Review paper

Persistent memory architecture in agents using DBMS

An agent's long-term memory is a database. We measure ten ways of building it — from a JSON file to PostgreSQL with a vector index — on one corpus with known answers.

Submitted by

Harshit Khemani

Co-authors

Kush Ahuja, Madhav Bassi, Kushagra Agrawal

Submitted to

Dr. Poonam Sangwan

Read online

dbms-memory.khe.money

Corpus, harness and dataset at github.com/HKTITAN/dbms-agent-memory.

Abstract

An AI agent that persists anything across sessions is operating a database, whether or not it calls it one. It writes records, indexes them, retrieves a subset under a budget, and must survive a crash without contradicting itself. Current practice largely ignores this: agent memory is typically a JSON file or a standalone vector index, and the properties a database management system was built to provide — a schema, a query language, transactions, concurrency control, recovery — are absent by construction.

This paper asks what that costs. We model agent memory as an entity-relationship schema, normalise it to BCNF, and implement it across 10 storage architectures spanning three families: file stores, SQLite 3.53.0 embedded, and PostgreSQL 18.3 with `pgvector` and GIN. All 10 are measured on one corpus of 8,282 agent memories (210,568 tokens) rendered from 2,500 ground-truth facts, against 313 labelled recall queries partitioned into 8 classes.

Four results. First, **most of a realistic recall workload is not a similarity problem**: 61.7% of our queries (193 of 313) require a predicate, a join or an aggregate, and no top-*k* similarity search can express them. Second, **dense retrieval fails hardest on exactly what agents remember** — identifiers. On queries naming a record by its key, the pure vector arm scores 0.002 nDCG

against 0.790 for an inverted index; in every probe the embedding's top hit was a distractor that shared the query's grammatical shape, while the memory that answered it sat at median rank 271 of 8,282. Third, **the properties that separate the families are the classical ones**: under eight concurrent writers the file store lost 175 of 200 updates (87.5%) where both DBMS arms lost none, and a crash during a whole-document rewrite left the entire store unreadable in 3 of 20 trials. Fourth, **normalisation is not bookkeeping here**: a fact is restated across 3.69 memories on average, so a correction applied through top-10 retrieval leaves 63.5% of the restatements asserting the old value — contradictions the agent will later retrieve and believe.

The practical conclusion is narrower than “use a database”. The best-scoring arm was SQLite · FTS5 + vectors at 0.726 nDCG, but SQLite · FTS5 inverted index reached 0.723 at 0.44 ms median latency and 497.5 bytes per memory — 8.4× less storage than the vector-bearing arms, whose embeddings dominate the store. Vectors earn their cost on paraphrase and nowhere else.

8,282 Memories in corpus	**10** Architectures measured
62% Queries needing SQL	**0.002** Vector nDCG on identifiers
63.5% Stale after top-k repair	

All figures produced by `tools/capture.mjs` on 13th Gen Intel(R) Core(TM) i5-13450HX, 16 cores, 32 GB, win32/x64. Captured 2026-08-09. Embeddings from Xenova/all-MiniLM-L6-v2 (384 d); token counts from the Xenova/gpt-4o tokenizer.

1. Introduction

A language model has no memory. Each request is answered from the tokens in front of it, and when the context window closes, everything in it is gone. An *agent* — a model wrapped in a loop that runs over hours or months — has to supply that memory from outside. It writes down what happened, and later it reads some of it back.

Stated that way, the problem is immediately familiar. Records are inserted. They are indexed so they can be found again. A query selects a subset under a budget. Concurrent writers must not overwrite each other. A crash must not leave the store asserting something that was never true. This is the problem a database management system exists to solve, and it has been studied for fifty years[1][3].

Agent frameworks have largely arrived at a different answer. Memory is usually a file of JSON, or a vector index holding one embedding per remembered utterance, retrieved by cosine similarity. Neither has a schema, a query language, a transaction, or a recovery protocol. The question this paper asks is direct: **what does an agent lose by storing its memory outside a DBMS, and which of the DBMS's properties actually matter?**

We answer it by measurement rather than argument. We build one corpus of agent memories with known ground truth, implement 10 storage architectures over it, and score them on the same 313 queries — then subject the same three families to a crash, to concurrent writers, and to a fact that changes after it has been remembered 3.69 times.

