SQLite for AI Agent Memory: 2025 Best Practices and Architectural Report

1. Executive Summary and Strategic Context

The landscape of Artificial Intelligence development in 2025 has undergone a profound architectural pivot. The centralization of intelligence—characterized by massive, cloud-hosted Large Language Models (LLMs) managing ephemeral context windows—is rapidly yielding to a decentralized, "Edge AI" paradigm. In this new era, the Holy Grail of AI utility is **Persistent Memory**: the ability for a localized agent to retain, recall, and synthesize user interactions over days, months, and years, without compromising privacy or incurring prohibitive cloud storage costs.

This report provides an exhaustive technical analysis of designing a persistent memory system for an AI assistant operating under strict embedded constraints: a 6GB RAM environment with a target capacity of 100,000 memory records. The central thesis of this research validates a pure **SQLite** architecture—augmented by the sqlite-vec vector search extension and the FTS5 full-text search engine—as the optimal solution for this specific scale. While the allure of specialized vector databases like LanceDB or Milvus remains strong for enterprise-scale retrieval (millions to billions of vectors), the overhead of managing these distributed systems is unjustified for the 100,000-record threshold, which sits comfortably within the capabilities of a modern, single-file embedded database.

We will explore the convergence of relational data, vector embeddings, and full-text search indices into a unified .sqlite artifact. This "Local-First" architecture minimizes serialization overhead, simplifies transaction management through ACID compliance, and leverages the mature, battle-hardened ecosystem of SQLite. However, implementing this stack effectively requires navigating a minefield of engineering challenges: mitigating brute-force search latency, selecting the correct tokenizer for conversational ambiguity, designing schemas that balance episodic and semantic recall, and optimizing recursive queries for knowledge graph traversals.

The following sections serve as a comprehensive blueprint for the modern AI engineer. We will dissect the internal mechanics of sqlite-vec, evaluate the trade-offs of the Trigram versus Porter tokenizers, mathematically deconstruct Reciprocal Rank Fusion (RRF) for hybrid search, and analyze the state-of-the-art embedding models of late 2024 and 2025 that make this localized intelligence possible.

---

2. The Architecture of sqlite-vec: Vector Search in

2025

The foundation of any semantic memory system is the ability to perform similarity searches on high-dimensional vectors. Historically, SQLite users relied on sqlite-vss, a wrapper around the Facebook AI Similarity Search (Faiss) library. However, the release of sqlite-vec has marked a significant departure, offering a vector search engine written entirely in C with no external dependencies. This shift is not merely a change in libraries; it represents a fundamental maturity in the embedded vector search ecosystem.

2.1 Stability and Ecosystem Status

As of January 2025, sqlite-vec has established itself as the successor to sqlite-vss. The library, while still carrying a "pre-v1" label (specifically version 0.2.3-alpha released in late December 2024), has achieved a level of stability suitable for production deployment in controlled environments.[1] The transition from sqlite-vss was driven by the need for broader portability—specifically, sqlite-vss struggled with Windows compatibility and WebAssembly (WASM) compilation due to its heavy Faiss dependency. In contrast, sqlite-vec runs ubiquitously, from high-performance Linux servers to Raspberry Pis, and crucially, inside web browsers via WASM.[3]

A critical milestone in the library's maturity was the addition of Android 16KB page support in the v0.2.3-alpha release. This update was necessitated by Google Play Store requirements for Android 15+, signaling the library's readiness for mobile-first AI agents—a key deployment target for the "embedded" context of this report.[1]

2.2 The "Brute Force" Design Philosophy

One of the most contentious yet deliberate design choices in sqlite-vec for the 2025 landscape is its reliance on optimized brute-force search (Exact Nearest Neighbor) rather than persistent Approximate Nearest Neighbor (ANN) indexing algorithms like HNSW or IVF.[5]

For an architect accustomed to server-side vector databases where HNSW (Hierarchical Navigable Small World) graphs are standard, this might initially appear as a regression. However, for a dataset of 100,000 records, the brute-force approach is often mathematically superior.

- **Recall vs. Speed:** HNSW indexes trade recall (accuracy) for speed. They might return the *approximate* top 5 matches. In an AI memory context, missing the *single most relevant* memory due to graph traversal errors can lead to hallucinations. Brute force guarantees 100% recall.
- **Memory Overhead:** HNSW graphs are memory-hungry. An HNSW index can easily consume 1.5x to 2x the memory of the raw vectors themselves to store the graph edges and navigational structures. In a 6GB RAM constrained environment, saving this memory allows for larger caching of the actual data or larger context windows for the LLM.

