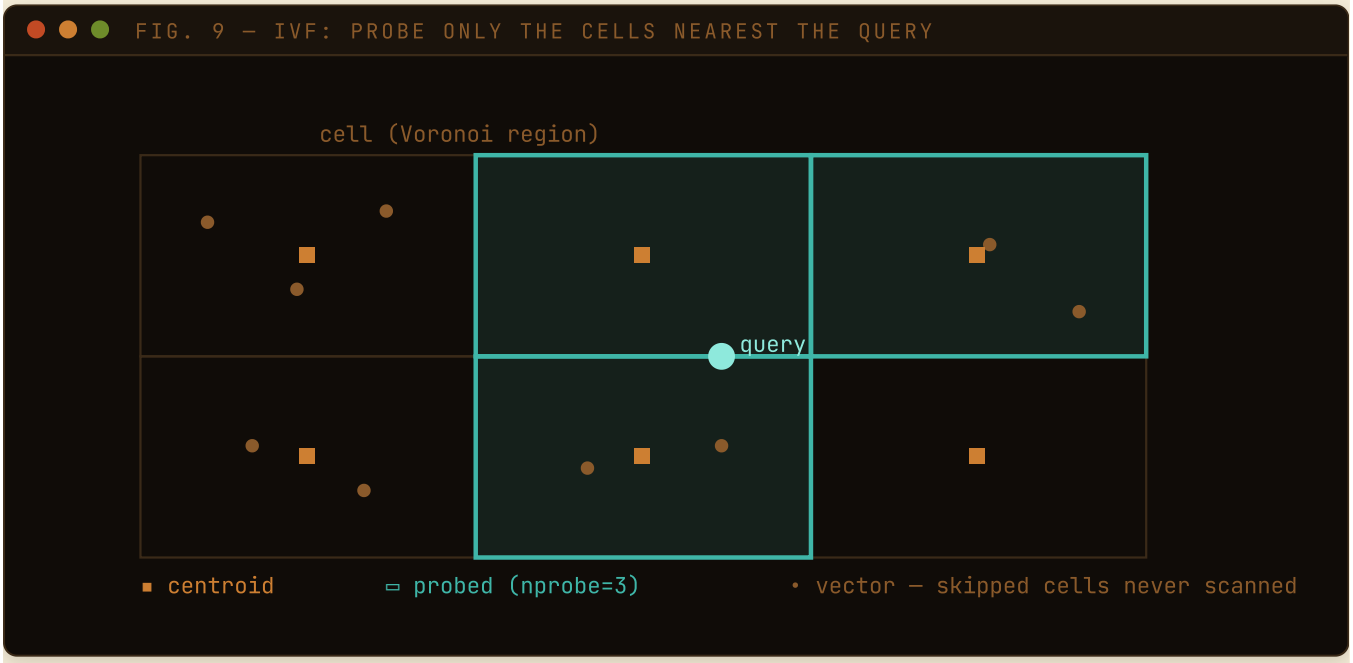

FIG. 9 — `nprobe` (how many cells to check) is IVF's recall ↔ speed dial.

Table 3 — choosing an index

INDEX	LIVES IN	STRENGTH	USE WHEN
HNSW	RAM	Highest QPS at high recall	You have the RAM and want raw speed
Vamana / DiskANN	SSD (compressed in RAM)	Tiny RAM footprint	Datasets bigger than your RAM budget
IVF	RAM or SSD	Fast build, predictable memory, easy updates	Streaming data, simpler tuning

⌥ UNDER THE HOOD — ALL THREE UPDATE INCREMENTALLY

A naïve ANN index rebuilds from scratch on every write, which is fatal for streaming data. Quiver's three families all avoid that: HNSW inserts and **soft-deletes** in place (tombstones, §4.2); IVF uses **SpFresh-style LIRE** rebalancing so cells split and merge as data streams in; and the Vamana graphs use **FreshDiskANN's StreamingMerge** — a read-only base graph plus a small in-memory delta graph and an O(1) deletion set, searched together and consolidated only past a churn threshold. So graph writes are size-independent, the index stays a *derived* structure (the WAL is the source of truth, so the crash gate is untouched), and the only rebuild that ever happens is the deferred, off-lock one of §5.4.

4.3 Quantization: the memory-frugality superpower

One million 768-d vectors at 4 bytes each is **3 GB** of RAM just for the vectors; ten million is **31 GB** — it does not fit on a laptop. **Quantization** is lossy compression for vectors: shrink each into a tiny "code," search on the codes, accept slightly fuzzy distances.

Table 4 — Quiver's three quantizers

QUANTIZER	CODE SIZE	COMPRESSION	HOW IT WORKS
Scalar (SQ)	dim bytes	~4×	Each number stored as an 8-bit int, not a 32-bit float
Product (PQ)	**m bytes**	**up to 32×**	**Split into chunks; replace each with the ID of its nearest learned prototype**
Binary (BQ)	dim/8 bytes	~32×	Keep only the *sign* of each number (1 bit); compare by fast bit-counting

Product Quantization, simply. Imagine describing a face not with exact measurements but as "nose type #7, eyes type #3, jaw type #12." If everyone shares a catalogue of types, those three small numbers reconstruct the face approximately. PQ learns a *codebook* of prototype chunks, then stores each vector as a handful of prototype IDs — a 3072-byte vector becomes ~96 bytes, **32× smaller**. The genius that makes lossy compression safe:

THE APPROXIMATE-THEN-RE-RANK PATTERN — THE ENGINE'S GOLDEN RULE

1. Rank cheaply on the tiny codes. Use the compressed vectors to quickly find the top 100–800 candidates. Fast, but fuzzy.

2. Re-rank precisely on the originals. Fetch the full-precision vectors for only that shortlist, compute exact distances, return the true top 10. You recover almost all the recall "lost" to compression.

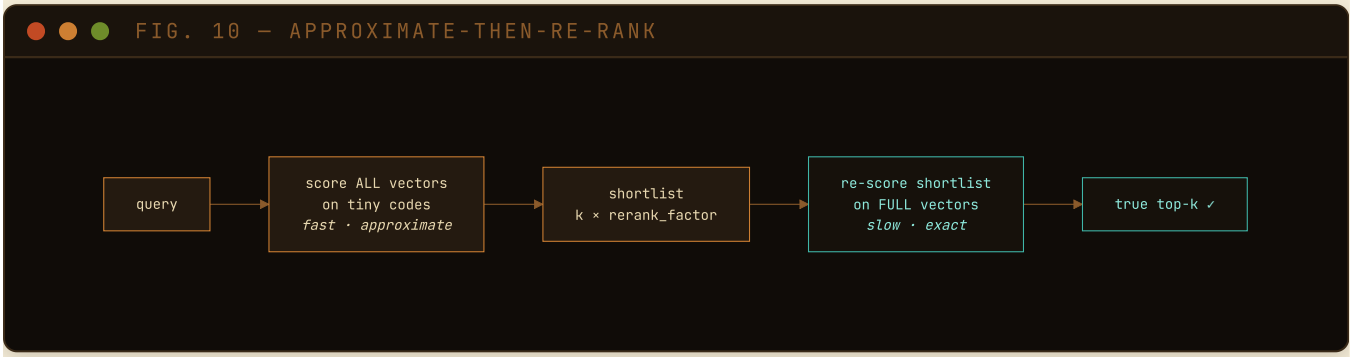

FIG. 10 — The shortlist depth (`rerank_factor`) is the master recall ↔ latency dial. A deeper pool can only *add* true matches.

⌥ UNDER THE HOOD

PQ uses **asymmetric distance computation (ADC)**: the query stays full-precision and a small lookup table is precomputed once per query, so scoring each code is a few table lookups and adds — extremely fast. Binary quantization leans on the SIMD **Hamming kernel** from §4.1: XOR the sign-bit codes, count differing bits, which a CPU does blisteringly fast — an ideal coarse pre-filter before the exact re-rank.

4.4 Putting it together: the disk-resident path *(the "32×" headline)*

Combine the **Vamana graph** (built for SSD) with **Product Quantization** (32× smaller), and you get Quiver's signature feature.

FIG. 11 — THE DISK-RESIDENT SERVE PATH

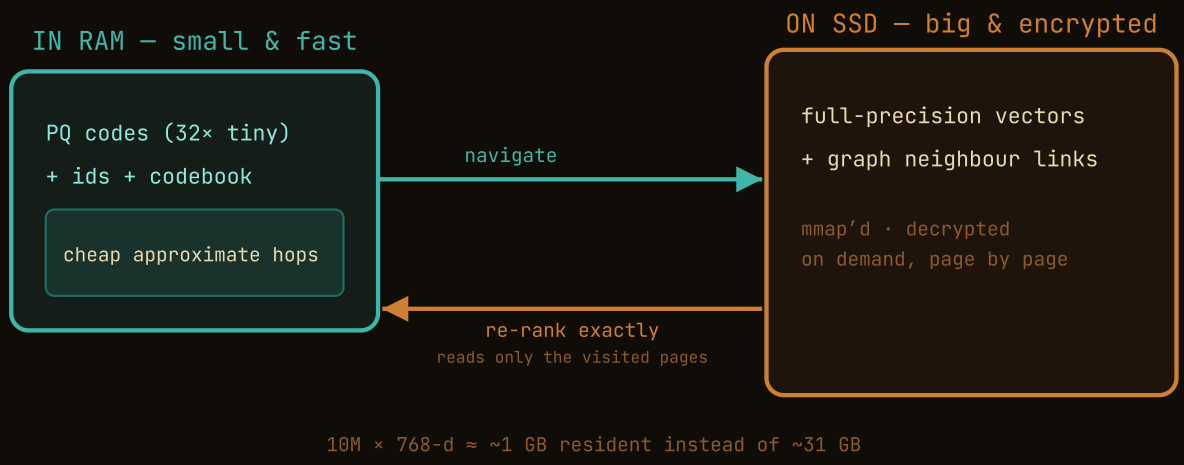

FIG. 11 — Only tiny PQ codes stay in RAM; full vectors and the graph live encrypted on SSD, touched only when visited.

The result, in Quiver's measured terms: a dataset serves from **roughly its PQ-code footprint** instead of the full vectors. For a 10M × 768-d collection that is about **~1 GB resident instead of ~31 GB** — exact arithmetic, and why "hundreds of millions of vectors on a laptop" is a real claim, not marketing.

The in-memory floor — and what v0.23.0 fixed. "Roughly the PQ-code footprint" is a *floor*, not zero: what stays resident is the **PQ codes + the id map + the PQ codebook + a small in-memory FreshDiskANN delta** (recent writes not yet folded into the base graph); the full-precision vectors and base graph are `mmap`'d on disk and demand-paged. A subtlety undercut the claim *in practice* until **v0.23.0**: a restarted server **rebuilt the disk index from every full-precision vector on open**, so its post-restart RSS briefly looked like the in-memory path. The **durable on-disk DiskVamana index** (ADR-0063) fixes that — the server `mmap`s its frugal base and replays only the post-checkpoint WAL tail on open (falling back to the authoritative rebuild on any mismatch, so the `kill -9` gate is untouched) — so it serves from the resident-PQ floor *immediately after a restart*, and the wedge benchmark now **cold-reopens the server before sampling RSS** to measure exactly that floor.

4.5 Filtering: search by meaning *and* by rules

Real queries are hybrid: "films like Blade Runner, **but only sci-fi released after 1980**." That is a similarity search **plus** a structured filter on metadata. Quiver expresses filters as a **typed predicate tree** you can nest arbitrarily (`eq, ne, in, lt, lte, gt, gte, exists` plus `and/or/not`, over dot-paths like `user.age`):

FILTER — JSON WIRE FORM

```
{ "and": [
    { "eq":  { "field": "genre", "value": "sci-fi" } },
    { "gt":  { "field": "year",  "value": 1980 } },
    { "not": { "eq": { "field": "rating", "value": "G" } } }
] }
```

The interesting part is *how* it runs the filter — the planner picks one of two strategies:

STRATEGY	WHAT IT DOES	CHOSEN WHEN
Pre-filter	Find rows matching the filter first (via a secondary index), then search similarity only over those	The filter is **selective** (few rows) — e.g. `user_id = 42`
Post-filter	Search similarity first, then drop results that fail the filter	The filter is **broad** (many rows) — e.g. `year > 1980`

> WHY BOTH
>
> Pre-filtering a *broad* filter wastes time building a huge candidate set; post-filtering a *selective* filter risks all candidates being filtered out, leaving nothing. The filter is **re-checked on every surviving result**, so the answer is always exact — the planner only affects speed, never correctness.

4.6 Hybrid search: combining meaning with keywords (RRF)

Sometimes pure semantic search is not enough — someone searching a product code like "SKU-4417" wants that *exact* term, not "vibes." **Hybrid search** runs both a **dense** (embedding) search and a **sparse** (keyword/term-weight) search, then merges the two ranked lists. But their scores live on incomparable scales. The elegant fix is **Reciprocal Rank Fusion (RRF)**: ignore scores, use only *rank*.

RECIPROCAL RANK FUSION

```
# each list gives a doc  1/(k0 + rank)  points; sum across lists
RRF(doc) = Σ  1 / (k0 + rank_in_list)      (k0 = 60, the standard constant)
         over each result list
```

Because it is purely rank-based, RRF needs **no score normalisation** — which is exactly why it is the robust, industry-standard fuser. A document ranking high in *both* lists wins. A sparse vector (e.g. from a SPLADE model) rides inside the point's payload, so enabling hybrid search needs *no change to the on-disk format*.

> ⌥ UNDER THE HOOD — MULTI-VECTOR / COLBERT
>
> Quiver also supports **late-interaction (ColBERT-style)** retrieval: a document is stored as a *set* of token vectors and ranked by **MaxSim** (for each query token, find its best-matching document token, then sum). Cleverly, each document is modelled as a *group of ordinary rows* in the same storage engine — so there is no new on-disk format and the crash-safety guarantees are untouched. A ColBERT corpus (many small token vectors) is exactly the pool the IVF+PQ compression path was built for.

PART 5 DURABILITY

How It Survives `kill -9`

A database has one sacred promise: if it told you a write succeeded, that write is not lost — not to a crash, not to a power cut.

5.1 The Write-Ahead Log (WAL)

The idea is older than databases and simple: **write down what you are about to do *before* you do it.** Like a ship's logbook — before changing course, log the new heading. If interrupted, the log tells you exactly where things stood. In Quiver, every mutation is encoded, **appended** to the WAL, **fsync'd** (forced all the way to physical disk, not just the OS cache), and *only then* acknowledged. That ordering is the entire guarantee: once you get "ok," the record is physically on disk. The index update happens *after* the ack — and if the machine dies first, recovery replays the log to redo it.