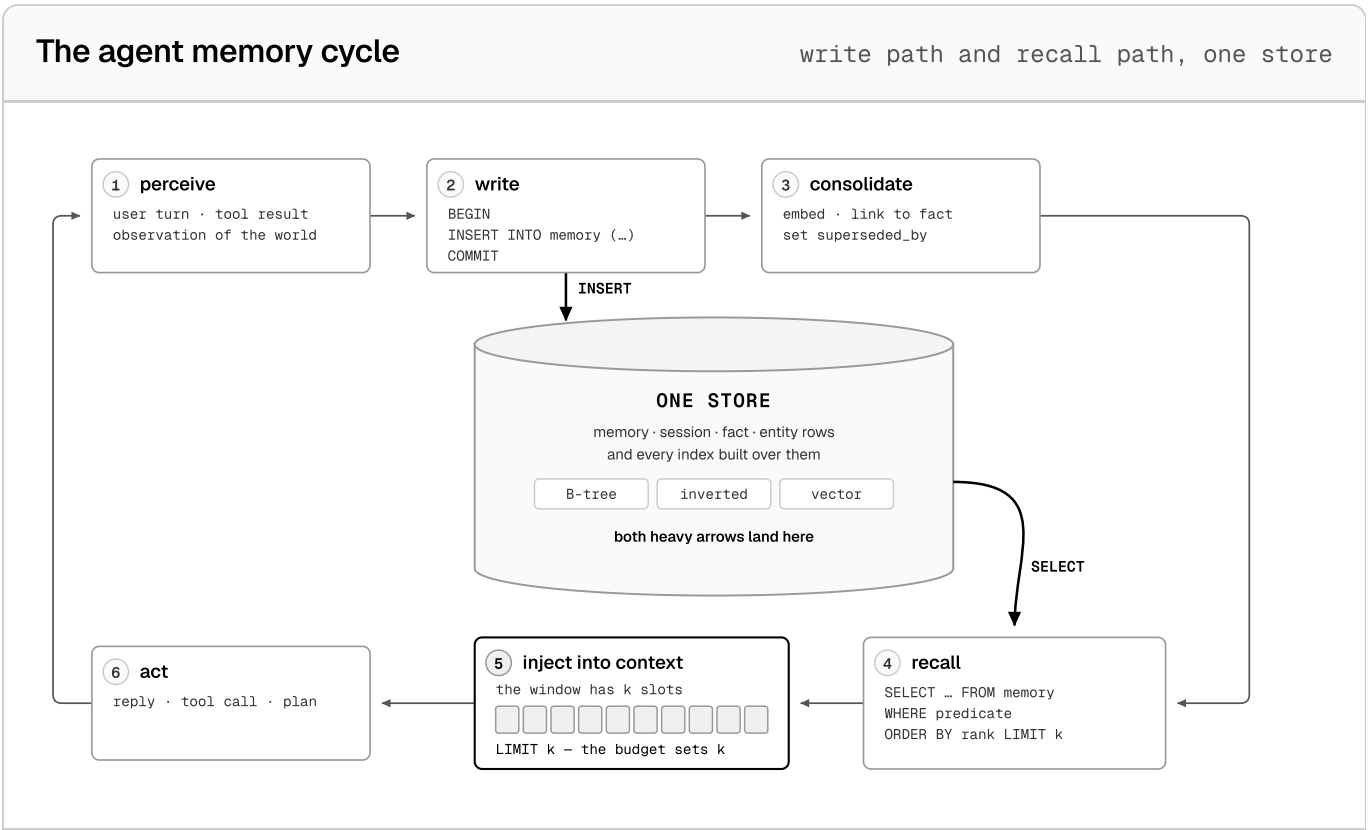


Figure 1. The agent memory cycle. Every arrow crossing the store boundary is a database operation; the context-window budget on the recall path is what makes retrieval a top-*k* problem rather than a scan.

2. What agent memory is

The word covers three different things, and conflating them is the first source of confusion. The division follows Tulving's[12] and is now standard in agent architectures[13]:

episodic — Something that happened, tied to a time and a session.

semantic — A standing fact about the world, time-independent.

procedural — An ordered way of doing something.

In the relational model these are not three stores. They are one relation with a discriminator: a disjoint, total specialization on `kind`. That is a design decision with consequences we return to in §4, and it is the first place where having a data model at all changes what the system can do — a query can ask for procedural memory only, and the engine can use an index to answer it.

What an agent actually writes is narrower than “everything it saw”. In our corpus a memory is a short natural-language restatement of one fact, produced during one turn of one session. It is roughly 25.4 tokens. It carries a provenance (`shell_output` and similar), a confidence, a position in time, and — critically — it may later be contradicted by something the agent learns afterwards.

Table 1. Six memories from the corpus, as stored. These are the rows every architecture in this paper was given.

id	session	turn	kind	body	day	tokens
M-000000	S-0065	1	procedural	RUN-409 — rolling back a migration on reconciler-5: 1. stop the deploy 2. run the down migration in a transaction 3. verify the constraint set 4. redeploy the previous image	62	45

id	session	turn	kind	body	day	tokens
M-000001	S-0177	1	semantic	ADR-568 rejected follower reads and per-tenant routing in favour of pinned primary.	171	17
M-000002	S-0319	1	semantic	We settled the secret rotation argument in search-6: quarterly manual won.	305	15
M-000003	S-0047	1	semantic	PER-190 — Ravi Adeyemi is staff engineer on data.	46	14
M-000004	S-0019	1	episodic	Postmortem for INC-3742 — the query planner fell back to a sequential scan. The underlying problem was a missing index after a migration.	18	30
M-000005	S-0212	1	semantic	SVC-5093 reconciler-5: max_pool_size is 250. Changing it needs a restart.	206	23

3. How agent memory is built today

Two architectures dominate, and we implement both rather than describe them.

The file store. Memories are appended to a JSON or JSONL file and recall reads the whole thing, scoring by keyword overlap. It has no index, so recall is linear in the corpus; it has no transaction, so a crash mid-write is whatever the filesystem left behind; and it has no concurrency control, so two writers race. We implement the careful variant (append-only) and, for the durability experiment, the common one (rewrite the whole document on every change).

The vector index. Each memory is embedded once and recall is a top-k nearest-neighbour search in that space[9]. This is the architecture most often described as “giving the agent memory”, and it is genuinely good at one thing: finding a memory that means the same as the query while sharing none of its words. We implement it twice — once with no filtering at all, and once with metadata post-filtering, because every production vector store offers the latter and comparing against the former alone would be a straw man.

Against these we put the same logical schema in two database engines: SQLite 3.53.0 in process[7], and PostgreSQL 18.3[6] with the pgvector (https://github.com/pgvector/pgvector)[10] extension. Each family is measured with progressively more indexing, so the paper can attribute a result to an access method rather than to a product.

Table 2. What each of the 10 architectures can express, as declared by its loader and confirmed by the query plans in the appendix. A check is an unqualified capability; a word is a qualified one and names the mechanism; — is an absence. No units — this table is definitional, and it does not depend on the corpus.

Engine	Predicates	Joins	Aggregates	Transactions	Vector search
JSONL file `file-jsonl`	—	—	—	—	—
Flat vector index `file-vec`	—	—	—	—	✓
Vector index + post-filter `file-vec-meta`	post	—	—	—	✓
SQLite · B-tree only `sqlite-btree`	✓	✓	✓	✓	—
SQLite · FTS5 inverted index `sqlite-fts`	✓	✓	✓	✓	—
SQLite · FTS5 + vectors `sqlite-hybrid`	✓	✓	✓	✓	scan
Postgres · B-tree only `pg-btree`	✓	✓	✓	✓	—

Engine	Predicates	Joins	Aggregates	Transactions	Vector search
Postgres · GIN inverted index `pg-gin`	✓	✓	✓	✓	—
Postgres · pgvector HNSW `pg-hnsw`	✓	✓	✓	✓	`hnsw`
Postgres · GIN + HNSW hybrid `pg-hybrid`	✓	✓	✓	✓	`hnsw`

`post` — the predicate is applied to the rows the index already returned, so it can only remove, never recover. `scan` — cosine is computed over every stored vector, because no access method exists for it. `hnsw` — an approximate nearest-neighbour graph the planner can combine with an ordinary predicate. 3 of 10 architectures offer both unqualified predicates and a vector access method.

Table 2 is the paper's argument in one grid, and everything after it is the measurement of what those columns are worth.