- **Latency Profile:** The critical question is: "Is brute force fast enough?" Benchmarks indicate that scanning 100,000 vectors of 384 dimensions (typical for the all-MiniLM-L6-v2 model) takes approximately 15 to 50 milliseconds on modern embedded CPUs (e.g., Apple M-series, high-end Snapdragons, or modern Intel i5s).[6] In the context of a Chat RAG (Retrieval Augmented Generation) pipeline, where the LLM takes seconds to generate a response, a 50ms retrieval latency is negligible. The bottleneck is the generation, not the search.

2.3 Internal Mechanics: Virtual Tables and Shadow Storage

sqlite-vec utilizes SQLite's Virtual Table mechanism to manage vector data. When a user creates a vector table:

SQL

```sql
CREATE VIRTUAL TABLE vec_memories USING vec0(
    embedding float
);
```

The extension creates "shadow tables" within the database file to store the raw binary blobs of the vectors. This integration ensures that vector data is managed by SQLite's robust paging system. When a query is executed, the extension loads these pages—often utilizing the operating system's file system cache—and performs the distance calculations using SIMD (Single Instruction, Multiple Data) instructions.[8]

The library automatically detects the CPU architecture and dispatches the optimized calculation kernel:

- **AVX/AVX2/AVX-512** for x86_64 processors (Intel/AMD).
- **NEON** for ARM64 processors (Apple Silicon, Raspberry Pi, Android).

This hardware acceleration is what allows the "brute force" scan to remain performant at the 100,000-record scale. Without SIMD, the pure C loop would likely be 4-8x slower, pushing latency into the noticeable 200ms+ range.

---

3. Vector Quantization and Storage Optimization

Given the constraint of 6GB RAM and the target of 100,000 records, efficient storage and retrieval are paramount. While 100,000 vectors of 384 dimensions fit comfortably in RAM (approx. 150MB), future-proofing the system or expanding the embedding dimension necessitates a look at quantization.

3.1 Float32 vs. Int8 vs. Binary

Standard embeddings are 32-bit floating-point numbers (float32).

- **Float32 Size:** $100,000 \times 384 \times 4 \text{ bytes} \approx 153.6 \text{ MB}$.
- **Int8 Quantization:** sqlite-vec supports storing vectors as 8-bit integers. This reduces the size by a factor of 4.
  - **Size:** $100,000 \times 384 \times 1 \text{ byte} \approx 38.4 \text{ MB}$.
  - **Accuracy:** Research indicates that converting standard embeddings to Int8 results in a negligible accuracy drop (often < 2% on retrieval benchmarks) while quadrupling the memory bandwidth efficiency.[9]
- **Binary (Bit) Quantization:** This compresses the vector to single bits.
  - **Size:** $100,000 \times 384 / 8 \text{ bytes} \approx 4.8 \text{ MB}$.
  - **Performance:** Distance calculation becomes a Hamming distance operation (XOR followed by a population count), which is incredibly fast.[3]
  - **Risk:** Standard models like all-MiniLM-L6-v2 are *not* trained for binary quantization. Binarizing them directly often leads to a catastrophic collapse in retrieval quality. Binary quantization requires specialized models (like nomic-embed-text-v1.5 with binary support or mixedbread-ai) or re-ranking layers to be effective.[8]

3.2 Matryoshka Embeddings

A significant trend in late 2024 and 2025 is the adoption of Matryoshka Representation Learning (MRL). MRL models are trained such that the most critical semantic information is concentrated in the earlier dimensions of the vector. This allows the database architect to "slice" the vector, storing only the first 64, 128, or 256 dimensions, without retraining the model.[6]

sqlite-vec natively supports this via the vec_slice() function:

SQL

```sql
SELECT rowid, distance
FROM vec_memories
WHERE embedding MATCH vec_slice(:query_vector, 128)
ORDER BY distance;
```

This query compares only the first 128 dimensions. If the model supports MRL (like nomic-embed-text-v1.5), this reduces the computational load by 3x (384 vs 128) with minimal accuracy loss. This is a powerful lever for tuning the performance/accuracy trade-off dynamically.