5.2 Catching corruption: checksums and torn writes

What if a crash happens *mid-write*, leaving a half-written record? Or a disk bit silently rots? Every WAL record is framed with a length prefix and a **CRC32C checksum**:

ON-DISK FRAMES & PAGES

```
WAL frame:
  [ length:u32 ][ CRC32C:u32 ][ … the (encrypted) record bytes … ]
                       |
        recovery reads frames until one FAILS its length/CRC check
        — the signature of a crash mid-append — and treats
        everything after it as "never happened" (point-in-time recovery)

Data page (16 KiB):
  [ magic | version | type | page_id | lsn | payload_len | CRC32C ]   ← 32-byte header
  [ …data… | zero-padding to 16 KiB ]
```

FIG. 12 — Length + checksum framing turns a torn tail into a clean, detectable boundary.

On recovery, Quiver reads frames until one **fails its check** — the unmistakable signature of a crash mid-append — and treats everything from that point on as "never happened." Because the log is append-only and every record was fsync'd before its ack, the *only* place a broken frame can legitimately appear is the very tail. A torn trailing record was, by definition, never acknowledged, so discarding it loses nothing. The same discipline applies to the 16 KiB data pages: each carries its own CRC, so **corruption is detected on read and never silently served**.

THE BOTTOM LINE

Acknowledged writes survive `kill -9` and power loss; corruption is always *detected* rather than served as a wrong answer; and all of this holds **whether or not encryption is on**, because the checksums guard the plaintext path and encryption sits on top. Quiver's test suite literally kills the process with `kill -9` mid-operation and asserts the data is intact on restart — the "crash gate."

5.3 Many readers, one writer

Durability is about *not losing* writes; throughput is about *serving* reads. Quiver is **single-writer, many-reader**: the server guards the engine with a reader–writer lock (ADR-0057). A search takes the *shared* lock, so many searches run **in parallel** — read throughput scales with cores instead of serializing on one mutex. A write takes the *exclusive* lock; the single-writer model, and everything in §5.1–5.2, is unchanged.

There is one subtlety, and it is the whole story of this section. Some writes — an HNSW in-place update, a bulk load, a delete, a replicated write — cannot be absorbed into the index in place, so the engine **defers** the rebuild: it marks the collection stale and leaves the prior, still-valid index in place. The question is what the *next* reader does about it. The naïve answer (what v0.21.0 shipped) is: take the exclusive lock and rebuild before serving. That is correct, but it has a brutal failure mode — measured below.

5.4 The rebuild that doesn't stop the world

Rebuilding an ANN index is not cheap: it is roughly a single-threaded build pass over the whole collection. If a reader does it *under the exclusive lock*, every other reader blocks for the entire build. We measured exactly that, with a reproducible harness (`mvcc_measurement.rs`, ADR-0062):

Table 5 — the deferred-rebuild reader stall, before the fix (dev box · WSL2 · HNSW · dim 128 · indicative)

COLLECTION SIZE	SINGLE-THREAD REBUILD	STEADY READ P99 (CONCURRENT)	READER STALL DURING REBUILD
20 000 vectors	7.3 s	422 µs	8.1 s
50 000 vectors	26.7 s	379 µs	29.7 s
100 000 vectors	**68.7 s**	**408 µs**	**76.6 s**

The stall tracks the rebuild duration and is **four to five orders of magnitude** above the steady-state p99 — borne by *every* read that arrives during the window, and worse at scale (a 1 M disk-Vamana rebuild is minutes). So the fix is not a micro-optimization; it is the difference between a database and a doorstop under write-then-read load.

THE FIX — REBUILD OFF THE EXCLUSIVE LOCK (ADR-0062)

Serve the prior snapshot, build with no lock held, swap at the end. A stale reader returns results from the *prior* index (still a valid graph over the prior ids) instead of blocking. Meanwhile one rebuild is driven off-lock: its inputs are captured under the *shared* lock (other reads continue), the new index is built holding **no lock at all**, and only the final pointer-swap takes a brief exclusive lock. A per-collection **write-generation counter** guards against a write that lands mid-build: if the generation moved, the collection stays stale and another rebuild is scheduled — so no write is ever lost.

FIG. 13 — The seconds-long stall (top) collapses to the cost of serving the prior snapshot plus a brief swap (bottom). No `unsafe`, no lock-free data structure — just `Arc` and the existing `RwLock`.

VISIBILITY, NEVER DURABILITY

This moves read *visibility*: a server read may briefly miss a write committed a millisecond ago (snapshot isolation, sanctioned by ADR-0053), but it never sees a half-applied one, and the WAL-fsync acknowledgement of §5.1 is byte-for-byte unchanged. Embedded `&mut` callers still rebuild synchronously, so an in-process program always reads its own writes. That fixed the rebuild's *lock scope*; §5.5 closes the remaining gap — the *write's* lock window.

5.5 Reads that never wait on a writer — lock-free MVCC

The off-lock rebuild solved the worst case. But even an *ordinary* write has to briefly lock the index to record itself — and a "lock" here means exactly what it sounds like: while the writer holds it, every reader waits at the door. The window is short (milliseconds), but it adds up. We measured how much, by running searches and writes at the same time and asking *what fraction of its read speed does the database keep?* (1.0× would mean

writes cost reads nothing.) The answer was sobering: **one** steady writer dropped reads to about a tenth of their speed, and a **second** writer collapsed them to almost nothing — the readers spend their time waiting at the door, not searching.

The fix is a classic database trick called **MVCC — multi-version concurrency control**. Picture a librarian reorganising a shelf. The slow way: she closes the whole reading room while she works, and everyone waits. The MVCC way: she leaves the current shelf untouched for readers, prepares an updated copy off to the side, and swaps it in the instant it's ready — readers never stop, and never catch the shelf half-finished. Quiver does exactly this with a search index (ADR-0064), behind an off-by-default `QUIVER_MVCC_READS` switch.

> KEEP A FROZEN COPY; KEEP RECENT EDITS ON A STICKY NOTE
>
> The writer keeps the last fully-built index frozen as a **snapshot**. New writes that arrive afterwards don't disturb it — they go onto a small **overlay**, like a sticky note that says *"also include these few new points, and ignore those deleted ones"* (a deleted point is marked with a **tombstone** — a note that it's gone, rather than erasing it on the spot). A reader picks up the current snapshot in a single, instant step that needs no lock — that lock-free hand-off is what the `arc-swap` library gives us — then searches the frozen index, glances at the short sticky note, and combines the two. Because the note stays small, the writer can rewrite it cheaply on *every* write; once it grows past a threshold, the next off-lock rebuild folds it into a fresh frozen copy and the note starts blank again. Old copies tidy themselves up: Rust counts how many readers still hold each one and drops it automatically when the last reader walks away — no manual cleanup, no `unsafe` code. And a reader always sees one whole, consistent version — never a write applied halfway.

This is split by *what the read needs*. A plain **vector search** (just "find the nearest points") needs only the snapshot, so it runs entirely lock-free and never waits on a writer. A search that also returns each point's stored data — its payload, or a filter on it — still has to read the main store, which isn't safe to read while the writer is changing it, so those reads take the shared lock as before (but still answer from the snapshot). The common, hot path — nearest-neighbour search — is the one that goes fully lock-free.

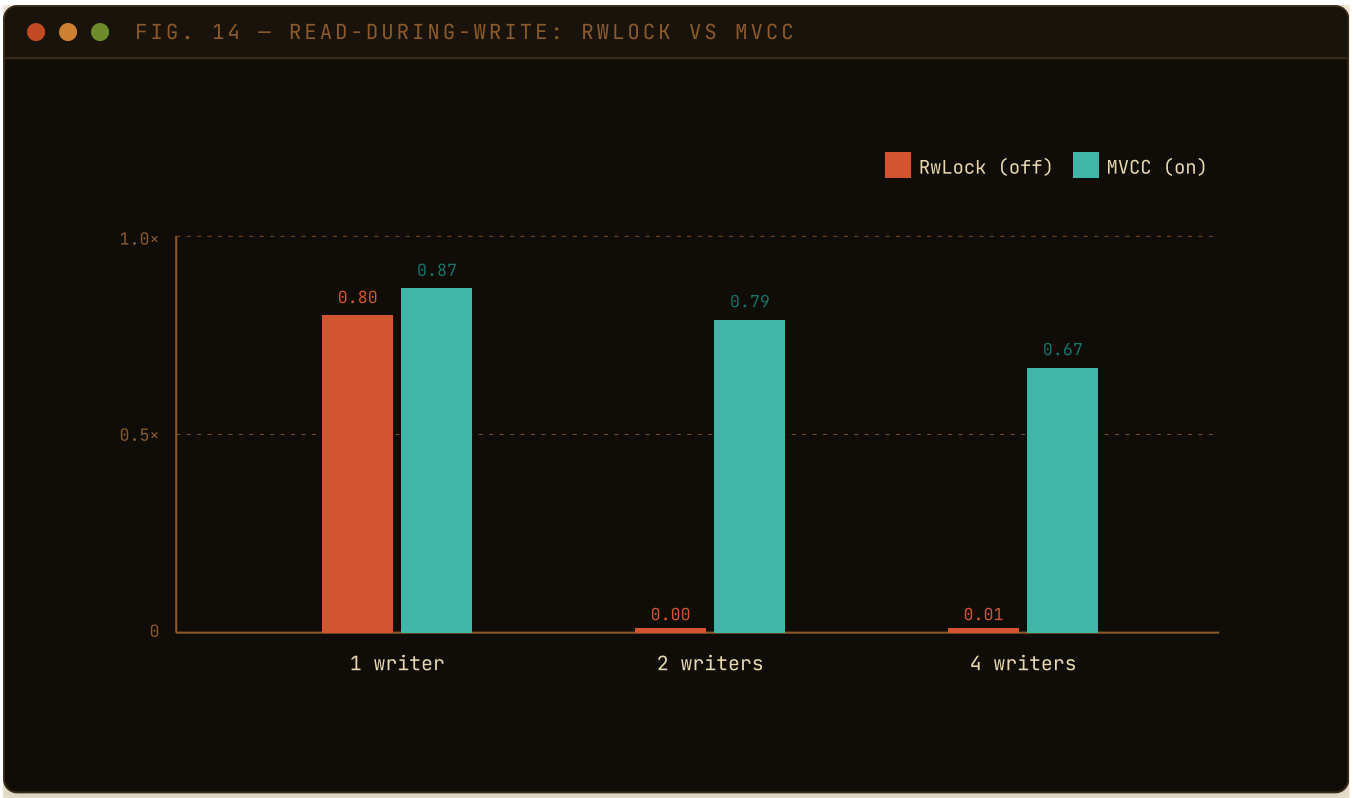

FIG. 14 — Retained read-QPS at batch 1 (the worst case for the lock), off → on. The multi-writer collapse is gone: under the `RwLock` a second small-upsert writer starves readers to ~0; lock-free MVCC holds ~0.67×–0.79×. The *ratio* on identical hardware is the proof; absolute QPS is reference-hardware-pending.

The full grid, `QUIVER_MVCC_READS` **off → on** (retained read-QPS, 1.0× = no penalty):

Table 6 — read-during-write, RwLock vs MVCC (SIFTSMALL, dev box, same sweep)

READERS VS.	BATCH 1 (OFF→ON)	BATCH 64 (OFF→ON)	BATCH 512 (OFF→ON)
1 writer	0.80× → 0.87×	0.66× → 0.71×	0.50× → 0.74×
2 writers	**0.00× → 0.79×**	0.35× → 0.75×	0.30× → 0.75×
4 writers	**0.01× → 0.67×**	0.06× → 0.69×	0.07× → 0.71×

Durability is untouched: MVCC changes *visibility, not durability* — the overlay is derived from the same WAL the store replays, and the `kill -9` gate holds by construction. It ships **opt-in**: the proven `RwLock` path stays the default until the win is confirmed on dedicated hardware (on a shared dev box only the ratio is honest; the absolute QPS is reference-hardware-pending).

PART 6 SECURITY

Security, the Defining Feature

This is where Quiver makes its strongest claim. We go from the disk outward.

6.1 Encryption at rest — *on by default*

Most databases make you opt *in* to encryption. Quiver makes you opt *out* — and won't let you, on a non-loopback bind. Out of the box the server demands a 256-bit master key and **seals every durable byte** — segments, the manifest, *and* the write-ahead log — with **XChaCha20-Poly1305**, a modern, audited authenticated cipher. "Authenticated" (AEAD) matters: it not only hides data, it **detects tampering** — flip a single bit and decryption fails loudly instead of returning subtly wrong data.

> ⌥ UNDER THE HOOD
>
> Only audited cryptography is used — RustCrypto's AEAD/KDF and `rustls` (backed by the audited `ring` library) for TLS. **No home-grown primitives, ever** — the cardinal rule of applied cryptography. Key material is wrapped in zeroizing types so it is scrubbed from memory when dropped.

6.2 Envelope encryption + crypto-shredding

Instead of encrypting everything with the one master key, Quiver uses a **two-level key hierarchy**. Each collection gets its own random **Data-Encryption Key (DEK)**, itself stored encrypted ("wrapped") under the master key.

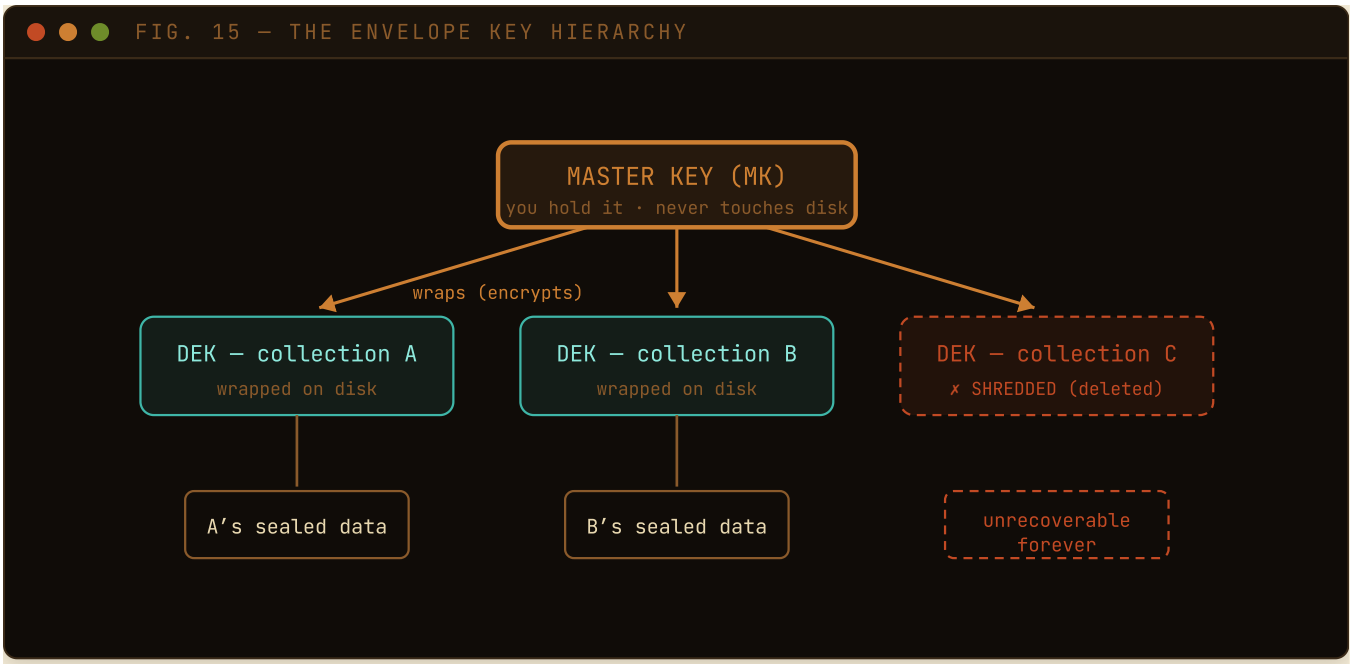

FIG. 15 — Per-collection keys, each wrapped by the master key — the structure that makes instant, provable deletion possible.