---

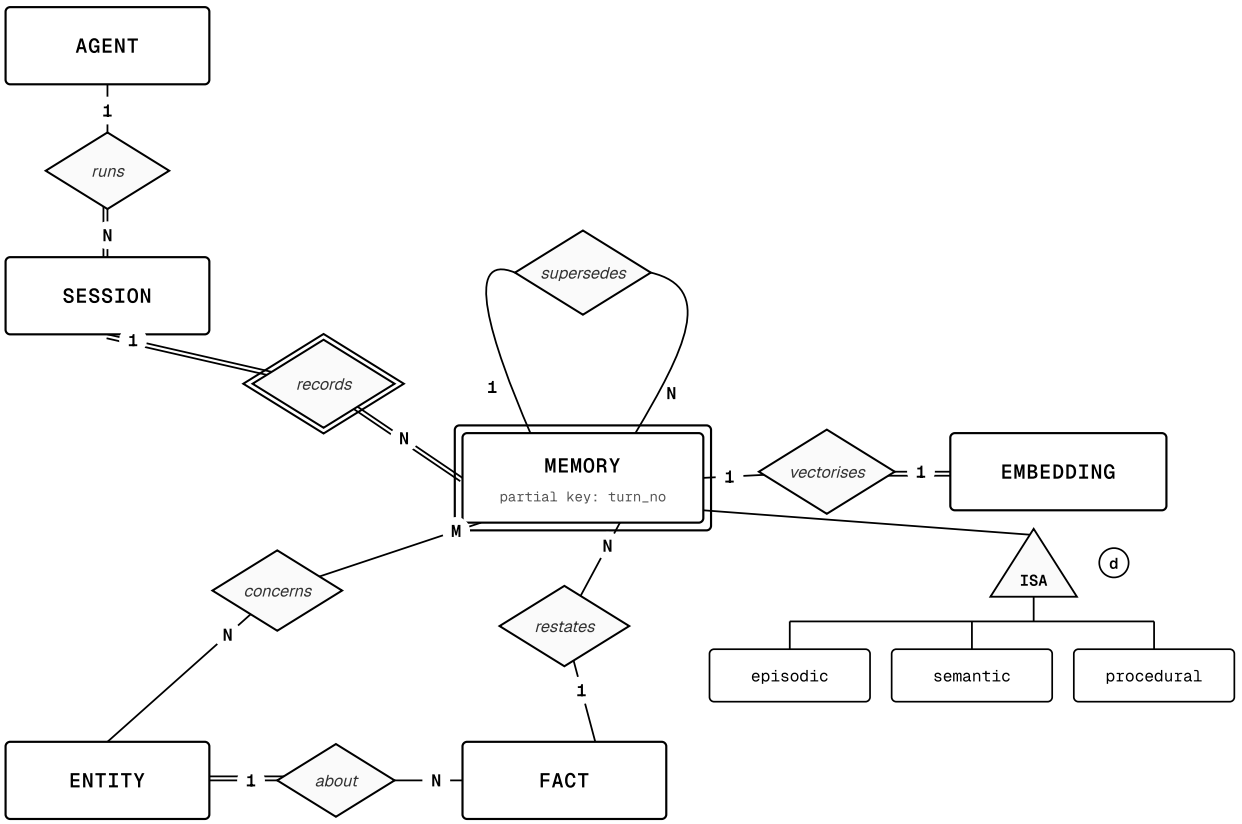
4. The data model

Before anything can be measured, the memory has to have a shape. We give it one using the entity-relationship model[2], then normalise it.

4.1 Entities and relationships

Seven entities. The one that matters is `MEMORY`, and it is a **weak entity**: a memory has no identity apart from the session that produced it, so its natural key is `(session_id, turn_no)` with `turn_no` as the partial key. This is not a modelling nicety. A vector store treats each memory as a free-floating document with a global identity, and every question of the form “what did I learn *in that session*” becomes unanswerable as a direct consequence.

Conceptual model — Chen notation



Double rectangle = weak entity · double diamond = identifying relationship · double line = total participation · (d) = disjoint specialization. MEMORY is identified by (session_id, turn_no).

Figure 2. The conceptual model in Chen notation, and the same model as tables. Both views are generated from `engines/schema.mjs` — the file the loaders execute — so every box is a table that was created and every dashed edge a foreign key that was enforced during measurement.

Three features of the diagram carry weight later:

The identifying relationship *records* is drawn double because `SESSION` supplies part of `MEMORY`'s key. Participation is total on both ends: a memory without a session cannot exist.

The recursive relationship *supersedes* runs from `MEMORY` to itself. It records that a later memory corrects an earlier one, and it is the structure that distinguishes what the agent *currently believes* from what it has *ever written down*. No similarity function can recover it, because a superseded memory matches a query at least as well as its replacement — usually better, since the replacement carries the word “correction”.

The specialization on `kind` is disjoint and total: every memory is exactly one of episodic, semantic or procedural.

4.2 Normalisation

The file baselines store a self-contained record per memory: session metadata, agent model, entity list and the restated fact all inline. That is the unnormalised form, and walking it up to BCNF names exactly what each file architecture is giving up.

Normalization ladder

5 rungs · UNF → BCNF

UNF

memory_blob(memory_id, session_id, turn_no, agent_model, session_day, body, entities[], fact_text, fact_id, source, confidence)

A relation is in 1NF when every attribute holds a single atomic value.

violates entities[] is a repeating group held inside one attribute.

> ANOMALY IT LEAVES BEHIND
> You cannot ask "which memories mention SVC-4470" without parsing an array in application code, so the DBMS cannot index it.

→ Project the repeating group into MEMORY_ENTITY, keyed by (memory_id, entity_id).

1NF

memory_flat(memory_id, session_id, turn_no, agent_model, session_day, body, fact_text, fact_id, source, confidence)

A relation is in 2NF when no non-prime attribute is partially dependent on a candidate key.

violates session_day depends on session_id alone, which is only part of the natural key (session_id, turn_no).

> ANOMALY IT LEAVES BEHIND
> A session's date is restated on every memory it produced. Correcting it means rewriting every row, and missing one splits the session in two.

→ Move session_day into SESSION.

2NF

memory_2nf(memory_id, session_id, turn_no, agent_model, body, fact_text, fact_id, source, confidence)

A relation is in 3NF when no non-prime attribute is transitively dependent on a candidate key.

violates memory_id → session_id → agent_model is a transitive dependency.

> ANOMALY IT LEAVES BEHIND
> The agent model is stored once per memory rather than once per session. It cannot be corrected atomically, and a session with no memories cannot record which model ran it at all.

→ Move agent_model into AGENT and reference it from SESSION.

3NF

memory_3nf(memory_id, session_id, turn_no, body, fact_text, fact_id, source, confidence)

A relation is in BCNF when every determinant is a candidate key.

violates fact_id → fact_text, and fact_id is not a superkey of MEMORY.

> ANOMALY IT LEAVES BEHIND
>
> This is the one that hurts an agent. The same fact is restated across many memories; when the fact changes, every restatement is now a claim the agent will retrieve and believe. §6.7 measures how many contradictions this produces.

→ Move fact_text into FACT and leave MEMORY holding only the reference.

BCNF

memory(memory_id, session_id, turn_no, seq, kind, body, fact_id, created_day, source, confidence, superseded_by)

Every determinant is a candidate key.

`terminal form` No anomaly is left to remove: every determinant is a candidate key.

This is the schema the SQLite and Postgres arms were measured on.