3.3 Pre-Filtering Performance

A common performance pitfall in vector search is the "Post-Filtering" trap: performing a vector search to find the top 100 items, and *then* filtering by metadata (e.g., user_id = 42). If the top 100 vectors belong to other users, the result set is empty.

sqlite-vec implements an optimized rowid IN (...) clause to handle **Pre-Filtering**.

SQL

```sql
SELECT rowid, distance
FROM vec_memories
WHERE rowid IN (SELECT rowid FROM memories WHERE user_id = :uid)
  AND embedding MATCH :query
ORDER BY distance;
```

In this execution plan, SQLite first executes the subquery using standard B-Tree indexes on the memories table to identify the valid set of rowids. The vector engine then restricts its brute-force scan *only* to those rows.[3] For an AI agent where memories are strictly partitioned by user_id or session_id, this optimization is critical, potentially reducing the scan space from 100,000 to a few hundred records, resulting in microsecond-level latency.

---

4. Full-Text Search (FTS5): The Lexical Anchor

While vector search excels at semantic "vibes" (matching "canine" to "dog"), it often fails at specific, rigorous keyword matching (matching "Error Code 503" or "Project Apollo"). An effective AI memory must implement Hybrid Search, which necessitates a robust Full-Text Search (FTS) layer. SQLite's FTS5 extension is the industry standard for this task.

4.1 The Tokenizer Dilemma: Trigram vs. Porter

The configuration of the FTS5 tokenizer is the single most important decision for the lexical layer. The default tokenizers are often ill-suited for the messy, ungrammatical, and code-heavy nature of "chat logs."

4.1.1 The Porter Tokenizer

The porter tokenizer implements the Porter Stemming Algorithm, which reduces words to their root form (e.g., "running", "ran", "runs" -> "run").

- **Pros:** It significantly reduces the vocabulary size and handles grammatical variations well.
- **Cons:** It is destructive. It cannot handle substrings. Searching for "run" matches "running," but searching for "unni" (a substring) returns nothing. Furthermore, it treats punctuation as delimiters, meaning email addresses or technical codes (e.g., user_id:1234) are often split into meaningless tokens.[12]

4.1.2 The Trigram Tokenizer

The trigram tokenizer, introduced in newer versions of SQLite (and fully mature in 2025), breaks text into sliding windows of three characters.

- **Example:** "hello" -> hel, ell, llo.
- **Pros:** It enables **substring matching** (LIKE '%ell%' behavior) at index speed. This is crucial for an AI agent. If a user asks, "What was that code that ended in 503?", a trigram index can find "Error-503" instantly. It is also language-agnostic and robust against typos.[13]
- **Cons:** The index size is larger. A trigram index can often be larger than the text corpus itself. However, with 100,000 records of chat text, this is typically still within the 100-200MB range, which is acceptable for the 6GB RAM constraint.

Recommendation: For an AI Assistant Memory system, the **Trigram** tokenizer is superior. The ability to recall partial strings (names, codes, URLs) is more valuable than the linguistic stemming of the Porter algorithm.

4.2 Scoring and The BM25 Quirk

FTS5 uses the Okapi BM25 algorithm for relevance scoring. However, developers must be aware of a specific quirk in the SQLite implementation: the bm25() function returns a value where *more negative* often implies a better match (or the sorting order logic requires inversion). Standard usage often multiplies the result by -1.0 to normalize it to a "higher is better" scale, which simplifies integration with vector scores.[15]

Correct FTS5 Query Pattern:

SQL

```
SELECT
  rowid,
  bm25(fts_memories) * -1.0 as score
FROM fts_memories
WHERE fts_memories MATCH :query
ORDER BY rank;
```

The ORDER BY rank clause is optimized in FTS5 to use the index statistics to sort results efficiently without loading the entire result set into memory.[15]

---

5. Hybrid Search: Fusing Semantics and Keywords

The "Hybrid Search" pattern combines the precision of FTS5 with the recall of sqlite-vec. The challenge lies in combining two disparate scores: the BM25 score (which is unbounded and distribution-dependent) and the Vector Distance (which is geometric).