> CRYPTO-SHREDDING — INSTANT, PROVABLE DELETION
>
> To permanently delete a collection, Quiver does not overwrite gigabytes. It just **deletes that collection's tiny wrapped DEK file.** The DEK existed nowhere else. Once gone, the collection's encrypted bytes are **mathematically unrecoverable — even by the holder of the master key, even from a backup tape that still has the ciphertext.**

This is the gold-standard pattern for the GDPR "right to erasure": you can *prove* deletion happened (the key is provably gone) without trusting that every copy on every disk and backup got physically scrubbed. Quiver's tests demonstrate exactly this — seal a page, shred the collection, then show a fresh key-ring with the correct master key can no longer decrypt it.

6.3 In transit and access control

- **TLS / mTLS:** encrypted connections required for any non-loopback bind; optional mutual TLS makes clients prove identity with a certificate too.
- **Default-deny RBAC:** access by scoped API key, with a **role** (`read ⊆ write ⊆ admin`) and a **collection scope** (exact names, or an `acme.*` prefix for per-tenant isolation). Over-reach returns `403`; listing even *hides* out-of-scope collections.
- **Append-only audit log:** every mutating/admin operation and every denial is recorded — who, what, which resource, what outcome — but **never the secret**.
- **Query cost limits:** the server caps how expensive a single query can be, closing off "authenticated denial-of-service."

6.4 The frontier: encrypting the *vectors* themselves

By default, client-side encryption protects **payloads** (metadata): you can seal `{"ssn":"…"}` with a key the server never sees, while leaving `{"genre":"sci-fi"}` cleartext so the server can still filter on it. But can a server rank vectors it cannot read? Quiver offers two honest, opt-in answers — and is scrupulous about the trade-offs, because **no scheme gives fast server-side ranking, zero leakage, and good performance all at once.**

Table 7 — the two encrypted-vector modes

MODE	SERVER SEES	SERVER RANKS?	SECURITY	HONEST COST
client_side	Only opaque ciphertext + a zero placeholder	No — learns *nothing*	Strongest (IND-CPA)	Client fetches the (pre-filtered) set and ranks locally
dcpe (experimental)	Ciphertext it can compare by approximate L2	Yes — without the plaintext or key	Weaker — not semantically secure	Leaks the approximate distance-ordering *by design*; L2-only

The DCPE mode implements a *published* academic scheme ("Scale-And-Perturb" distance-comparison-preserving encryption) built only from audited primitives, with the paper's hardening steps; the docs tell you to read the threat model before using it.

> THE SECURITY-FIRST ETHOS IN MINIATURE
>
> A lesser project would ship DCPE and call it "encrypted search," full stop. Quiver ships it behind an experimental flag, names the exact academic paper, and spells out precisely what it leaks and which attacker breaks it. That candour is the point.

6.5 Not just claimed — *audited, fuzzed, and scanned*

"Security-first" is easy to put on a README. Quiver backs it with evidence, and writes down what it finds — including what it *doesn't*.

- **A written security audit.** The codebase is reviewed OWASP-style against the threat model — broken access control, injection, SSRF, cryptographic failures, misconfiguration, denial-of-service. Every finding is **fixed and pinned by a regression test** (one that fails before the fix and passes after); every control that turns out *not* to be vulnerable is recorded with the reason.
- **A real penetration-testing tool.** A live, encrypted, authenticated server is scanned with **OWASP ZAP** — the industry-standard web-application security scanner — running its full *active* rule set: SQL injection, OS-command injection, cross-site scripting, server-side template injection, XML external entities, path traversal, the padding oracle, cloud-metadata exposure, and more. The v0.29.0 scan came back **clean — zero failures across 119 rules**; the only two findings (a pair of missing hardening response headers) were fixed and the scan re-run to **zero warnings**.
- **Fuzzing the dangerous parts.** The code that touches attacker-controlled bytes — the search-filter wire format and the on-disk page/WAL decoders — is continuously **fuzzed**. The latest pass ran **tens of**

millions of randomized inputs with zero crashes: malformed input is rejected with a typed error, never a panic.

- **A locked-down supply chain.** Every dependency is pinned and scanned (`cargo deny`, `cargo audit`) with no blanket suppressions, and static analysis (CodeQL) runs on every change.

THE HONEST CORE

None of this claims Quiver is "unhackable" — no serious project would. It claims something more useful: the security is *measured*, the results are *written down*, every real finding is *fixed with a test that proves it*, and the residual risks are *stated plainly* rather than buried.

PART 7 EVIDENCE

Benchmarking Quiver — Method, Results, Verdict

Performance claims are worth only the methodology behind them. Here is how we measured, what we found, and what it proves about Quiver's value.

7.1 Why we benchmark at a fixed recall

It is trivial to make a vector search "fast" — just return less-accurate results. So a throughput number on its own is meaningless. Every figure here is reported **at a fixed recall bar** (recall@10 ≥ 0.95 unless noted): the systems are tuned until they all find ~the same fraction of the true nearest neighbours, and only *then* is their speed compared. That is the only honest way to compare ANN engines.

7.2 The method

- **Same box, every system.** An `ann-benchmarks`-style harness runs each database on one machine — an Intel i7-12700H laptop, 20 threads, 15.5 GB RAM — so the comparison is apples-to-apples.
- **Seven competitors:** FAISS, Qdrant, Milvus, Chroma, pgvector, LanceDB, Weaviate. Milvus is run as the full *server* (Docker), not the in-process Lite build.
- **Honest caveats, stated up front:** FAISS/Chroma/LanceDB run *in-process*, so their reported RAM is inflated by the Python harness (not directly comparable to the isolated-server numbers); Qdrant memory-maps vectors to disk by default (so its RAM looks tiny); and Quiver's "build" time is the *bulk-ingest* path (`points:bulk`) measured as honest *time-until-queryable* — ingest plus the deferred index build forced by the first query, the same thing every competitor's build column captures.
- **Never fabricate.** Absolute RSS on reference hardware and the 10M-scale disk path are explicitly marked "pending" rather than guessed.

7.3 Quiver's own recall ↔ speed curve (SIFT1M, 1M × 128-d, in-memory HNSW)

This single table shows the central trade-off of the whole field: as you turn the `ef_search` dial up, recall climbs and throughput falls.

Table 8 — Quiver SIFT1M, HNSW (M=16, efC=200), single thread

EF_SEARCH	16	32	64	128	256
recall@10	0.793	0.895	0.958	0.986	0.995
QPS (1T)	1539	1424	1222	955	701
p95 latency (ms)	0.8	0.8	1.0	1.3	1.7

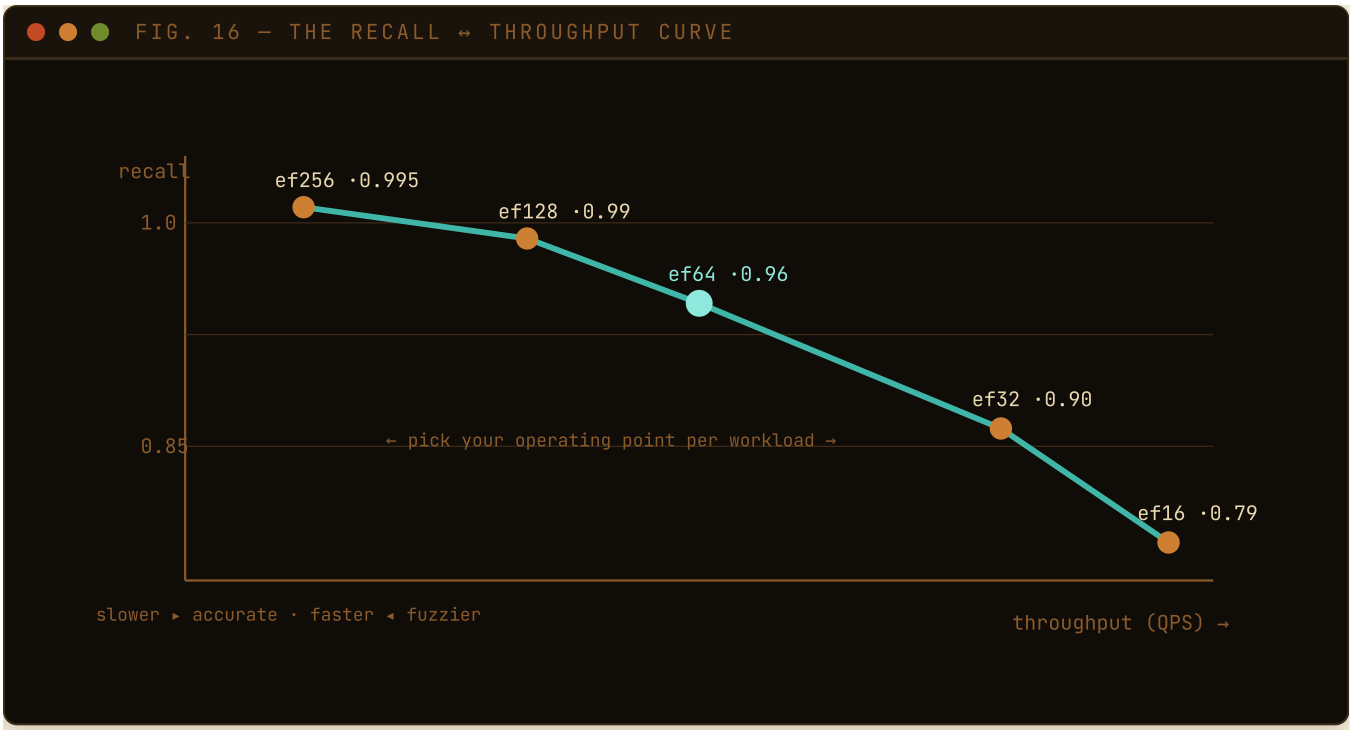

FIG. 16 — RAG pipeline that re-ranks anyway? Run at ef=64. Legal discovery where a miss is unacceptable? ef=256.

7.4 Head-to-head (SIFT1M, peak single-thread QPS at recall@10 ≥ 0.95)

Table 9 — every system on the same box; [1] in-process RSS inflated · [3] Qdrant disk-backed

SYSTEM	RECALL@10	QPS (1T)	P95 (MS)	RSS (MB)	BUILD
FAISS 1.14	0.968	3842	0.4	1234[1]	82 s
Quiver v0.20	**0.958**	**1222**	**1.0**	**2069**	**581 s**[2]
Chroma 1.5	0.977	1009	1.1	3752[1]	153 s
Weaviate 1.27	0.983	663	1.7	2218	38 min
Milvus 2.5 (server)	0.986	649	1.9	2075	26 s
Qdrant 1.13	0.974	358	4.5	258[3]	98 s
pgvector 0.7	0.980	118	11.8	1291	132 s
LanceDB 0.33	0.557[4]	219	5.3	2475[1]	15 s

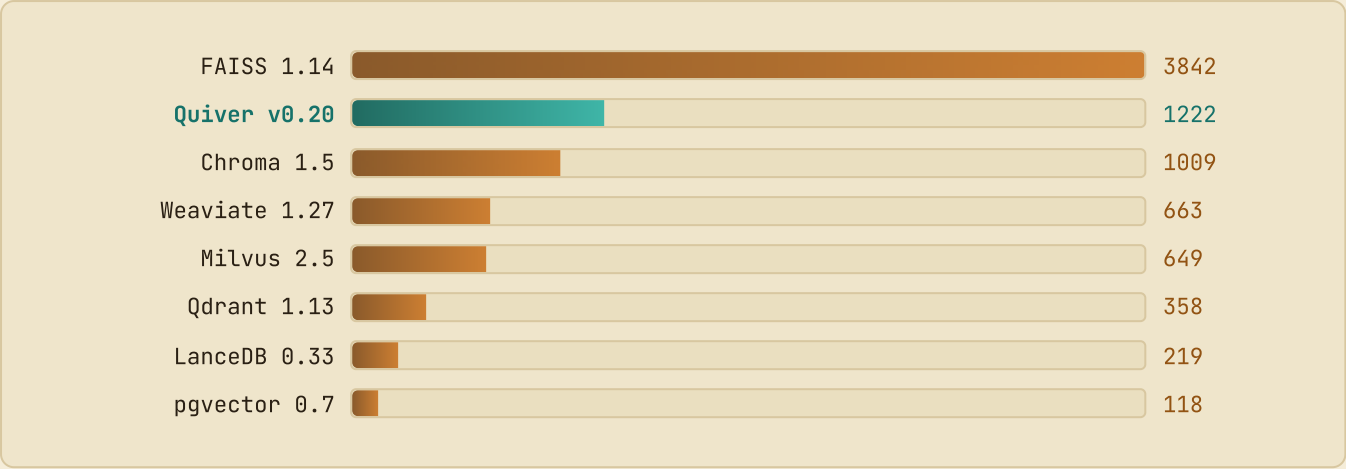

FIG. 17 — Single-thread QPS at recall@10 ≥ 0.95 (v0.20.0). Quiver is **second only to FAISS**, with the field's second-best tail latency (1.0 ms).

7.5 The harder test: GIST1M (1M × 960-d)

GIST1M is higher-dimensional and far more demanding — most systems plateau below recall 0.95 in a bounded sweep. Here Quiver **matches FAISS on recall (0.923 vs 0.919)**, and on the v0.20.0 engine is markedly faster than v0.18.0 at the same recall (182 → 268 QPS, p95 7.7 → 4.4 ms).