Each rung names the form the relation is *already* in, the rule the next form imposes, and the anomaly that survives until you climb. Read the accented boxes alone and you have the argument: an unnormalised memory store does not merely waste bytes, it stores the same claim many times, and every stale copy is a sentence the agent will retrieve and believe.

Figure 3. From the unnormalised memory blob to BCNF. The final step is the one that decides an agent's behaviour rather than its disk usage: `fact_id → fact_text` is a dependency on a non-key attribute, and leaving it in place is what §6.10 measures.

The functional dependencies that survive in the final schema are the ones a key determines and nothing else:

Table 3. Functional dependencies in the BCNF schema. Every determinant is a candidate key.

Relation	Determinant	Determines	Note
memory	memory_id	session_id, turn_no, seq, kind, body, fact_id, created_day, source, confidence, superseded_by	—
memory	session_id, turn_no	memory_id	the natural key of the weak entity
session	session_id	agent_id, started_day	—
agent	agent_id	model, family	—
fact	fact_id	subject_id, predicate, object_value, memory_kind, valid_from_day	—
embedding	memory_id	model, dim, vec	—

5. Methodology

5.1 Ground truth that is not circular

A retrieval benchmark is only as good as its labels, and labels chosen by looking at results are worthless. We invert the usual order. First we build a world of entities — services, incidents, configuration records, decisions, people. From it we derive 2,500 **facts**, each a subject-predicate-object triple with a validity interval. Each fact is then *rendered* into one or more memories: short natural-language restatements an agent would plausibly have written, in different surface forms, scattered across 333 sessions.

Relevance is therefore definitional rather than judged: the memories relevant to a query about a fact are exactly the memories rendered from that fact, intersected with the query's structural predicate. Nothing is scored by eye. The generator is seeded, so the corpus reproduces byte for byte.

Two deliberate contaminants make the benchmark hard. **Distractors** (1,218 memories, 15% of the corpus) mention a record's identifier while asserting nothing about it — the agent-memory equivalent of “checked X, unrelated”. And **supersessions** (844 memories) revise a fact after it was first recorded, so the best-matching text and the currently true statement are different rows.

5.2 Query classes

The 313 queries are partitioned into 8 classes chosen to stress different machinery. The column that matters is the last one: whether the query can be answered by a top-*k* similarity search alone, with no predicate, no join and no aggregate.

Table 4. The 313 queries of the workload by class, with the predicate that decides each class and the reason it can or cannot be answered by nearest-neighbour search over the memory text. Share is of the whole query set; the predicate column names the field and operator the correct answer depends on, or — where correctness depends only on the text.

Query class	Queries	Share	Similarity-expressible	Deciding predicate	Rationale
`lexical`	40	12.8%	✓	—	Rare identifier present verbatim in the target memories.
`semantic`	40	12.8%	✓	—	Paraphrase sharing no rare tokens with the target memories.
`temporal`	33	10.5%	—	`day <=`	Correct answer depends on a range restriction the index cannot apply.
`provenance`	40	12.8%	—	`sessionId =`	An equality join on a foreign key. Text similarity is irrelevant to correctness.
`currency`	40	12.8%	—	`supersededBy is null`	Superseded restatements match the query text at least as well as the current one.
`aggregate`	40	12.8%	—	`factId count`	A cardinality question. Top-k truncation makes the answer wrong by construction.
`negation`	40	12.8%	—	`supersededBy is null`	The excluded rows are the highest-similarity rows.
`hybrid`	40	12.8%	✓	—	Rewards a retriever that can use both an exact token and a paraphrase.
All classes	**313**	**100.0%**	**120 of 313**		

120 queries (38.3%) are similarity-expressible. The remaining 193 (61.7%) turn on a range restriction, a foreign-key equality, a null test or a count — four operators an embedding index does not have. That fraction, not any quality score, is the ceiling on a similarity-only store.

5.3 Validating the corpus before using it

A benchmark that claims “lexical queries share identifiers and semantic ones do not” should demonstrate it rather than assert it, because if the property fails, every downstream comparison measures nothing. Table 5 reports what the corpus actually has.

Table 5. Lexical and embedding evidence for each of the 8 query classes, measured over the 313 labelled queries and the 1,775 query–memory pairs their ground truth contains. Term overlap is the fraction of query terms present in a relevant memory; identifier overlap is the same fraction restricted to rare tokens (ticket and ADR identifiers); cosine is over the 384-dimensional `Xenova/all-MiniLM-L6-v2` embedding. The final column records whether the class can be expressed as a similarity query at all.

Query class	Queries	Relevant / query	Term overlap	Identifier overlap	Mean cosine	Similarity-expressible
`lexical`	40	2.60	32.2%	64.4%	0.277	✓
`semantic`	40	1.77	58.6%	66.9%	0.738	✓
`temporal`	33	2.64	22.5%	32.8%	0.400	—
`provenance`	40	24.10	3.3%	0.0%	0.204	—
`currency`	40	1.00	33.3%	100.0%	0.510	—
`aggregate`	40	7.00	11.9%	45.7%	0.251	—
`negation`	40	1.00	25.0%	100.0%	0.589	—
`hybrid`	40	4.72	30.2%	63.8%	0.627	✓
Baseline — 400 random memory pairs					**0.266**	

Read the cosine column against the baseline row, not against 1.0: two unrelated memories already score 0.266, so a class carries embedding signal only to the extent it clears that floor. 2 of the 8 classes average at or below it (provenance, aggregate) — for those the embedding is not a weak signal, it is no signal, and no retriever built on it can be tuned into correctness.

The design holds where it needs to. Lexical queries share the target's identifier but sit at cosine 0.277, barely above the 0.2655 random-pair baseline — they name the record without describing it. Semantic queries invert exactly that: cosine 0.738 with an identifier overlap of 0.00. A retriever that wins one and loses the other is telling us about its access method, which is the point.

5.4 Instrumentation

One harness, `tools/capture.mjs`, drives every measurement into a single JSON document. Engines return an ordered list of memory ids and nothing else; precision, recall, MRR and nDCG are computed centrally, so no engine can flatter itself. Latency is the median of three timed runs after a warm-up. Embeddings are computed once and shared, so a retrieval difference can never be an embedding difference. Storage is read from the engines' own accounting — `dbstat` for SQLite, `pg_class` for Postgres — not from file sizes.

Both engines run without a server: SQLite through Node's built-in `node:sqlite`, and PostgreSQL through PGlite (https://pglite.dev/)[11], a genuine Postgres 18.3 build compiled to WebAssembly. The whole study therefore reproduces from `npm install`, at the cost discussed in §10.

6. Results

6.1 Overall retrieval quality

Aggregated over all 313 queries at k=10, the ordering is already informative — and not the ordering the current literature would predict.