5.1 Reciprocal Rank Fusion (RRF)

In 2025, Reciprocal Rank Fusion (RRF) has emerged as the standard for hybrid search, displacing complex score normalization logic (like min-max scaling). RRF ignores the absolute values of the scores and relies solely on the **rank** of the document in each result set.[17]

The RRF score for a document d is calculated as:

$$RRF(d) = \sum_{r \in R} \frac{1}{k + r(d)}$$

Where k is a smoothing constant (typically 60) and $r(d)$ is the position (rank) of the document in a specific result list (e.g., 1st, 2nd, 10th). This method ensures that a document appearing at the top of both lists (Vector and FTS) gets a significantly higher score than one appearing at the top of only one.

5.2 Implementation via SQL Common Table Expressions (CTEs)

Implementing RRF efficiently requires performing the fusion within the database engine to avoid moving large result sets to the application layer. We use CTEs to generate ranked lists and then join them.

Optimized Hybrid Query:

```
  SQL


WITH
```

```sql
-- 1. Get Top 100 via Vector Search
vec_results AS (
  SELECT
    rowid AS doc_id,
    distance,
    ROW_NUMBER() OVER (ORDER BY distance ASC) as rank_vec
  FROM vec_memories
  WHERE embedding MATCH :query_vector
  ORDER BY distance ASC
  LIMIT 100
),
-- 2. Get Top 100 via Full-Text Search
fts_results AS (
  SELECT
    rowid AS doc_id,
    bm25(fts_memories) as bm25_score,
    ROW_NUMBER() OVER (ORDER BY rank) as rank_fts
  FROM fts_memories
  WHERE fts_memories MATCH :query_text
  ORDER BY rank
  LIMIT 100
),
-- 3. Combine IDs (Emulating Full Outer Join)
combined_ids AS (
  SELECT doc_id FROM vec_results
  UNION
  SELECT doc_id FROM fts_results
)
-- 4. Calculate RRF Score
SELECT
  c.doc_id,
  m.content,
  (
    COALESCE(1.0 / (60 + v.rank_vec), 0.0) * :vec_weight +
    COALESCE(1.0 / (60 + f.rank_fts), 0.0) * :fts_weight
  ) as rrf_score
FROM combined_ids c
LEFT JOIN vec_results v ON c.doc_id = v.doc_id
LEFT JOIN fts_results f ON c.doc_id = f.doc_id
JOIN memories m ON c.doc_id = m.id
ORDER BY rrf_score DESC
LIMIT 20;
```

This query is highly performant because it pushes the heavy lifting (sorting and ranking) to the C-based engines of FTS5 and sqlite-vec, handling only the lightweight arithmetic of RRF in the final projection.

---

6. Embedding Models: The 2025 Landscape

The user's initial plan referenced all-MiniLM-L6-v2. While this model is a legendary workhorse, relying on it in 2025 for a new system is suboptimal.

6.1 The Limitations of MiniLM

all-MiniLM-L6-v2 was released in 2021. It produces 384-dimensional vectors and has a context window of 256 tokens (truncated at 384).

- **Context Limit:** 256 tokens is roughly one paragraph. If the AI agent attempts to store a user's detailed instruction or a summarized email that exceeds this length, the model effectively "lobotomizes" the memory, seeing only the first paragraph.
- **Performance:** On the Massive Text Embedding Benchmark (MTEB), MiniLM's performance has been surpassed by models that are barely larger in size but significantly advanced in architecture.[19]

6.2 Top Contenders for Embedded Systems in 2025

For a 6GB RAM system, we need models that are "Small" (< 500MB) but "Smart."

6.2.1 Nomic-Embed-Text-v1.5

This is the premier recommendation for 2025.

- **Context Window:** 8192 tokens. This is a transformative capability. It allows the agent to embed entire conversation histories, long documents, or complex function definitions as a single vector.
- **Matryoshka Support:** It outputs 768 dimensions by default but is explicitly trained to be sliced down to 256 or 384 dimensions. This offers the best of both worlds: use 256 dims for "chat log" search (speed) and 768 dims for "knowledge base" search (accuracy).[19]
- **Quantization:** It performs exceptionally well when quantized to Int8.

6.2.2 BGE-Micro-v2

If the 384-dimension constraint is hard (e.g., for compatibility with legacy systems), BGE-Micro-v2 is the modern replacement for MiniLM. It matches the 384-dimension output but consistently scores higher on retrieval benchmarks due to better training data and contrastive learning techniques.[22]

Verdict: Switch to **Nomic-Embed-Text-v1.5**. The 8k context window is invaluable for an AI memory system, ensuring that long-tail context is not lost during ingestion.