Table 10 — GIST1M, each system at its most efficient config (best point at ef_search ≤ 256)

SYSTEM	RECALL@10	QPS (1T)	P95 (MS)	RSS (MB)
Quiver v0.20	**0.923**	**268**	**4.4**	**10117**
FAISS 1.14	0.919	471	2.7	7526[1]
Chroma 1.5	0.790	577	2.1	8156[1]
Weaviate 1.27	0.828	418	2.8	8880
Qdrant 1.13	0.955	185	6.3	391[3]
Milvus 2.5 (server)	0.961	53	29.4	6821
pgvector 0.7	0.980	8	194	4393

The three systems that clear 0.95 here (Qdrant, Milvus, pgvector) pay for it in latency/QPS. LanceDB did not complete GIST1M — building a 960-d/1M index in-process exhausted memory (an honest DNF even with 27 GB of swap, not a fabricated row).

7.6 The v0.22.0 dimensions: recall depth, concurrency, and filtering

The v0.22.0 release added four measurement dimensions (ADR-0061), all on the same SIFT1M, all *dev-box · indicative*, and **every number below traces to a committed CSV** in `docs/benchmarks/results/comparison-v0.22.0/`.

Recall depth — how far down the list stays right

Recall@10 is the headline, but a RAG pipeline that re-ranks 50 candidates cares about recall@100. Measuring all three at once shows where the beam runs out: at a fixed `ef_search`, recall@100 needs a *deeper* candidate beam than recall@10, so it only catches up once you widen the search.

Table 11 — Quiver SIFT1M recall at depth 1 / 10 / 100 (HNSW, single thread)

EF_SEARCH	16	32	64	128	256
recall@1	0.853	0.928	0.966	0.984	0.988
recall@10	**0.793**	**0.895**	**0.958**	**0.986**	**0.995**
recall@100	0.918	0.918	0.918	0.944	0.983

Saturated concurrency — the payoff of the reader-writer lock

This is the dimension the §5.3 concurrent-reads work exists to move: **QPS under 8 saturating client threads (NT) versus one (1T).** Read it *honestly*. A single Python client process (the GIL plus one HTTP socket) is itself a concurrency ceiling, so for *light* queries (low `ef`, sub-2 ms) the client saturates first and NT sits at or below 1T. The server-side win appears on *heavier* queries, where the client is no longer the bottleneck and the parallel readers pull ahead — up to **1.76× at ef=256** (recall 0.995). It is not a fabricated "8×"; it is the real, client-limited shape of the speed-up.

Table 12 — SIFT1M throughput, 1 thread vs 8 saturating threads (NT), same recall per column

EF_SEARCH	16	32	64	128	256
recall@10	0.793	0.895	0.958	0.986	0.995
QPS (1T)	1131	1001	855	673	506
QPS (8T)	949	968	928	938	892
speed-up	**0.84×**	**0.97×**	**1.08×**	**1.39×**	**1.76×**

FIG. 18 — SINGLE-THREAD QPS FALLS WITH EFFORT; SATURATED THROUGHPUT STAYS FLAT

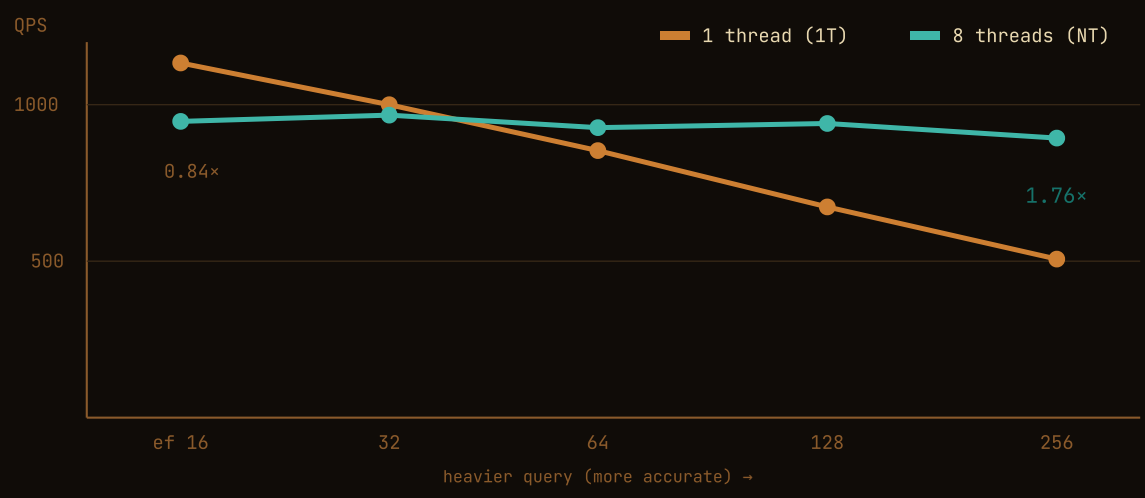

FIG. 18 — As queries get heavier, the single client thread (bronze) falls from 1131 to 506 QPS while the 8-thread server throughput (verdigris) holds near 900 — so the lines cross and saturated throughput pulls ahead, 1.76× at ef=256.

Filtered selectivity — the planner switching regimes

A payload pre-filter (`bucket < s`) keeps `s`% of the collection; recall is measured against the *filtered* exact ground truth (brute force over the matching subset). The planner crosses between two regimes, and the curve shows it: a **very selective** filter pre-filters to an *exact scan* (recall ≈ 1.0, but the scan to find the subset is the latency cost — ~142 ms); a **looser** filter post-filters an ANN result, which dips into a **recall valley** at mid-selectivity before recovering as more candidates survive the filter; and an empty filter is just pure ANN again (878 QPS).

Table 13 — SIFT1M recall & latency vs filter selectivity (ef_search = 128)

KEPT BY FILTER	RECALL@10	QPS (1T)	P50 (MS)	P95 (MS)	REGIME
1%	1.000	7	142.3	156.1	pre-filter · exact scan
5%	**0.618**	**7**	**134.6**	**147.0**	**post-filter · recall valley**
25%	0.970	5	189.2	205.4	post-filter · recovering
50%	0.981	4	253.6	273.7	post-filter
100%	0.986	878	1.1	1.4	no filter · pure ANN

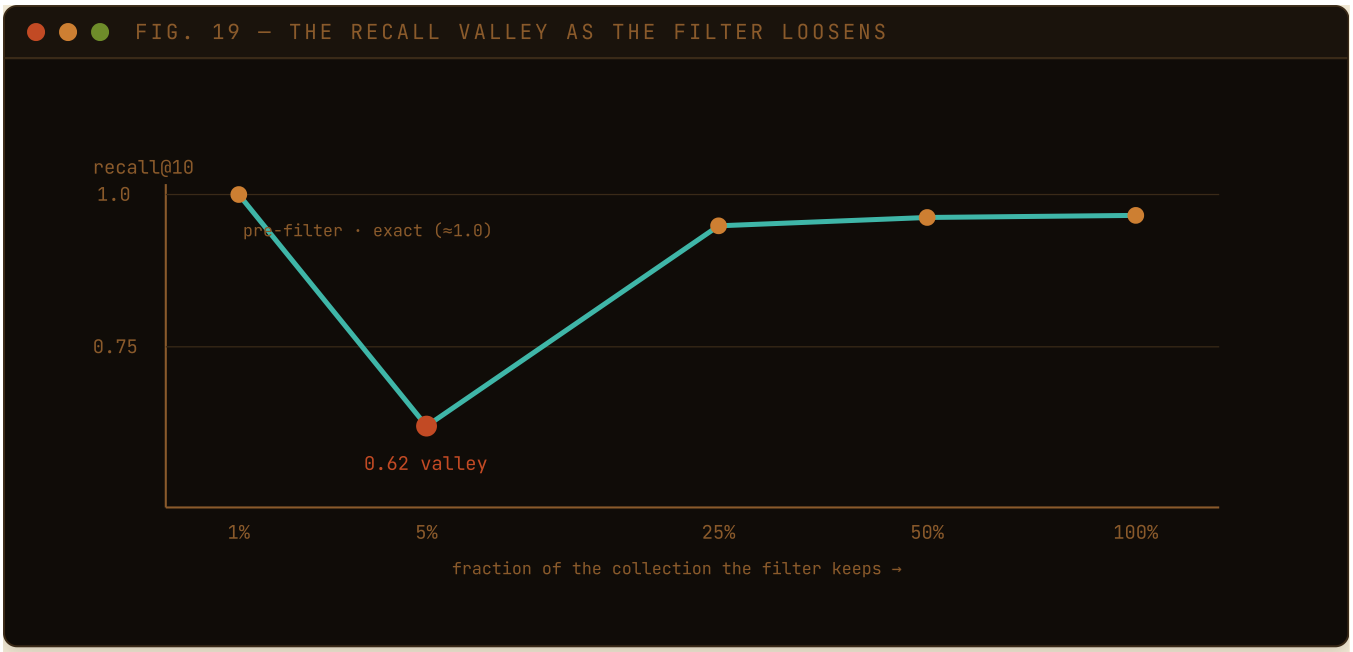

FIG. 19 — Always correct, never wrong: the filter is re-checked on every survivor (§4.5), so the "valley" is a *recall* dip the planner trades for speed, not an incorrect answer. Very selective filters pay in scan latency instead.

7.7 The real wedge: memory at a fixed recall

Critically — and Quiver says this itself — the tables above are an **in-memory HNSW** comparison, so they are *not* where Quiver's memory wedge shows up. The wedge is the **disk-resident path** (Part 4.4), which holds only PQ codes in RAM. The v0.22.0 quantization sweep shows the trade it makes, honestly: the disk-Vamana + PQ path holds recall@10 close to in-memory HNSW, but PQ trades away the *deep* tail — recall@100 falls from 0.94 to 0.71.

Table 14 — SIFT1M index/quantization trade-off, each built in its own fresh server (ef_search = 128)

CONFIG	RECALL@1	RECALL@10	RECALL@100	BUILD (S)	QPS (1T)
hnsw (in-memory)	0.984	0.986	0.944	619	883
disk_vamana + pq16	**0.974**	**0.966**	**0.709**	**1025**	**598**

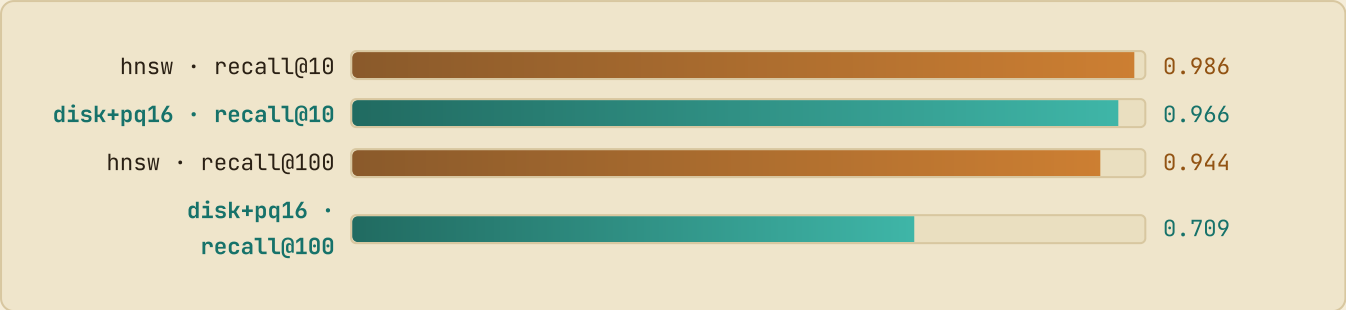

FIG. 20 — Compression keeps the head (recall@10 barely moves) but trades the tail (recall@100 collapses). The serving-RAM number is **reference-hardware-pending** on the shared dev box: post-build RSS here is the build's allocator high-water mark, while the representative figure is the **cold-reopen** footprint where only the PQ codes (plus the small delta and codebook) stay resident — the metric the durable disk index (ADR-0063) finally makes real and the wedge runner now samples after a cold restart. So it is omitted, not estimated.

~32×	~1 GB	~31 GB	1.000
LESS RAM AT FIXED RECALL	RESIDENT · 10M × 768-D	FULL-PRECISION, IN RAM	DISK-PATH RECALL@10 (SIFTSMALL)

FIG. 21 — On SIFTSMALL the disk path serves recall@10 up to 1.000 while holding only PQ codes in RAM. The head-to-head RSS vs Qdrant/LanceDB is explicitly reference-hardware-pending — never fabricated.

7.8 The verdict — the value of Quiver

WHAT THE NUMBERS PROVE

On a like-for-like in-memory test, Quiver is **second only to FAISS** in throughput-at-recall and second-best in tail latency — ahead of Chroma, Weaviate, Milvus, Qdrant, pgvector, and LanceDB — and on the punishing 960-d GIST1M it **matches FAISS on recall.** For a from-scratch engine, that is genuinely competitive territory.

But raw QPS was never the point. Quiver's value is the **combination** the benchmarks bracket:

- **Competitive speed & recall** — you are not trading performance away to get the other benefits. It holds its own against purpose-built C++ engines.
- **The memory wedge** — the differentiator the head-to-head table deliberately does *not* capture: ~32× less RAM at a fixed recall via the disk-resident PQ path, turning a "needs a big server" dataset into a "runs on a laptop" one.
- **Security by default** — encryption-at-rest on, crypto-shredding, optional encrypted vectors, audited crypto only. No competitor in this table makes that its default posture.

- **One binary, two modes, agent-native** — embeddable *and* server, with SDKs, a cockpit, and an MCP server, from a single static binary.
- **Trustworthy numbers** — every caveat is stated; absolute figures pending reference hardware are marked, not guessed. You can reproduce the comparative standings yourself.

Put plainly: **if you want competitive ANN performance with security on by default and a memory footprint small enough to self-host on modest hardware, Quiver occupies a niche the incumbents leave open.**

PART 8 PHILOSOPHY

The Decisions Behind the Code

Great engineering is visible in the choices — especially the restraint. Each is recorded as an Architecture Decision Record (ADR) in the repo.

Table 15 — the decisions that define Quiver

DECISION	WHAT THEY CHOSE	WHY
Language	Rust	Memory safety without a garbage collector → predictable low latency and no whole classes of security bugs.
Storage engine	Built from scratch (no RocksDB/SQLite/LMDB)	Full control over the on-disk format, encryption, and crash semantics — the core differentiators.
Cryptography	Only audited libraries (RustCrypto, ring/rustls)	"Don't roll your own crypto" — the one universal rule of security engineering.
Crash safety	WAL + fsync-before-ack + CRC + point-in-time recovery	The non-negotiable database promise, proven by a `kill -9` test gate.
Scope	Single-node excellence first; distribution is a labelled stretch	Do one thing superbly before doing everything adequately.
Honesty	No number ships unless reproducible on documented hardware	Trust is the product when you store someone's data.
Error handling	`unwrap()` / `expect()` banned by lint	Force every failure to be handled — no surprise panics in production.

THE THROUGHLINE

Correctness and security first, performance second, features last — and never lie about any of them.

Beyond Search: RAG, Full-Text & Operations

A search engine is only half a product. The rest is the work of using it — turning text into vectors, mixing keyword and meaning, backing data up, watching the system breathe, and keeping one noisy client from starving the others.