Table 6. Retrieval quality, latency and size for the 10 storage architectures, each measured over the same 313 queries against the same 8,282 memories at k = 10. Quality columns are means over all queries; latency is per query, measured end to end inside the process; size is the whole store on disk after the load. Bold marks the best value in each column — highest for quality, lowest for latency and size.

Engine	Index	nDCG@10	Recall	MRR	p50	p95	Bytes / memory	Total size
JSONL file append-only text file	none	0.516	0.547	0.605	19.4 ms	25.7 ms	**328**	**2.6 MB**
Flat vector index in-process float array	vector	0.161	0.200	0.192	5.4 ms	6.3 ms	1,864	14.7 MB
Vector index + post-filter in-process float array with metadata post-filter	vector	0.186	0.229	0.234	6.5 ms	7.5 ms	1,864	14.7 MB
SQLite · B-tree only SQLite (node:sqlite)	btree	0.668	0.629	0.835	3.8 ms	20.1 ms	429	3.4 MB
SQLite · FTS5 inverted index SQLite (node:sqlite)	inverted	0.723	0.653	**0.886**	**0.44 ms**	**1.9 ms**	498	3.9 MB
SQLite · FTS5 + vectors SQLite (node:sqlite)	hybrid	**0.726**	**0.773**	0.745	6.4 ms	11.0 ms	2,569	20.3 MB
Postgres · B-tree only PostgreSQL 18.3 (PGlite)	btree	0.668	0.629	0.835	26.1 ms	152 ms	577	4.6 MB
Postgres · GIN inverted index PostgreSQL 18.3 (PGlite)	inverted	0.704	0.634	0.882	1.0 ms	23.3 ms	827	6.5 MB
Postgres · pgvector HNSW PostgreSQL 18.3 (PGlite)	vector	0.393	0.346	0.420	1.5 ms	2.3 ms	3,927	31.0 MB
Postgres · GIN + HNSW hybrid PostgreSQL 18.3 (PGlite)	hybrid	0.700	0.742	0.735	2.7 ms	26.9 ms	4,177	33.0 MB

Bytes per memory is given as a plain byte count rather than through the size formatter: the column spans 328 B to 4,177 B, and switching half the column into kilobytes would break the comparison the column exists to support. All 10 configured engines completed the run.

Figure 4. Every arm on the same corpus and the same labels. The two pure-similarity arms are last.

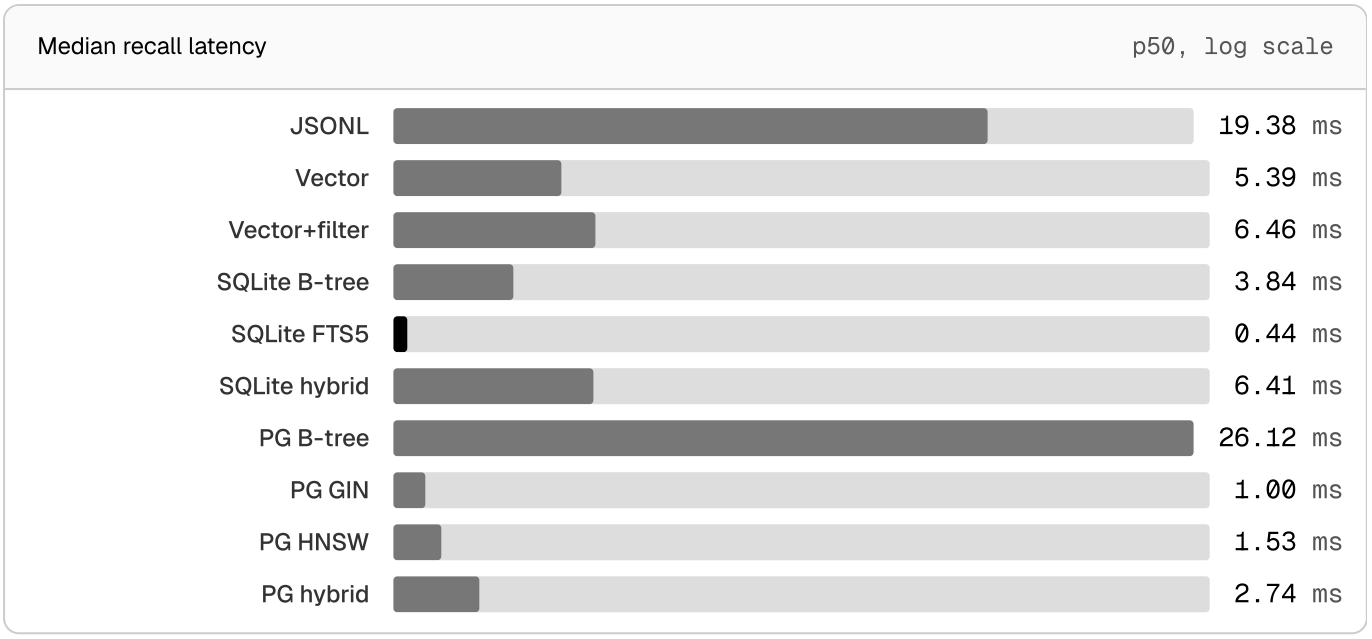


Figure 5. The unindexed file scan is 44.3× slower than an inverted index over the same 8,282 memories.

6.2 The central result: quality depends on the question

Aggregate scores hide the finding. Broken down by query class, the arms do not merely differ in degree — the ordering inverts, and each architecture has classes on which it is essentially unable to answer.

Columns marked “needs SQL” cannot be expressed as a top-k similarity search at all.

	lexical	semantic	temporal needs SQL	provenance needs SQL	currency needs SQL	aggregate needs SQL	negation needs SQL	hybrid
JSONL append-only text file	0.61	0.55	0.27	0.01	1.00	0.17	1.00	0.49
Vector in-process float array	0.00	0.56	0.16	0.00	0.07	0.01	0.03	0.44
Vector+filter in-process float array with metadata post-filter	0.00	0.56	0.32	0.01	0.07	0.06	0.03	0.44
SQLite B-tree SQLite (node:sqlite)	0.61	0.55	0.41	0.54	1.00	0.71	1.00	0.49
SQLite FTS5 SQLite (node:sqlite)	0.79	0.65	0.47	0.56	1.00	0.71	1.00	0.55
SQLite hybrid SQLite (node:sqlite)	0.62	0.69	0.60	1.00	0.65	1.00	0.64	0.59
PG B-tree PostgreSQL 18.3 (PGlite)	0.61	0.55	0.41	0.54	1.00	0.71	1.00	0.49
PG GIN PostgreSQL 18.3 (PGlite)	0.82	0.61	0.40	0.56	1.00	0.71	1.00	0.49
PG HNSW PostgreSQL 18.3 (PGlite)	0.00	0.50	0.18	1.00	0.04	1.00	0.01	0.37
PG hybrid PostgreSQL 18.3 (PGlite)	0.65	0.61	0.50	1.00	0.62	1.00	0.65	0.54

nDCG@10 0.00 1.00 — not applicable

Figure 6. nDCG@10 by engine and query class. Reading down a marked column shows what a predicate is worth; reading across the `Vector` row shows what its absence costs.