---

7. Schema Design Patterns: Episodic vs. Semantic Memory

A naive "one table for everything" approach fails to capture the nuance of human memory. Cognitive science—and effective AI engineering—distinguishes between **Episodic Memory** (autobiographical events) and **Semantic Memory** (facts and knowledge).

7.1 Episodic Memory Schema

Episodic memory logs the "stream of consciousness." It is time-series heavy, immutable, and high-volume.

SQL

```sql
CREATE TABLE episodes (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  session_id TEXT NOT NULL,
  role TEXT CHECK(role IN ('user', 'assistant', 'system')),
  content TEXT NOT NULL,
  token_count INTEGER,
  created_at INTEGER DEFAULT (unixepoch()),
  -- Foreign key to vector table via rowid implicitly
  -- Metadata for filtering
  importance_score REAL DEFAULT 0.5
);

-- Shadow Vector Table
CREATE VIRTUAL TABLE vec_episodes USING vec0(
  embedding float
);
```

7.2 Semantic Memory Schema

Semantic memory stores distilled facts. These are updated, reinforced, and decayed. This table is smaller but higher value.

SQL

```sql
CREATE TABLE facts (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  category TEXT,        -- 'user_preference', 'biography', 'world_knowledge'
  fact_text TEXT NOT NULL,
  confidence REAL,       -- 0.0 to 1.0
  source_episode_id INTEGER, -- Traceability
  last_accessed INTEGER, -- For Time Decay
  access_count INTEGER,  -- For Reinforcement
  FOREIGN KEY(source_episode_id) REFERENCES episodes(id)
);

CREATE VIRTUAL TABLE vec_facts USING vec0(
  embedding float
);
```

7.3 Advanced Mechanism: Time Decay and Reinforcement

An AI agent must "forget" irrelevant details to avoid context pollution. We implement **Exponential Time Decay** directly in the retrieval SQL query. This mimics biological memory: memories that are not accessed fade, while those accessed frequently are reinforced.[23]

The Scoring Formula:

$$Score = (Similarity \times \alpha) + (Importance \times \beta) \times e^{-\lambda \cdot \Delta t}$$

- **Similarity:** The vector distance (converted to similarity).
- **Importance:** A static score assigned at creation (e.g., "User name" = 1.0, "Weather talk" = 0.1).
- **Reinforcement:** access_count increases the base importance.
- **Decay:** The exponential term reduces the score as Δt (time since last access) increases.

SQL Implementation:

SQL

```
SELECT
    f.fact_text,
    vec.distance,
    f.access_count,
    -- Time Decay Calculation
    -- unixepoch() returns seconds.
    -- 0.00001 decay rate implies a "half-life" of roughly 19 hours.
    (f.access_count * EXP(-0.00001 * (unixepoch() - f.last_accessed))) as recency_score
FROM facts f
JOIN vec_facts vec ON f.id = vec.rowid
WHERE vec.embedding MATCH :query
ORDER BY (vec.distance * 0.7) + (recency_score * 0.3) DESC;
```

Implementation Note: SQLite does not have EXP() built-in by default. It requires the math extension (loadable via .load or compiled in). If unavailable, one can approximate decay using 1.0 / (1.0 + decay_rate * delta_time) or load a simple custom function.

---

8. Graph Data Structures and Recursive CTE Limits

To move beyond simple similarity, the agent needs to model relationships (e.g., "Project X" is_part_of "Initiative Y", "User" lives_in "City"). This requires a Graph structure.

8.1 The Edge Schema

In a relational database, a graph is simply a table of edges.

SQL

```
CREATE TABLE edges (
    source_id INTEGER,
    target_id INTEGER,
    relation_type TEXT,
    weight REAL,
    PRIMARY KEY (source_id, target_id),
    FOREIGN KEY(source_id) REFERENCES entities(id),
    FOREIGN KEY(target_id) REFERENCES entities(id)
);
```

```
CREATE INDEX idx_edges_reverse ON edges(target_id, source_id);
```

8.2 The Recursive CTE Challenge

The user query asks about "Graph CTE limits." Recursive Common Table Expressions are powerful but dangerous in a graph of 100,000 nodes.