9.1 Bring your own *nothing*: server-side embedding

The number-one friction in retrieval is the embedding step: "I have text, but the database wants vectors, so I must run a model myself." Quiver stays model-agnostic at its core — but at the *edge* (never in the engine) it offers an opt-in, per-collection embedding hook. Configure a provider, then send **text** and let the server embed it.

> THE CORE IDEA
>
> `upsert_text` sends a document's words; the server embeds them with the collection's configured provider and stores the vector. `search_text` embeds your query the same way, searches, and can **rerank** the shortlist with a second, sharper model — retrieve-then-rerank in a single call.

The provider is chosen by configuration, never hard-wired to one vendor. Secrets are never stored: a config field names an *environment variable*, resolved at startup so a missing key fails fast.

Table 16 — built-in embedding / rerank providers (opt-in, per collection)

PROVIDER	SHAPE	NOTES
OpenAI	`/v1/embeddings`	The de-facto standard shape; also used by many local servers.
Ollama	OpenAI-compatible	Local models on your own hardware — no data leaves the box.
HTTP	OpenAI-compatible	Any endpoint that speaks the shape, parameterised by base URL + auth.
Cohere	`/v2/embed`, `/v2/rerank`	Its own shape, including a native reranker.
`fake`	deterministic	A hash-to-vector stub for tests and offline demos — no network.

FIG. 22 — THE EMBEDDING & RERANK PIPELINE (SEARCH_TEXT)

1	Client → Server	`search_text("a fast fox", rerank: true)` — just words, no vector
2	Server → Embedder	embed the query with the collection's provider (key from an env var)
3	Server → Engine	HYBRID dense ANN ⊕ BM25 full-text, fused with RRF → an over-fetched candidate pool
4	Server → Reranker	SHARPER, SLOWER ✓ a cross-encoder scores each *(query, document)* pair and reorders the pool
5	Server → Client	the reranked top-k — one round trip, no model on the client

FIG. 22 — Embedding and reranking live at the server edge, opt-in and provider-agnostic; the engine still only ever sees vectors.

WORKED EXAMPLE — THE MOVIE CORPUS, END TO END

Our running example, finally as real calls. Point a `movies` collection at an embedding provider, send *words* (no vectors — payloads like `{"year":1982}` ride along), and ask in plain language:

CINEMA — BUILD & SEARCH A MOVIE CORPUS BY TEXT

```
# the "movies" collection embeds text with its provider (key from $OPENAI_API_KEY)
upsert_text("movies", "br",   "Blade Runner — a replicant hunter in a neon dystopia")
upsert_text("movies", "gits", "Ghost in the Shell — a cyborg cop questions identity")
upsert_text("movies", "pad",  "Paddington 2 — a kind bear clears his name")

# query by meaning, then rerank the shortlist with a sharper model
search_text("movies", "a moody sci-fi about androids and identity", k=2, rerank=True)
#   → [ br (0.83), gits (0.79) ]   Paddington is correctly far away
```

⌥ UNDER THE HOOD — THE SAME CALLS DRIVE AN AI AGENT

Those two verbs, `upsert_text` and `search_text`, are not just SDK methods — they are exposed as **MCP tools** (ADR-0058), so an autonomous agent can build and query this corpus with no embedding model of its own, in the same words a human would use. The agent calls the tool; Quiver embeds, searches, reranks, and returns the films. (More on the agent surface in §9.7.)

9.2 Full-text without leaving home: BM25

Meaning search is powerful, but sometimes you want the *exact word* — a product code, a name, an acronym a model has never seen. Quiver indexes text for classic keyword ranking too, with no extra dependency.

TWO KINDS OF RELEVANCE, FUSED

Dense (vector) search finds things that *mean* the same. **BM25** — the lexical workhorse behind Lucene/Elasticsearch — scores by *which rare words two texts share, and how often*. Quiver tokenises text (Unicode split, lowercase, stop-words, a **Snowball/Porter2 stemmer** so "running", "ran", and "runs" conflate), builds an inverted index, and scores with Okapi BM25. A single `query_text` can run dense, lexical, or — best of all — both, fused with RRF.

9.3 Backups that don't stop the world

Your data directory is already portable — stop the process and copy it. But you should not have to stop. An **online snapshot** captures a consistent copy of a *running* database, anchored at one point in its history.

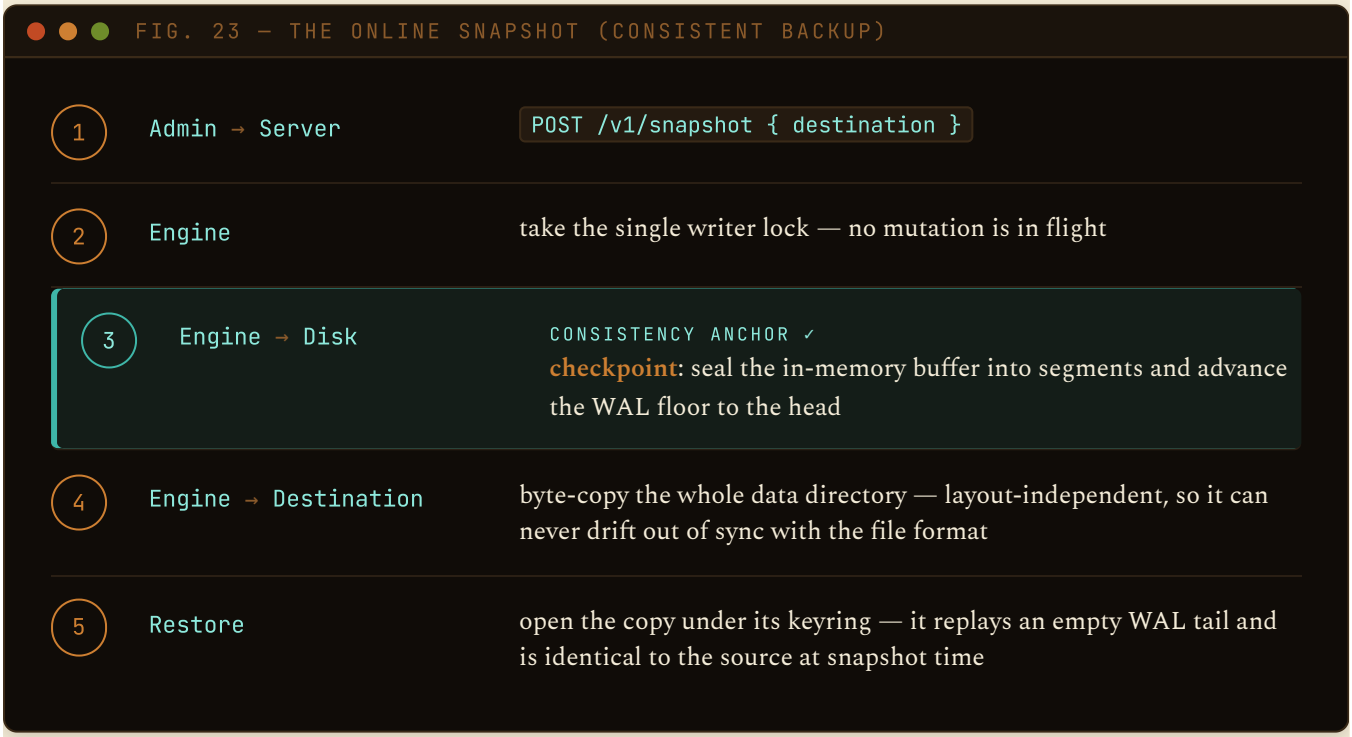

FIG. 23 — Consistency comes from the checkpoint plus the single-writer lock; the copy itself never parses a segment, so it captures new artifacts for free. The crash gate is untouched.

9.4 Scaling reads: leader & followers

Quiver is single-node first, but reads can still fan out. Point a follower at a leader (`QUIVER_LEADER_URL`) and it bootstraps from a snapshot, then tails the leader's writes **asynchronously** — no consensus, no failover, no write coordination. Followers are read-only warm standbys that absorb query load; the leader stays the single writer, so durability and the crash gate are exactly as Part 5 describes.

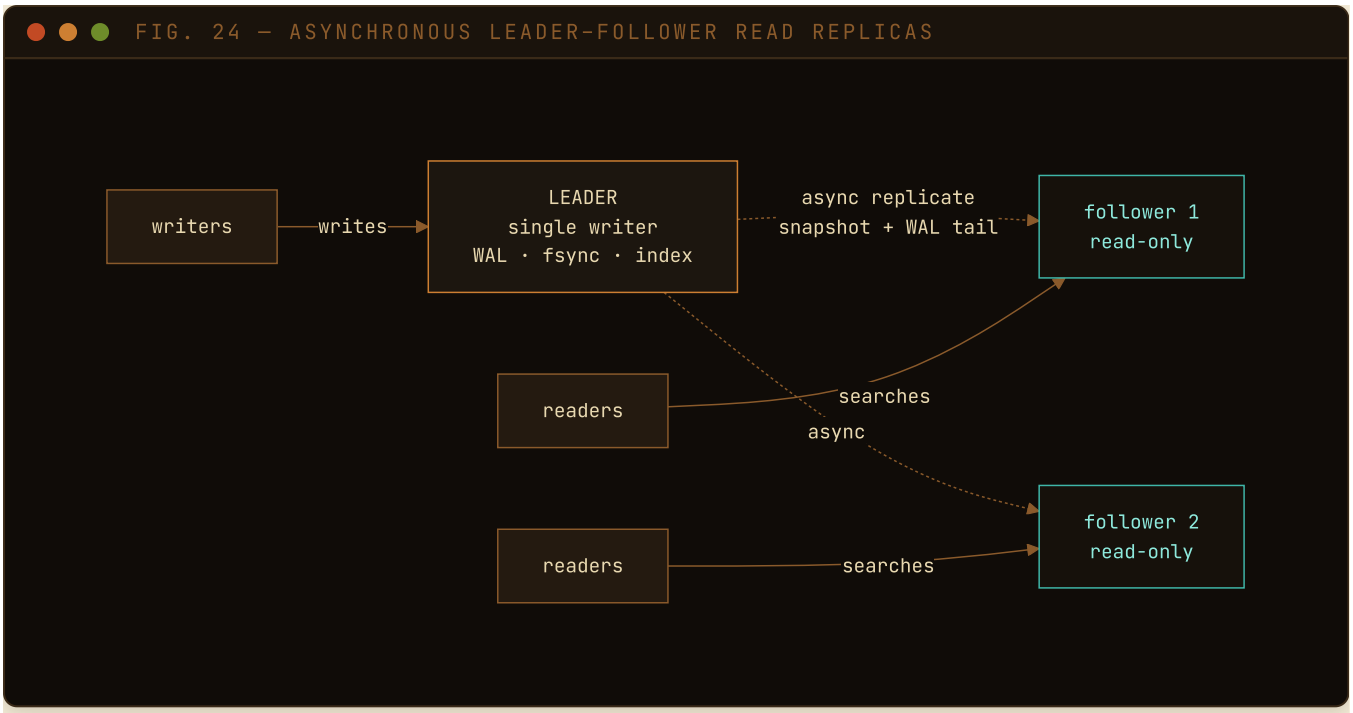

FIG. 24 — A clearly-labelled stretch feature: read scale-out and warm standbys *without* the cost of consensus. Followers are eventually consistent; the leader is the one source of truth.

9.5 Seeing inside: metrics, tracing & OpenTelemetry

An operator needs to see the system breathe. Quiver serves a Prometheus `/metrics` endpoint (open, so a scraper needs no key) with real request counters and latency histograms per route, plus security counters; engine operations carry secret-free `tracing` spans, and an importable Grafana dashboard ships in the repo. New in v0.22.0, those spans can be **exported over OpenTelemetry** (ADR-0059): opt in behind the `otlp` build feature and a runtime endpoint (`QUIVER_OTLP_ENDPOINT`), and the existing spans stream over OTLP/gRPC to Jaeger, Tempo, or Grafana — off by default (a normal build links no extra dependency, and a failed exporter degrades to console logging rather than failing startup). The metrics below are also the vocabulary the rest of this guide measures in.

Table 17 — the metrics that matter, defined (with a worked example)

METRIC	IN PLAIN WORDS	WORKED EXAMPLE
recall@k	Of the *true* k nearest neighbours, what fraction did the (approximate) search actually return?	Ask for 10; 9 of the true top-10 come back → recall@10 = **0.90**.
QPS	Queries served per second — raw throughput.	30,000 searches in 20 s → **1,500 QPS**.
p50 / p95 / p99	Tail latency: the time under which 50 / 95 / 99 % of requests finish. p99 is what your slowest users feel.	p95 = 8 ms means 95 of 100 queries answered within 8 ms.
RRF	Reciprocal Rank Fusion — merge two ranked lists using only ranks: score = Σ 1/(k₀+rank).	Rank 1 on the dense list + rank 3 on BM25 (k_0=60) → 1/61 + 1/63.
IDF	Inverse Document Frequency — rare words carry more signal than common ones; the heart of BM25.	"the" appears everywhere → ~0 weight; "xylophone" is rare → high weight.
RSS	Resident Set Size — the physical RAM a process actually holds. The memory-frugality headline.	Disk path: ~1 GB resident for 10 M vectors vs ~31 GB fully in RAM.
build time	Wall-clock time to ingest a dataset and build its index — the "time to first query."	One bulk load: one WAL fsync + one index pass, not N round trips.

9.6 Fairness under load: per-key rate limiting

THE TOKEN BUCKET

Each API key gets a bucket that refills at a steady rate up to a burst capacity; every request spends one token. Spend faster than the refill and the bucket empties — further requests get `429 Too Many Requests` with a `Retry-After` and the standard `RateLimit-*` headers. It is opt-in (off by default), enforced at one auth choke point so *every* REST and gRPC call is covered, and per-node in memory — a clustered limiter is a job for the distributed-mode design.

9.7 The client fleet: SDKs, agents, and the cockpit

One engine, many front doors. Beyond REST and gRPC, Quiver ships SDKs for **Python** (sync + async), **TypeScript**, and **Go** (standard-library only), each mirroring the same surface — collections, points, search, hybrid, full-text, server-side embedding, and snapshots. The three are kept at **method parity**: alongside the core calls, each carries the same bulk/maintenance helpers — batched upload (`upsertIter` / `UpsertBatch`), a `scroll` over a whole collection for export and re-embedding, and a paged `deleteByFilter` for GDPR erasure — with the TypeScript client taking a sync *or* async iterable and the Go client a context on every call. For AI agents, an **MCP server** exposes the database as tools — create, upsert, search, hybrid, `database_stats`, `delete_collection`, `snapshot`, and (v0.22.0) the text tools `upsert_text` / `search_text` — so an autonomous agent can not only query Quiver but *operate* it, in plain words.