Three things happen in that grid.

Pure similarity collapses on structural questions. Averaged over the 5 classes requiring a predicate, join or aggregate, Vector reaches 0.057 nDCG. Its provenance score — "everything I recorded during session S" — is 0.000, because the question has no similarity content whatsoever. The answer is defined by a foreign key.

The same index inside a DBMS recovers most of it. PG HNSW uses the identical embeddings and an HNSW graph, and averages 0.446 on those classes — reaching 1.000 on provenance, because the planner applies the predicate and the vector index is simply not consulted. What changed is not the retrieval; it is that a query language existed to state the constraint.

Adding a vector index can make things worse. On *currency* — "what is the current position on X" — the B-tree arms score 1.000, because `superseded_by IS NULL` is exactly the right answer. The hybrid arms score 0.621: rank fusion re-ranks away from a filter that was already correct. Hybrid retrieval is not uniformly better, and treating it as a default costs accuracy on precisely the questions a predicate settles.

The *aggregate* class makes the point without any ranking at all. Asked how many times a fact was recorded, 7 of the 10 arms return the exact cardinality — every arm with a `SELECT COUNT(*) … GROUP BY`. The file arms return none, and cannot: a top-*k* list truncated at 10 is the wrong shape of answer to a counting question.

6.3 Why dense retrieval fails on identifiers

The lexical column deserves its own explanation, because a score of 0.002 invites the reader to assume a bug. It is not a bug, and the mechanism is worth seeing.

In 100% of probed queries, the memory the embedding ranked first was a *distractor* — a record that mentions the identifier while explicitly disclaiming it. The memory that actually answered the question sat at a median rank of 271 out of 8,282.

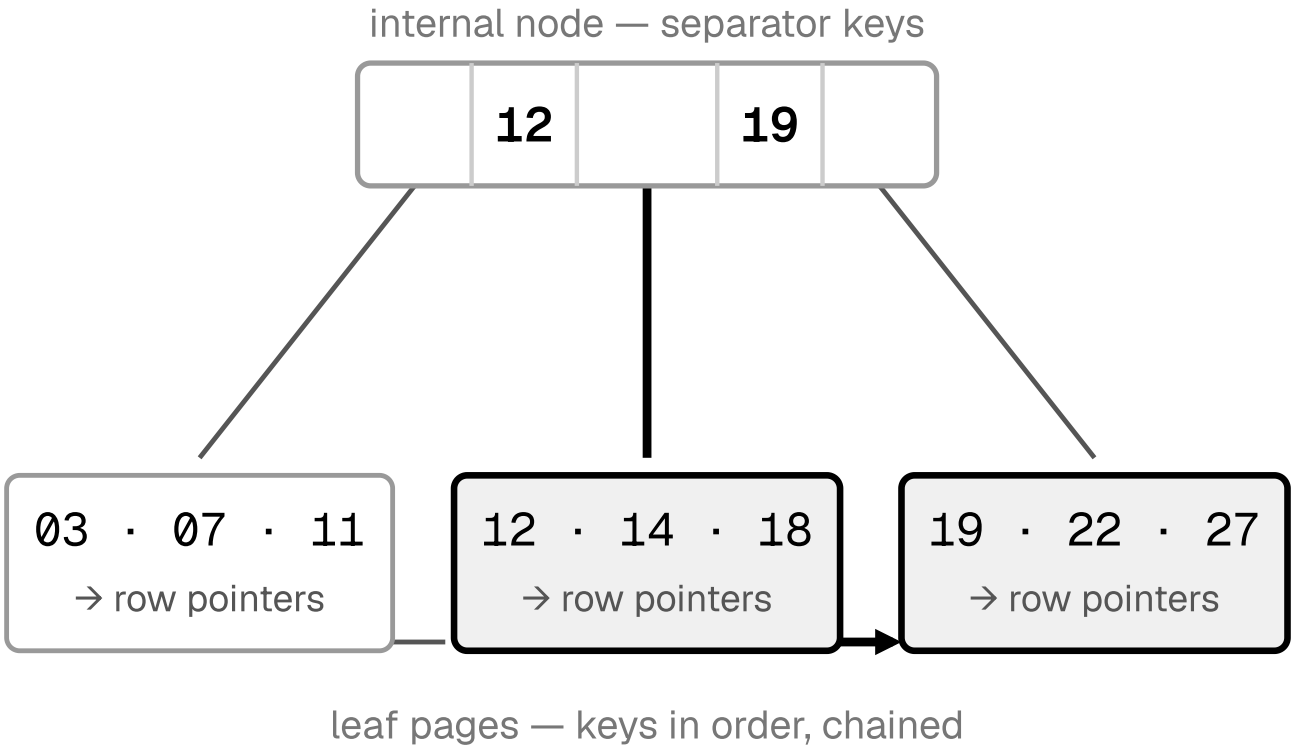
The reason is structural. "What do we know about ADR-297?" is a question *about a lookup*, and the distractor is a sentence *about a lookup*. They share their grammatical shape, which is most of what survives mean-pooling into 384 dimensions. The identifier itself is a handful of subword pieces averaged in with everything else, and it carries almost no weight. An inverted index has the opposite bias: a rare term is the most informative thing in the query, which is why the same corpus yields 0.790 nDCG for BM25.

What an access method physically is

B-tree · inverted · HNSW

B-tree

balanced search tree over one ordered key




```
created_day BETWEEN 12 AND 19
```

Descend once, then walk the chain.