1. **Recursion Depth:** SQLite has a hard limit on recursion depth, controlled by PRAGMA max_recursive_iterations (default is often 1000). In a "small world" graph, 6 degrees of separation covers everything, but a linear path can easily exceed 1000.
2. **Cycle Detection:** A graph with 100K nodes will have cycles. A naive CTE will enter an infinite loop until it hits the recursion limit.
3. **Performance Wall:** Recursive CTEs build a transient table (the "queue"). If a query is not constrained, this table can grow exponentially (e.g., traversing 3 hops where each node has 100 connections = 1,000,000 rows). This spills to disk, causing query times to spike from milliseconds to tens of seconds.[25]

8.3 Safe Traversal Pattern

To safely traverse a graph in SQLite, one must explicitly track the path to detect cycles and impose a hard depth limit.

SQL

```
WITH RECURSIVE traversal(id, path, depth) AS (
  -- Anchor: Start at a specific entity
  SELECT source_id, '/' |

| source_id |
| '/', 0
  FROM edges WHERE source_id = :start_node

  UNION ALL

  -- Recursive Step
  SELECT e.target_id, t.path |

| e.target_id |
| '/', t.depth + 1
  FROM edges e
  JOIN traversal t ON e.source_id = t.id
  WHERE t.depth < 3              -- Hard depth limit (e.g., 3 hops)
  AND instr(t.path, '/' |

| e.target_id |
| '/') = 0 -- Cycle detection
)
SELECT * FROM traversal;
```

Architectural Advice: Do not attempt "global" graph algorithms (like PageRank) inside SQLite CTEs on 100K nodes. Use CTEs only for "local" neighborhood lookups (e.g., "What is related to this memory?"). For global analysis, export the edge table to a specialized graph library like NetworkX (in Python) or a Graph DB, though for 100K nodes, in-memory NetworkX is likely sufficient and faster.[26]

---

9. Performance Tuning and Operational Best Practices

To extract maximum performance from this stack on 6GB RAM, specific operational tuning is required.

9.1 Memory Mapping (mmap)

The single most effective optimization for read-heavy vector search in SQLite is memory mapping. By mapping the database file into the process's virtual address space, SQLite can access the vector blobs as if they were raw memory arrays, bypassing the buffer cache overhead.

- **Command:** PRAGMA mmap_size = 300000000; (Set to ~300MB or larger, enough to cover the vector table).
- **Impact:** This can reduce vector scan latency by 20-30% by reducing system calls and memory copies.

9.2 WAL Mode and Concurrency

- **WAL (Write-Ahead Log):** Essential. PRAGMA journal_mode = WAL;. This allows the "Reader" (the LLM looking up memories) and the "Writer" (the agent saving new interactions) to operate concurrently without blocking each other.
- **Synchronous:** PRAGMA synchronous = NORMAL;. In WAL mode, this setting provides a massive performance boost for writes while maintaining durability against power loss (mostly).

9.3 The VACUUM Strategy

SQLite does not automatically reclaim disk space to the OS; it marks pages as "free." In a vector system where large blobs are frequently inserted and maybe deleted (consolidating memories), the database file can become fragmented.

- **Routine:** Execute PRAGMA optimize; before closing connections. Schedule a VACUUM; operation during low-activity periods (e.g., nightly). Note that VACUUM requires disk space equal to the size of the database to reconstruct the file.[28]

---

10. Alternatives Analysis: Why Not LanceDB or DuckDB?

While this report champions SQLite, due diligence requires comparing it to the alternatives mentioned in the user query.

10.1 LanceDB

- **Architecture:** Built on the Lance file format (columnar, zero-copy, Arrow-native). It is designed for multi-modal data (images, video, text) and scale.[29]
- **Pros:** It uses disk-based indices (IVF-PQ), meaning it does *not* need to load all vectors into RAM. It scales to hundreds of millions of vectors.
- **Cons:** It is a separate system/library to manage. It lacks the transactional rigor (ACID) of SQLite for the *metadata* and *relational* aspects (e.g., managing the graph edges or complex user logs).
- **Verdict:** For 100K records, LanceDB is overkill. The benefits of disk-based indexing don't materialize until the dataset exceeds RAM, which (at 100K vectors) it does not on a 6GB system.

10.2 DuckDB

- **Architecture:** OLAP (Online Analytical Processing) database.
- **Pros:** Incredible speed for aggregations (SELECT AVG(score) FROM memories).
- **Cons:** Poor performance for OLTP (Online Transactional Processing) workloads. Inserting a single chat log row is expensive in columnar stores. Furthermore, its vector index (HNSW) in the vss extension is currently **not persistent** in many configurations—it must be rebuilt when the database loads, which causes a massive startup delay.[4]
- **Verdict:** Unsuitable for a real-time, persistent chat memory system.