The cockpit you can fly

The last front door is for humans. `quiver tui` opens a retro terminal **cockpit** in the Bronze Quiver palette — a live dashboard of connection health, a collections table with per-collection load bars, points-trend and ingest-rate sparklines, and a severity-tagged activity log. New in v0.22.0 (ADR-0060) it became *interactive*: press `/` to open a **query runner**, type a question, run a server-side embed-and-search, and inspect any result's payload in a side pane; `?` shows a keybinding overlay and `Ctrl-t` toggles a cool Slate theme. Pressing `v` on a collection opens the **constellation view** — a 2-D projection of the vector space you can pan and zoom to *see* how your data clusters. Every screen renders to a buffer behind a render-to-buffer API, so each is unit-tested with ratatui's `TestBackend` and the screenshots below are generated from the *real* render — they never go stale.

FIG. 25 — The cockpit: a live dashboard (top) and the interactive query runner (bottom) — a server-side `search_text` from the terminal, with ranked results, a payload inspector, and a recent-searches strip. Both are the *real* render, screenshotted from seeded demo data, never a mock-up.

9.8 Fitting into your stack: migration & integrations

You rarely start empty. Quiver ships **migration importers** that pull data straight from a *running* source — `quiver admin import --qdrant-url` / `--chroma-url` / `--postgres-url` stream collections from Qdrant, Chroma, and pgvector/Postgres with no manual export step (a file-based path covers the rest). Vectors, ids, and payloads come across together, so a switch is one command, not a project.

And because the SDKs mirror the common retrieval interfaces, Quiver drops into the popular RAG frameworks: it is usable as a **LangChain** and **LlamaIndex** vector store and a **Haystack** `DocumentStore`, so an existing pipeline can point at Quiver by swapping one component — keeping the security, frugality, and self-hosting it was built for.

9.9 Distribution: one binary, the `quiverdb-*` namespace

Everything above ships in **one static binary** — `quiver serve`, `quiver tui`, `quiver mcp`, and `quiver admin` are the same executable. For source and registry distribution, the crates publish under the **`quiverdb-*`** namespace (ADR-0056) — each package is `quiverdb-<crate>` while its library name stays `quiver_<crate>` and the binary stays `quiver`, so imports and `cargo install --path` are unchanged. A multi-stage Docker image, a docker-compose, and a Helm chart for Kubernetes round out the deploy story; secure defaults mean the server refuses to start on a non-loopback bind without a master key and TLS.

9.10 Scaling out: sharding & scatter-gather

One node already serves tens of millions of vectors, but a dataset eventually outgrows a single machine — or writes outpace one writer. Quiver's answer (ADR-0065, opt-in) is to **shard**: split the collection across N independent Quiver servers and put a thin **router** in front. Set `QUIVER_CLUSTER_SHARDS` to the shard URLs and the same server binary *becomes* that router — clients talk to it exactly as they would a single node.

Think of a library grown too big for one room. You split the books across several rooms, and a front desk — the router — knows which room holds what. To *file* a book (a write), the desk sends it to its one owning room. To *find the ten most relevant books* (a search), the desk asks **every** room for its ten best and merges those shortlists into the overall ten. That is a **scatter-gather**, and the merge is exact: the global top-ten can only be drawn from the rooms' individual top-tens, so collecting ten from each and keeping the best ten is provably correct.

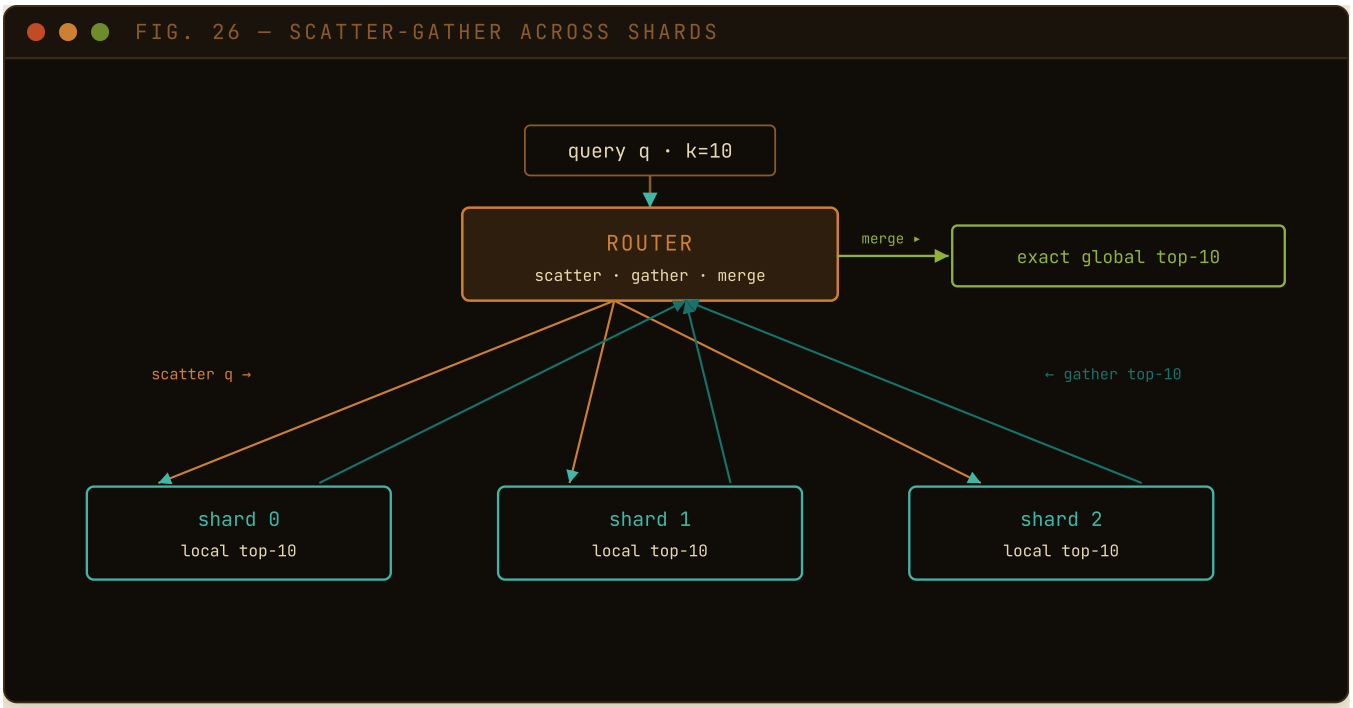

FIG. 26 — A search fans out to every shard, each returns its local top-10, and the router merges them into the *exact* global top-10. Which shard owns a point is rendezvous (HRW) hashing of its id, so adding or removing a shard moves only ~1/N of the data — the basis for dynamic, elastic scaling.

Which room owns a book is decided by **rendezvous hashing** of the book's id. The valuable property: add or remove a room and only about `1/N` of the books change hands — not a wholesale reshuffle. That is the foundation for **dynamic, elastic scaling**: grow or shrink the cluster under live load by moving only the small slice that moves. (Online rebalancing, a coordinator that tracks membership and health, per-shard write-failover via an audited Raft library, and autoscaling are the staged increments that build on this first one.) Each shard is just an ordinary single-writer Quiver server — same engine, same `kill -9` crash gate, same encryption — so the cluster is **composition over the existing engine, not a new one**, and strictly opt-in: configure no shards and the server is exactly the single node the rest of this guide describes, at zero overhead.

9.11 Read replicas: warm copies that share the reading

Sharding spreads a dataset across rooms, but inside one room a single librarian still does all the fetching — and when that room gets busy, readers queue behind every write. The fix is the one §9.4 already built for a single node, now applied **per shard**: give each room one or more **read replicas** — ordinary followers that continuously copy everything their room's primary writes (ADR-0030, reused unchanged). In the library, each room keeps a few **photocopies** of its books on a side table; a reader who only wants to *read* can take a copy while the original stays on the shelf — and can keep reading even while the librarian is busy re-filing the latest arrivals.

Declaring them is just wiring: set `QUIVER_CLUSTER_REPLICAS` to a list of `<shard_index>=<replica_url>` entries (a shard with no entry stays primary-only) and start each replica as a normal follower pointed at its shard's primary. The router now knows, for every shard, a primary **and** its replicas. **Writes, gets, and deletes still go to the one primary** — there is still a single writer per shard, so nothing can race. **Searches**, which only

read, are spread **round-robin across** `{primary} ∪ replicas`, so read traffic fans out over more machines; if a chosen copy is unreachable the router quietly tries the shard's others, so a down replica costs throughput, not answers.

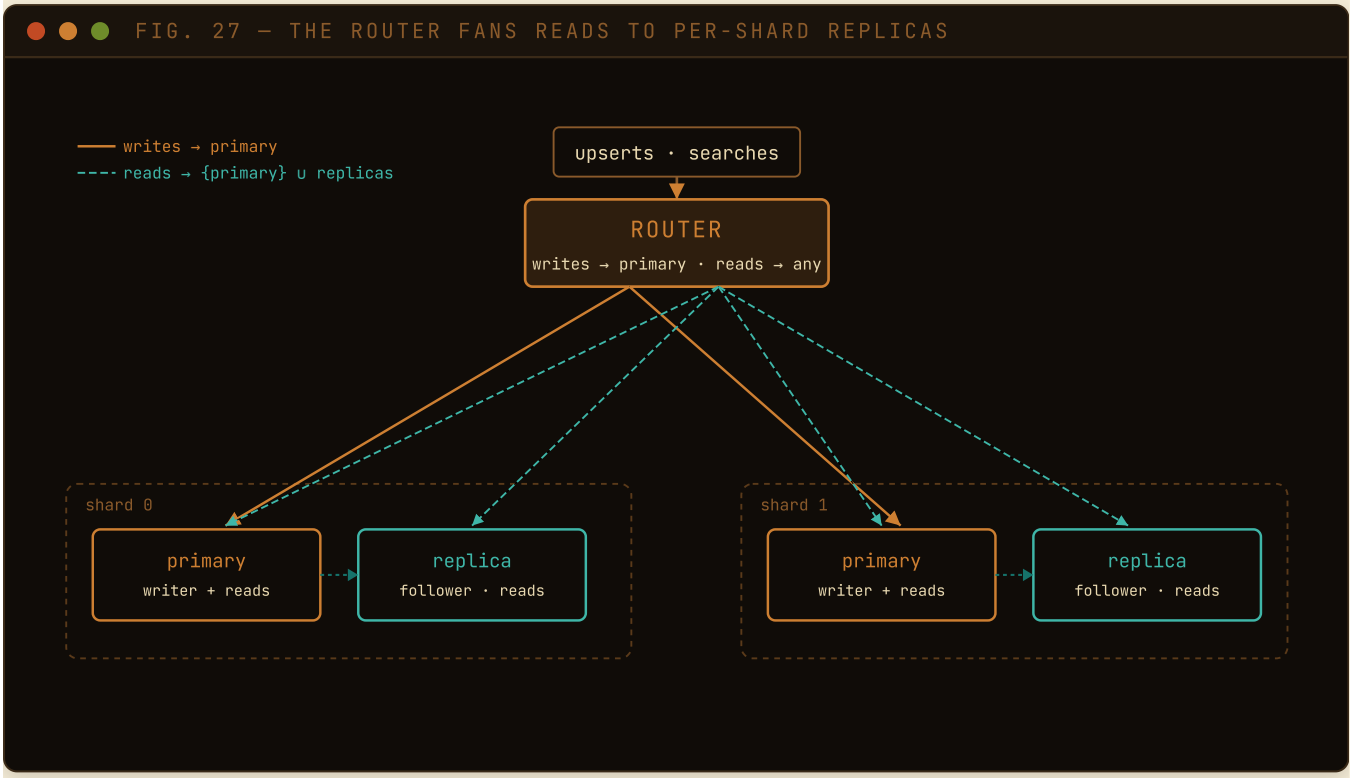

FIG. 27 — Each shard is a single-writer *primary* plus one or more follower *replicas*. Writes go to the primary alone (single writer per shard); searches round-robin across `{primary} ∪ replicas` to spread read load, falling back to another copy if one is down. Replicas are eventually consistent — they lag their primary by the replication delay — and a follower refuses direct writes, so a mis-route cannot corrupt one. This is read scale-out and warm standbys, not write failover.

The one honest caveat is **eventual consistency**: a replica lags its primary by however long replication takes (usually milliseconds), so a search served from a replica may miss a write committed an instant ago — the same trade as §9.4's single-node read replicas, and the reason replicas serve *searches* while point-lookups (`get`) still go to the primary for read-your-writes. A follower also **refuses direct writes**, so even a misrouted write can never corrupt a replica. This is read scale-out and warm standbys — *not* write failover; promoting a replica when its primary dies needs consensus, the audited-Raft increment in the roads not yet taken below.

9.12 Growing without downtime: dynamic, elastic scaling

The headline of the cluster: you can **add a shard to a live cluster** — under read and write load — and it rebalances itself with **no downtime and no lost writes**, moving only the small slice that has to move. A new desk supervisor, the **coordinator**, keeps the master floor-plan stamped with a **version number**; the routers re-read it every couple of seconds, so they learn about a new room *without ever closing*. Because of rendezvous hashing, opening that room moves only about `1/N` of the books — every other book stays put.

Moving that slice is a careful handshake, because readers and writers never stop. (1) the new room joins the floor-plan but the **old room still owns and serves** the slice, and every *new* arrival for those books is filed in **both** rooms (a **dual-write**), so nothing written mid-move is missed; (2) the coordinator **copies** the slice's

existing books across — never overwriting one the new room already holds from a dual-write, so a book updated mid-move keeps its newest version; (3) once the new room has caught up the coordinator **flips** the floor-plan to the next version — the new room now owns the slice and the old room **discards its copies**. A search always reaches a room that has the books, and a router still on the previous version is still correct (the old room serves until everyone has refreshed).