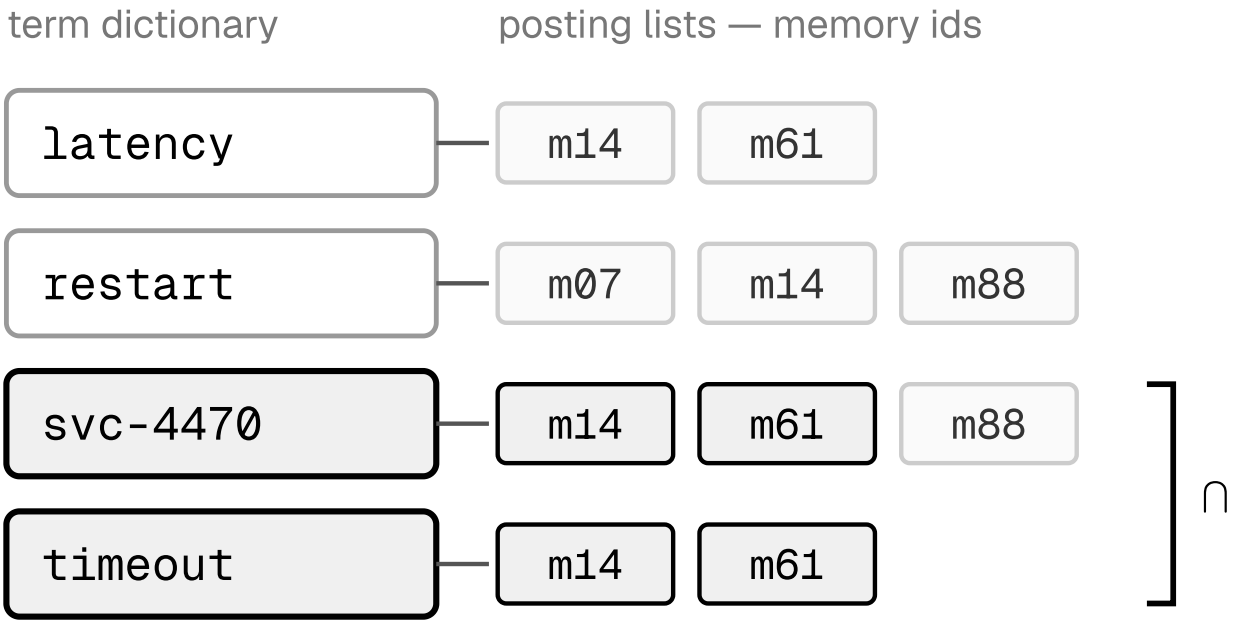
Cost is the answer, not the table.

answers ranges and equalities on the indexed column, in key order

cannot anything about what the text says — the key is a value, not a meaning

Inverted index

`term dictionary → posting lists`



`MATCH 'svc-4470 AND timeout'`

Intersect the two lists → {m14, m61}.

Rank the survivors by BM25.

The dictionary holds strings, not meaning.

answers **documents containing given terms, ranked by term statistics**

cannot **a paraphrase that shares no term — nothing files it under the query's words**

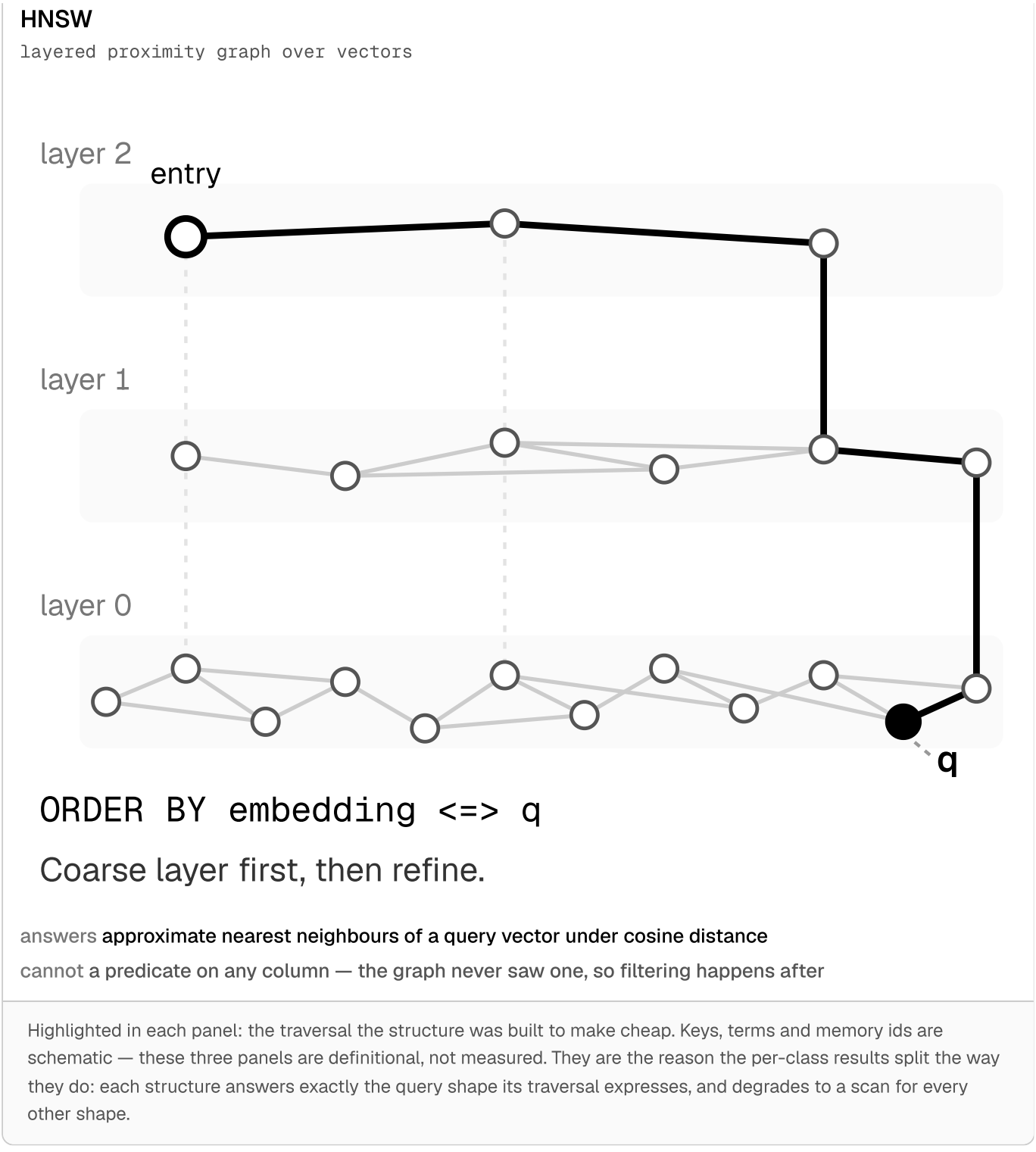
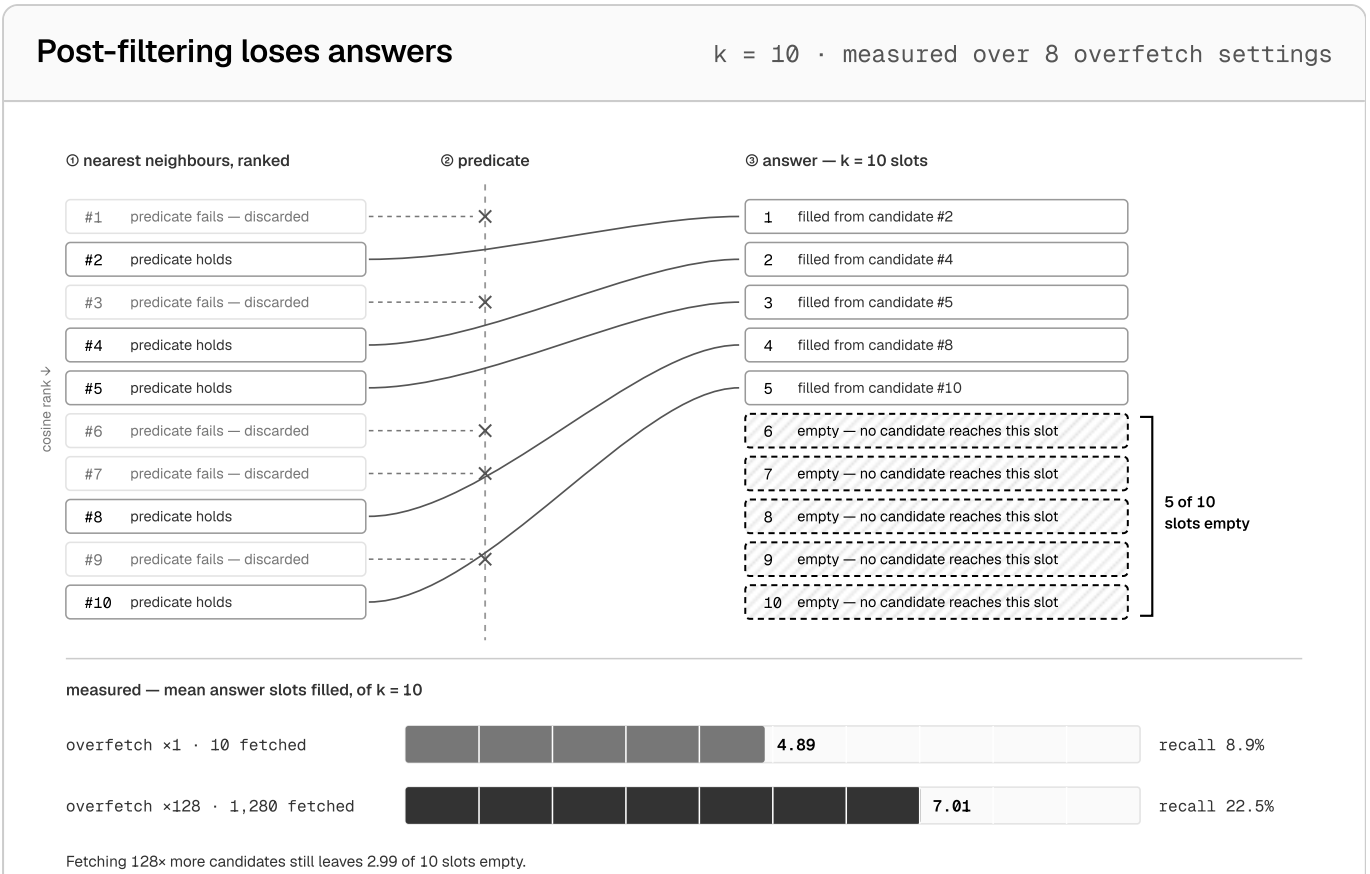