---

11. Conclusion

For a 2025-era embedded AI Agent managing 100,000 records on 6GB hardware, the **SQLite + sqlite-vec + FTS5** stack represents the optimal convergence of performance, simplicity, and capability.

The architectural analysis yields four key takeaways:

1. **Brute Force is Sufficient:** At 100,000 records, the sub-50ms latency of SIMD-accelerated brute force search renders the complexity of HNSW indexes unnecessary.
2. **Hybrid Search is Non-Negotiable:** Relying solely on vectors ignores the reality of precise information retrieval. FTS5 with **Trigram tokenization** and **RRF** fusion provides the necessary robustness.
3. **Models Matter:** Moving from MiniLM to **nomic-embed-text-v1.5** unlocks the context window required for storing rich, episodic memories without truncation.
4. **Schema is Intelligence:** The "intelligence" of the memory system lies not just in the vector search, but in the SQL schema design—specifically, the separation of Episodic and Semantic memory, and the implementation of time-decay ranking logic.

This stack transforms the humble SQLite file into a high-performance cognitive substrate, proving that "Local-First" AI is not just a privacy preference, but a viable engineering reality.

Component	**Recommendation**	**Justification**
Vector Engine	sqlite-vec (v0.2.x)	Dependency-free, WASM/Mobile support, optimized brute force.
Search Algo	Brute Force (Flat)	<50ms latency at 100K items; 100% recall; low memory overhead.
Embedding Model	nomic-embed-text-v1.5	8192 context window; Matryoshka slicing support.
Tokenizer	FTS5 trigram	Supports substring/fuzzy match for code/names; better than Porter for chat.
Hybrid Algo	Reciprocal Rank Fusion (RRF)	Robust rank-based fusion; no score normalization issues.
Graph Strategy	Recursive CTE with Depth Limit	Sufficient for local traversals; avoids external graph DB overhead.

Works cited

1. sqlite-vec (Vector Search in SQLite) version 0.2.3-alpha released : r/vectordatabase - Reddit, accessed on January 4, 2026, https://www.reddit.com/r/vectordatabase/comments/1py8905/sqlitevec_vector_search_in_sqlite_version/
2. asg017/sqlite-vec: A vector search SQLite extension that runs anywhere! - GitHub, accessed on January 4, 2026, https://github.com/asg017/sqlite-vec
3. I'm writing a new vector search SQLite Extension - Hacker News, accessed on January 4, 2026, https://news.ycombinator.com/item?id=40243168
4. Introducing sqlite-vec v0.1.0: a vector search SQLite extension that runs everywhere - Reddit, accessed on January 4, 2026, https://www.reddit.com/r/LocalLLaMA/comments/1ehlazq/introducing_sqlitevec_v010_a_vector_search_sqlite/
5. ANN (Approximate Nearest Neighbors) Index · Issue #25 · asg017/sqlite-vec - GitHub, accessed on January 4, 2026, https://github.com/asg017/sqlite-vec/issues/25
6. Introducing sqlite-vec v0.1.0: a vector search SQLite extension that runs everywhere | Alex Garcia's Blog, accessed on January 4, 2026, https://alexgarcia.xyz/blog/2024/sqlite-vec-stable-release/index.html
7. Open-source embedding models: which one to use? : r/LocalLLaMA - Reddit, accessed on January 4, 2026, https://www.reddit.com/r/LocalLLaMA/comments/1nrgklt/opensource_embedding_models_which_one_to_use/
8. How sqlite-vec Works for Storing and Querying Vector Embeddings | by Stephen Collins, accessed on January 4, 2026, https://medium.com/@stephenc211/how-sqlite-vec-works-for-storing-and-querying-vector-embeddings-165adeeeceea
9. How does quantization (such as int8 quantization or using float16) affect the accuracy and speed of Sentence Transformer embeddings and similarity calculations? - Milvus, accessed on January 4, 2026, https://milvus.io/ai-quick-reference/how-does-quantization-such-as-int8-quantization-or-using-float16-affect-the-accuracy-and-speed-of-sentence-transformer-embeddings-and-similarity-calculations
10. How does quantization (such as int8 quantization or using float16) affect the accuracy and speed of Sentence Transformer embeddings and similarity calculations? - Zilliz Vector Database, accessed on January 4, 2026, https://zilliz.com/ai-faq/how-does-quantization-such-as-int8-quantization-or-using-float16-affect-the-accuracy-and-speed-of-sentence-transformer-embeddings-and-similarity-calculations
11. Matryoshka (Adaptive-Length) Embeddings | sqlite-vec - Alex Garcia, accessed on January 4, 2026, https://alexgarcia.xyz/sqlite-vec/guides/matryoshka.html
12. Full-Text Search in SQLite: A Practical Guide | by Johni Douglas Marangon | Medium, accessed on January 4, 2026,