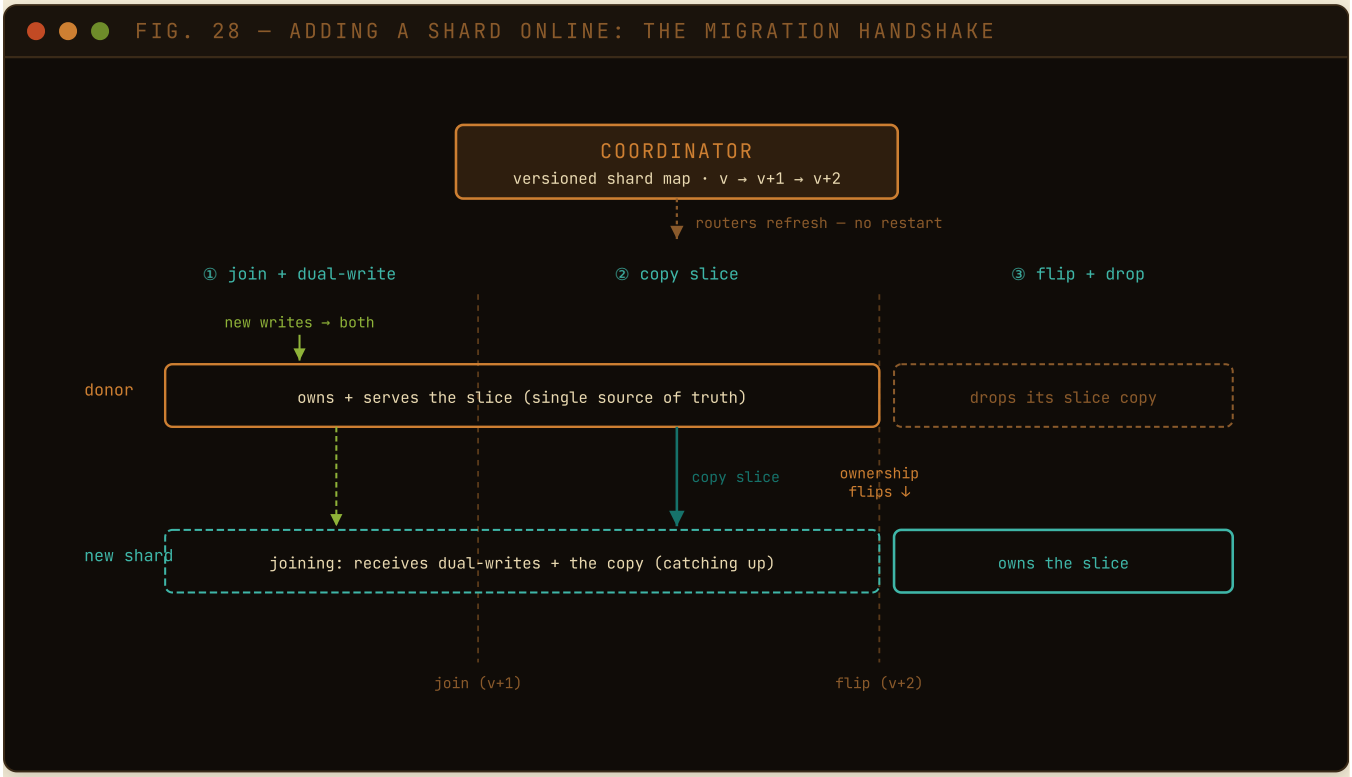

FIG. 28 — Adding a shard to a live cluster. The coordinator versions the shard map and routers refresh it with no restart. The new shard's ~1/N slice migrates online: the donor keeps serving it while every new write is dual-written to both, the coordinator copies the existing slice across, then ownership *flips* at the next version and the donor drops its copy. No downtime, no lost writes — only the moved slice changes hands.

In Quiver terms: run one server as the **coordinator** (`QUIVER_COORDINATOR=true`), point routers at it (`QUIVER_COORDINATOR_URL`), then `POST /cluster/shards/grow` with the new shard's URL — the coordinator does join, copy, flip, and drop in the background while the cluster keeps answering. The same machinery shrinks a cluster, and an autoscaler could one day call it from load signals. The one thing still missing — automatic **write** failover when a primary dies — is the next section.

9.13 Surviving a node failure: per-shard write HA

Sharding, replicas, and online growth all assume each room's librarian stays at the desk. But a librarian can call in sick — a shard's primary can crash — and until now that shard's slice went read-only until an operator stepped in. **Per-shard write high availability** (ADR-0067, opt-in) closes that last gap: each room keeps its books on a **quorum of shelves**, and if the head librarian for a room steps away, the others **elect** a new one on the spot — and not a single checked-out book is lost.

Concretely, each shard runs **one Raft group** — an *audited* consensus library, not a hand-rolled log (the same discipline that put `arc-swap` under MVCC). The shard's replicas (§9.11) become the group's **voters**; its primary is the **leader**. A write is **proposed** to the leader and **acknowledged only after a quorum** of voters has committed it — so the instant a client hears "done", a majority already holds that write, and every future leader must too. If the leader's process dies, the surviving voters **elect** a new leader automatically: no operator, and no lost acknowledged write.

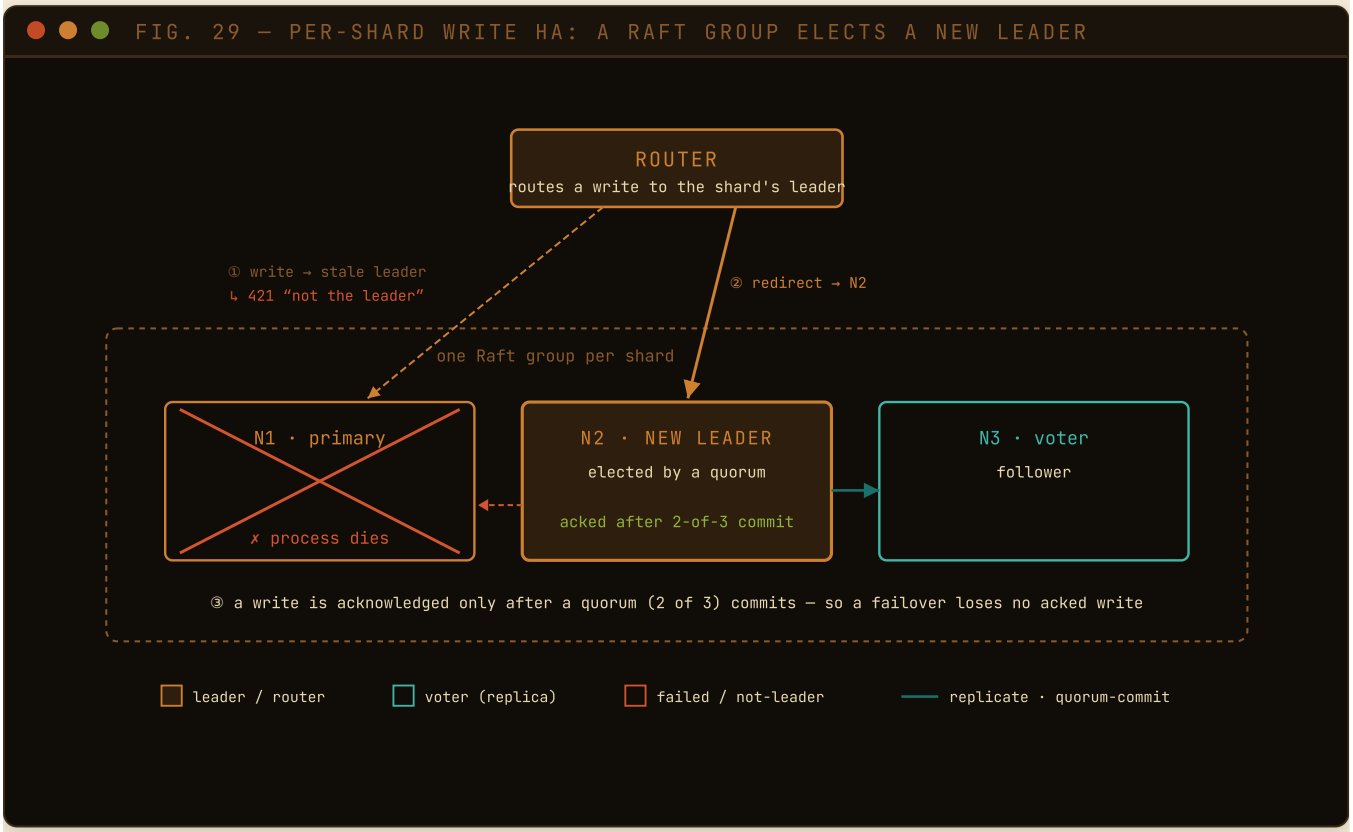

FIG. 29 — Per-shard write high availability. Each shard runs one Raft group whose voters are its replicas. A write is acknowledged only after a quorum (here 2 of 3) commits it, so an acknowledged write is never lost. When the leader's process dies the survivors elect a new leader; a router that wrote to the stale leader gets a *"not the leader"* (HTTP 421) reply and redirects to the new one, with no coordinator on the write path. A minority cannot reach a quorum, so it refuses writes — no split-brain. Opt-in per shard.

The router learns the new leader the self-correcting way it learns everything else (§9.12): it routes a shard's writes to the leader, and a write that reaches a former leader gets a **"not the leader"** reply — so the router redirects and carries on, with no coordinator on the write path. The safety side of the quorum rule is just as important: a **minority** that cannot reach a quorum **refuses** writes rather than inventing its own history, so a network split can never elect two leaders writing divergent data — **no split-brain**. The honest cost is a round-trip — every write now waits for a quorum, not just the local disk — which is why write HA is **opt-in per shard**: a shard that doesn't need failover keeps the cheaper single-writer path, and single-node Quiver is wholly unaffected. The on-disk format and the per-shard `kill -9` crash gate are unchanged; Raft is a replicated log *above* the engine, not a new one.

The group is **durable and elastic**, not a fixed three. The Raft log is itself **crash-safe** — a voter that dies recovers its own log and vote on restart and rejoins, the same `kill -9` discipline the engine already keeps — so a failure is never one-way. It **compacts** that log against periodic snapshots, so the log can't grow without

bound and a far-behind or freshly added voter catches up by **installing a snapshot** instead of replaying history. And its **voters can change online**: an operator (or, later, an autoscaler) adds or removes a voter on a running shard with no restart — the new node joins as a non-voting learner, catches up, and is then promoted. So a shard's redundancy can grow with its importance, live.

9.14 Growing on its own: autoscaling

§9.12 lets you *add a room by hand* under load; the obvious next step is to let the desk supervisor open one **by itself**. Turn the policy on and the **coordinator** watches how full each room is getting and, when the busiest crosses a line you set, opens a new room from a **pool you keep on standby** — running the exact same safe migration §9.12 already does. No pager, no operator, no lost book.

It is deliberately a **policy, not magic**. Nothing happens without two things you provide: a **threshold** (the per-shard point count that means "too full") and a **standby pool** (the spare shard URLs to grow into). And it is bounded: a **cooldown** keeps it from thrashing while a migration settles, an in-flight migration is **never interrupted**, and a **max-shards cap** stops runaway growth. The load signal is honest and legible — the number of vectors a shard holds — not an opaque score.

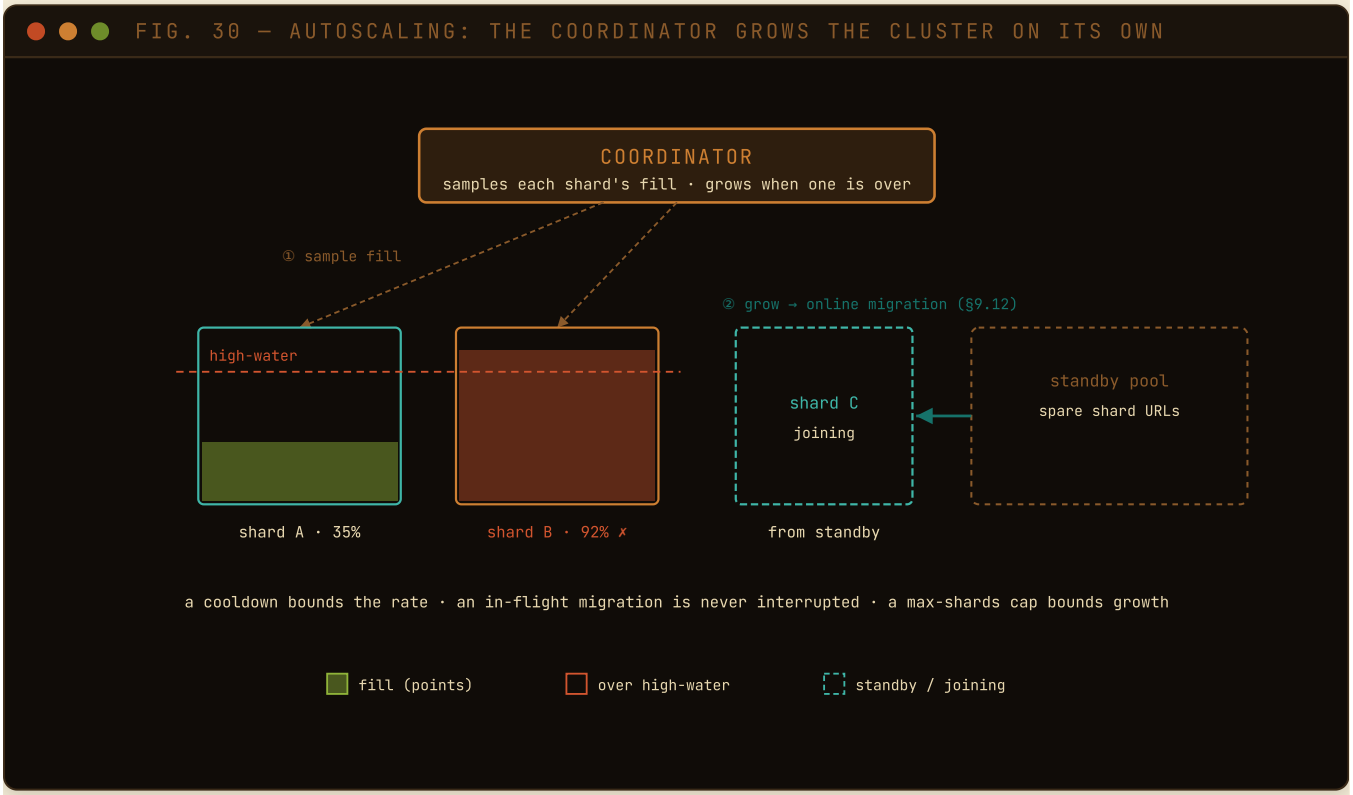

FIG. 30 — Autoscaling (scale-out). The coordinator samples each shard's point count; when the busiest crosses the configured high-water mark it pulls a shard from the standby pool and runs the same safe online migration as a manual grow — no operator, no lost write. A cooldown, a no-interrupt rule, and a max-shards cap bound it; it is opt-in, and scale-in stays a manual drained remove.

In Quiver terms: set an `[autoscale]` table on the coordinator (`enabled`, `high_water_points`, a `standby_urls` list, `interval_secs` / `cooldown_secs` / `max_shards`) and it grows the cluster from the standby pool on its own. **Scaling in is deliberately still manual** — safely *draining* a shard's slice back onto its

neighbours online is a reverse migration that is its own careful increment, and Quiver would rather make you run a drained `DELETE` than risk a silent dropped write to save a machine automatically. Off by default; a cluster without the policy is exactly the one §9.12 describes.

9.15 Optional muscle: GPU-accelerated distance

Everything Quiver does fast on a CPU, it does with one trick repeated billions of times: measure the distance between two vectors (§1.2). On a brute-force or exact scan that one kernel — *this query against those ten thousand vectors* — is essentially the whole bill. A CPU does it a few lanes at a time (§4.2's SIMD); a **GPU** does thousands of rows at once. So for that batch, a GPU is the right tool — when you have one.