https://medium.com/@johnidouglasmarangon/full-text-search-in-sqlite-a-practical-guide-80a69c3f42a4
13. Full-Text Search: Using the Trigram Tokenizer Algorithm to Match Peoples Names, accessed on January 4, 2026, https://davidmuraya.com/blog/sqlite-fts5-trigram-name-matching/
14. SQLite FTS5 Tokenizers: `unicode61` and `ascii` - Audrey Roy Greenfeld, accessed on January 4, 2026, https://audrey.feldroy.com/articles/2025-01-13-SQLite-FTS5-Tokenizers-unicode61-and-ascii
15. SQLite FTS5 Extension, accessed on January 4, 2026, https://sqlite.org/fts5.html
16. How to interpret rank output from BM25? - Data Science Stack Exchange, accessed on January 4, 2026, https://datascience.stackexchange.com/questions/112889/how-to-interpret-rank-output-from-bm25
17. Introduce Hybrid Search API using SQLite FTS5 + Vector search · Issue #1158 · llamastack/llama-stack - GitHub, accessed on January 4, 2026, https://github.com/meta-llama/llama-stack/issues/1158
18. Hybrid full-text search and vector search with SQLite - Simon Willison's Weblog, accessed on January 4, 2026, https://simonwillison.net/2024/Oct/4/hybrid-full-text-search-and-vector-search-with-sqlite/
19. Best Open-Source Embedding Models Benchmarked and Ranked - Supermemory, accessed on January 4, 2026, https://supermemory.ai/blog/best-open-source-embedding-models-benchmarked-and-ranked/
20. Benchmark of 11 Best Open Source Embedding Models for RAG - Research AIMultiple, accessed on January 4, 2026, https://research.aimultiple.com/open-source-embedding-models/
21. Why are my benchmark results so different from the MTEB leaderboard? - Models, accessed on January 4, 2026, https://discuss.huggingface.co/t/why-are-my-benchmark-results-so-different-from-the-mteb-leaderboard/168305
22. AI Retrieval Is Too Expensive — Here's How Small Embeddings Cut Costs by 80%, accessed on January 4, 2026, https://medium.com/everyday-ai/ai-retrieval-is-too-expensive-heres-how-small-embeddings-cut-costs-by-80-14d5ed78f5d7
23. AI Agent Memory on Your Existing Postgres: We are back to SQL - GibsonAI, accessed on January 4, 2026, https://gibsonai.com/blog/ai-agent-memory-on-postgres-back-to-sql
24. Building AI Agents That Actually Remember: A Developer's Guide to Memory Management in 2025 | by Nayeem Islam | Medium, accessed on January 4, 2026, https://medium.com/@nomannayeem/building-ai-agents-that-actually-remember-a-developers-guide-to-memory-management-in-2025-062fd0be80a1
25. Slow running Recursive CTE · Issue #409 - GitHub, accessed on January 4, 2026, https://github.com/sqlitebrowser/sqlitebrowser/issues/409

26. Custom/persistent graph implementation? - Google Groups, accessed on January 4, 2026, https://groups.google.com/g/networkx-discuss/c/XUsKKr999KQ
27. What scalability issues are associated with NetworkX? - Stack Overflow, accessed on January 4, 2026, https://stackoverflow.com/questions/7978840/what-scalability-issues-are-associated-with-networkx
28. SQLite Release 3.51.1 On 2025-11-28, accessed on January 4, 2026, https://sqlite.org/releaselog/3_51_1.html
29. 5 Powerful Vector Database Tools for 2025 | Cybergarden, accessed on January 4, 2026, https://cybergarden.au/blog/5-powerful-vector-database-tools-2025