The catch is that most self-hosted boxes **don't** have one, and a hard CUDA dependency would wreck the "single static binary that runs anywhere" promise. So the GPU lives entirely behind a seam (the distance kernel, §4.2) and an **off-by-default `cuda` build feature**, exactly like the `raft` and `otlp` features: a normal build links **zero** CUDA, the CPU SIMD kernel stays the default *and* the fallback, and the GPU produces **identical** numbers — a pure accelerator, never a correctness dependency, never touching the on-disk format or the crash gate. The CUDA driver is even loaded *dynamically* and the kernel compiled at runtime, so the feature **compiles without a CUDA toolchain** — only *running* the GPU path needs a device, and with no device the code quietly uses the CPU.

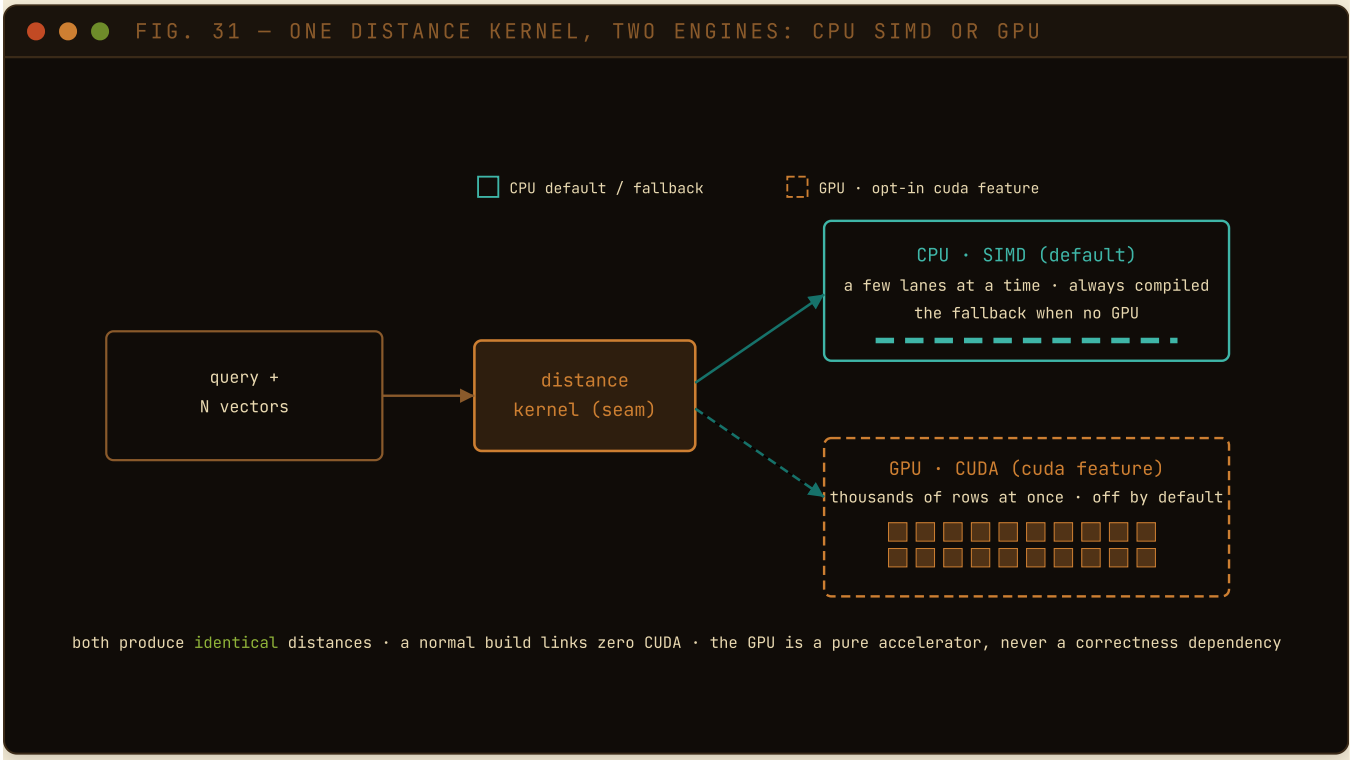

FIG. 31 — The distance kernel is a seam with two engines. The CPU SIMD path is always compiled, the default, and the fallback; the GPU path (the off-by-default `cuda` feature) computes thousands of rows at once when a device is present. Both produce identical distances, a normal build links zero CUDA, and the GPU is a pure accelerator — never a correctness dependency, never touching the on-disk format or crash gate.

This is the **batch-distance kernel** (the part of GPU acceleration that earns its keep), validated against the CPU kernel on real hardware. Folding it into the planner's exact-scan path and the k-means / quantizer training batches is the next step; the point today is the seam — Quiver can borrow a GPU's muscle where one exists, and lose nothing where one doesn't.

9.16 The roads not (yet) taken

What is *not* yet built is deliberately so, until the single-node story is unbeatable and the need is concrete. The big features now stand: write HA (§9.13), autoscaling (§9.14), and the GPU distance kernel (§9.15). What remains is mostly **wiring and evidence** — threading the GPU kernel through more of the search and build paths, automating safe scale-*in* (a reverse-migration drain), and the launch polish a `v1.0.0` is gated on: the published **reference-hardware** benchmark table (recorded on dedicated machines, never fabricated), a hosted docs site, and a recorded cockpit cast. None compromises the single static binary, and single-node remains the default everywhere.

A REFERENCE

Glossary for the Newcomer

Every term this guide leans on, defined once, in plain language.

`Vector / Embedding`
A list of numbers representing the *meaning* of content. Similar meanings → nearby numbers.

`Embedding model`
The neural network that turns text/images into vectors. (You bring your own.)

`Metric`
How "distance" is measured: cosine (angle), L2 (ruler), dot product (angle + length).

`ANN`
Approximate Nearest Neighbour — finding the *almost*-perfect closest vectors fast.

`Recall`
Accuracy of approximate search: of the true top-K, how many were found. (1.0 = perfect.)

`k / top-k`
How many results you want ("the 10 most similar movies").

`Index`
The data structure that makes search fast (HNSW, Vamana/DiskANN, IVF).

`HNSW`
A layered "navigable graph" index; fast, lives in RAM.

`Vamana / DiskANN`
A graph index engineered to live on SSD with a tiny RAM footprint.

`IVF`
An index that clusters vectors into cells and searches only the relevant ones.

`Quantization`
Lossy compression of vectors into tiny codes (Scalar ~4×, Product/Binary ~32×).

`Re-ranking`
After a fast fuzzy pass, recomputing exact distances on the shortlist.

`WAL`
Write-Ahead Log — the "write it down before doing it" log that survives crashes.

`fsync`
The command that forces data all the way onto physical disk.

`AEAD`
Encryption (e.g. XChaCha20-Poly1305) that also detects tampering.

`Crypto-shredding`
Deleting data permanently by destroying its (tiny) key.

`RBAC`
Role-Based Access Control: who may do what, to which data.

`RRF`
Reciprocal Rank Fusion — merging two ranked lists using only their ranks.

`Payload / metadata`
Structured data attached to a vector (`{"year":1982}`), used for filtering.

`MCP`
Model Context Protocol — lets AI agents call tools; Quiver exposes itself as such.

`BM25`
The classic keyword-ranking formula (Lucene/Elasticsearch): scores by which *rare* shared words two texts have, and how often.

`IDF`
Inverse Document Frequency — how rare (and therefore informative) a word is across the corpus.

Stemmer (Snowball/Porter2)

Reduces words to a root so "running", "runs", and "ran" match as one.

Rerank

A second, sharper pass (a cross-encoder) that reorders a retrieved shortlist for precision.

Embedding provider

An external model (OpenAI/Cohere/Ollama/HTTP) Quiver can call, at the edge, to turn your text into vectors.

Snapshot / checkpoint

An online, consistent backup of the whole database; the checkpoint seals memory to disk so the copy needs no replay.

Token bucket

A rate limiter: a bucket refills at a fixed rate; each request spends a token; empty → `429`.

Prometheus / OpenTelemetry

Standard formats for metrics (scraped from `/metrics`) and traces (spans) — what dashboards and alerting consume.

recall@k

The fraction of the *true* k nearest neighbours an approximate search actually returns — search quality.

p95 / p99

Tail latency — the time under which 95 % / 99 % of requests complete; what slow users feel.

RSS

Resident Set Size — the physical RAM a process actually uses; Quiver's memory-frugality headline.

RwLock (reader-writer lock)

A lock that lets many readers in at once but a writer alone — how Quiver runs searches in parallel.

Snapshot isolation / MVCC

A read sees one consistent version of the data; it may miss a just-committed write but never a half-applied one.

Off-lock rebuild

Rebuilding a stale index without holding the exclusive lock — the prior index is served while the new one is built, then swapped in.

Lock-free MVCC reads

Serving reads from an `arc-swap` snapshot (base index + a small overlay of recent writes) with no lock, so reads never block on the writer (§5.5, ADR-0064; opt-in via `QUIVER_MVCC_READS`).

Overlay

The small, immutable set of writes (recent vectors + tombstones) layered on a published base index so an MVCC read stays fresh without rebuilding the index.

Sharding

Splitting a collection across N independent Quiver servers (shards) so a dataset can exceed one machine. A router decides which shard owns each point (§9.10, ADR-0065).

Scatter-gather

Answering a search in a cluster by asking every shard for its local top-k, then merging into the exact global top-k.

Rendezvous (HRW) hashing

Picking a point's owning shard by the highest hash of (shard, id), so adding or removing a shard moves only ~1/N of the data — the basis for elastic scaling.

Saturated throughput (NT)

QPS measured under many concurrent client threads, versus one (1T) — the test for whether parallel reads actually help.

Selectivity

The fraction of a collection a filter keeps; it decides whether the planner pre-filters (selective) or post-filters (broad).

recall@1 / recall@100

Recall measured at different depths — recall@100 needs a deeper search beam than recall@10.

Replication (leader-follower)

Asynchronous read replicas: a follower tails a leader's writes to scale reads, with no consensus.

Sharding / scatter-gather

Splitting data across nodes (designed, not built); a query fans out to all shards and merges their results.

Cockpit (TUI)

The retro terminal interface (`quiver tui`): a live dashboard plus an interactive query runner.

OTLP

OpenTelemetry's wire protocol for exporting traces to a collector (Jaeger/Tempo/Grafana); opt-in in Quiver.

ColBERT / MaxSim

Late-interaction retrieval: a document is many token vectors, ranked by summing each query token's best match.

PQ / ADC

Product Quantization compresses vectors into codes; Asymmetric Distance Computation scores them with a per-query lookup table.

B SOURCES

References & Repository Documentation

Every claim in this guide traces to source code or to a documented, reproducible benchmark. The primary sources live in the Quiver repository.

Architecture & overview

R1 **Architecture overview & crate map**
`docs/architecture/overview.md`

R2 **C4 container view** · C4 system context
`docs/architecture/c4-container.md` · `c4-context.md`

R3 **Project README** — positioning, quickstart, benchmark tables
`README.md`

Indexes, kernels & quantization

R4 **HNSW** (Malkov & Yashunin, 2020; arXiv:1603.09320)
`crates/quiver-index/src/hnsw.rs` · ADR-0007

R5 **Vamana / DiskANN** (Subramanya et al., NeurIPS 2019)
`crates/quiver-index/src/vamana.rs` · `disk.rs` · ADR-0019

R6 **IVF** coarse Voronoi partitioning · SpFresh/LIRE updates
`crates/quiver-index/src/ivf.rs` · ADR-0007 · ADR-0023

R7 **Quantization** — scalar / product / binary
`crates/quiver-index/src/quant/` · ADR-0008

R8 **SIMD distance kernels** — runtime dispatch + scalar fallback
`crates/quiver-simd/src/` · ADR-0009

R9 **Hybrid (dense + sparse) search · RRF fusion**
`crates/quiver-query/src/sparse.rs` · ADR-0043

R10 **Multi-vector / late-interaction (ColBERT, MaxSim)**
`crates/quiver-index/src/colbert.rs` · ADR-0028 · ADR-0034

Storage, durability & security

R11 **Write-ahead log & page format** — CRC, recovery
`crates/quiver-core/src/wal.rs` · `page.rs` · ADR-0004 · ADR-0005

R12 **Envelope encryption & crypto-shredding**
`crates/quiver-crypto/src/envelope.rs` · `docs/security/crypto.md` · ADR-0010

R13 **Encrypted vectors** — client-side & DCPE
`docs/security/dcpe.md` · `client-side-vectors.md` · ADR-0031 · ADR-0032

R14 **Auth, RBAC, tenancy & threat model**
`docs/security/threat-model.md` · ADR-0011 · ADR-0013

Benchmarks & method

R15 **Head-to-head comparison (v0.20.0)** — SIFT1M + GIST1M, 7 competitors
`docs/benchmarks/results/comparison-v0.20.0/`

R16 **v0.22.0 dimensions** — recall@{1,10,100}, saturated concurrency, quantization & filter sweeps
`docs/benchmarks/results/comparison-v0.22.0/` · ADR-0061

R17 **Benchmark methodology & harness** · disk-path memory results · reference-hardware runbook
`docs/benchmarks/methodology.md` · `bench/`

Concurrency, operability & v0.22.0 features

R18 **Concurrent reads** (RwLock) · **off-lock index rebuild**
`crates/quiver-server/src/lib.rs` · ADR-0057 · ADR-0062

R19 **Interactive TUI cockpit** · **OTLP traces exporter** · **MCP text tools**
`crates/quiver-tui` · `quiver-server` · `quiver-mcp` · ADR-0060 · 0059 · 0058

R20 **Replication** · **packaging & the `quiverdb-*` namespace**
`docs/security/threat-model.md` · `CHANGELOG.md` · ADR-0056

Decisions & project

R21 **Architecture Decision Records** (ADR-0001 … ADR-0062)
`docs/adr/`

R22 **Documentation site** (mdBook) — concepts → API → security → architecture
`apps/docs/` · build with `just docs`

R23 **Security policy** · roadmap · contributing
`SECURITY.md` · `docs/roadmap.md` · `CONTRIBUTING.md`

Repository — **github.com/achref-soua/quiver** · License: AGPL-3.0-only · Edition v0.29.0 · All paths are relative to the repository root.

“A quiver holds arrows, and an arrow is a vector.
In mathematics a quiver is a directed graph —
which is exactly what an HNSW or Vamana index is.”

quiver@cockpit:~$ ./read --status=complete ✓

The security-first vector database
Client-side-encryptable · memory-frugal · runs on a laptop

github.com/achref-soua/quiver
Open-source · AGPL-3.0-only