Figure 7. What each access method physically is, and the question shape it cannot answer. The per-class results in Figure 6 follow from these three structures.

6.4 Metadata filtering does not close the gap

The obvious objection to §6.2 is that production vector stores support metadata filters. They do — but as a *post*-filter over an already-ranked candidate list, and that is a different operation from a predicate the planner may apply first.

With no overfetch, asking for 10 results returns 4.89 on average: 48.9% of the requested slots are filled, because the filter can only remove candidates, never introduce them. Raising the overfetch factor helps, but slowly and at a price — at ×128 the search touches 15.5% of the corpus to reach recall 0.225, which is to say it is becoming the sequential scan the index was adopted to avoid.



Which candidates satisfy the predicate is schematic; how many of them do is measured — 4.89 of 10, rounded to 5 survivors in the drawing. The index ranks on cosine alone, so the predicate can only be applied to what it already returned: every discarded candidate is a slot the answer never fills. Overfetching buys back 2.12 slots and 13.6% of recall for 15.5% of the corpus scanned per query. Mean latency does not order with overfetch across the 8 settings (7.3 ms–7.7 ms), so at this corpus size the price is paid in recall rather than in time. Pushing the predicate below the index instead of above it is the only version of this that terminates.

Figure 8. Post-filtering against a selective predicate. The slots are allocated by similarity before the constraint is consulted, so the constraint can only empty them.

Table 7. Retrieving k × overfetch rows by similarity and discarding those that fail the predicate, swept across 8 overfetch factors on the 8,282-memory corpus at k = 10. Slot fill rate is the mean number of surviving rows divided by k; scanned is the fraction of the corpus the fetch had to touch. Latency is the mean over all queries in the sweep.

Overfetch	Rows fetched	Slot fill rate	Recall	Mean latency	Corpus scanned
1×	10	48.9%	8.9%	7.5 ms	0.12%
2×	20	55.9%	12.1%	7.7 ms	0.24%
4×	40	59.2%	13.5%	7.7 ms	0.48%
8×	80	60.1%	14.1%	7.3 ms	0.97%
16×	160	60.9%	14.8%	7.5 ms	1.93%
32×	320	62.6%	16.8%	7.4 ms	3.86%
64×	640	64.6%	18.1%	7.6 ms	7.73%
128×	1,280	70.0%	22.5%	7.6 ms	15.46%

Work grows exactly with overfetch — 10 rows to 1,280, 128× — while recall moves from 8.9% to 22.5%. At the last point the fetch already touches 15.5% of the corpus and 29.9% of the k slots are still empty: overfetching buys a

diminishing amount of the answer, because the filter can only remove rows the index already ranked, never recover one it did not.

6.5 What the planner actually did

The claim that a DBMS “uses an index” is checkable. Both engines report their plans, and the harness captures one per query class per engine. Two are worth reading side by side.

PG GIN · provenance — EXPLAIN ANALYZE

```
Limit (actual 0.244 ms, rows 4, buffers 15h/0r)
  Sort (actual 0.241 ms, rows 4, buffers 15h/0r)
    Bitmap Heap Scan on memory (actual 0.232 ms, rows 4, buffers 15h/0r)
      BitmapAnd (actual 0.177 ms, rows 0, buffers 11h/0r)
        Bitmap Index Scan using ix_memory_session (actual 0.011 ms, rows 33, buffers 2h/0r)
        Bitmap Index Scan using ix_memory_ts (actual 0.162 ms, rows 1601, buffers 9h/0r)
```

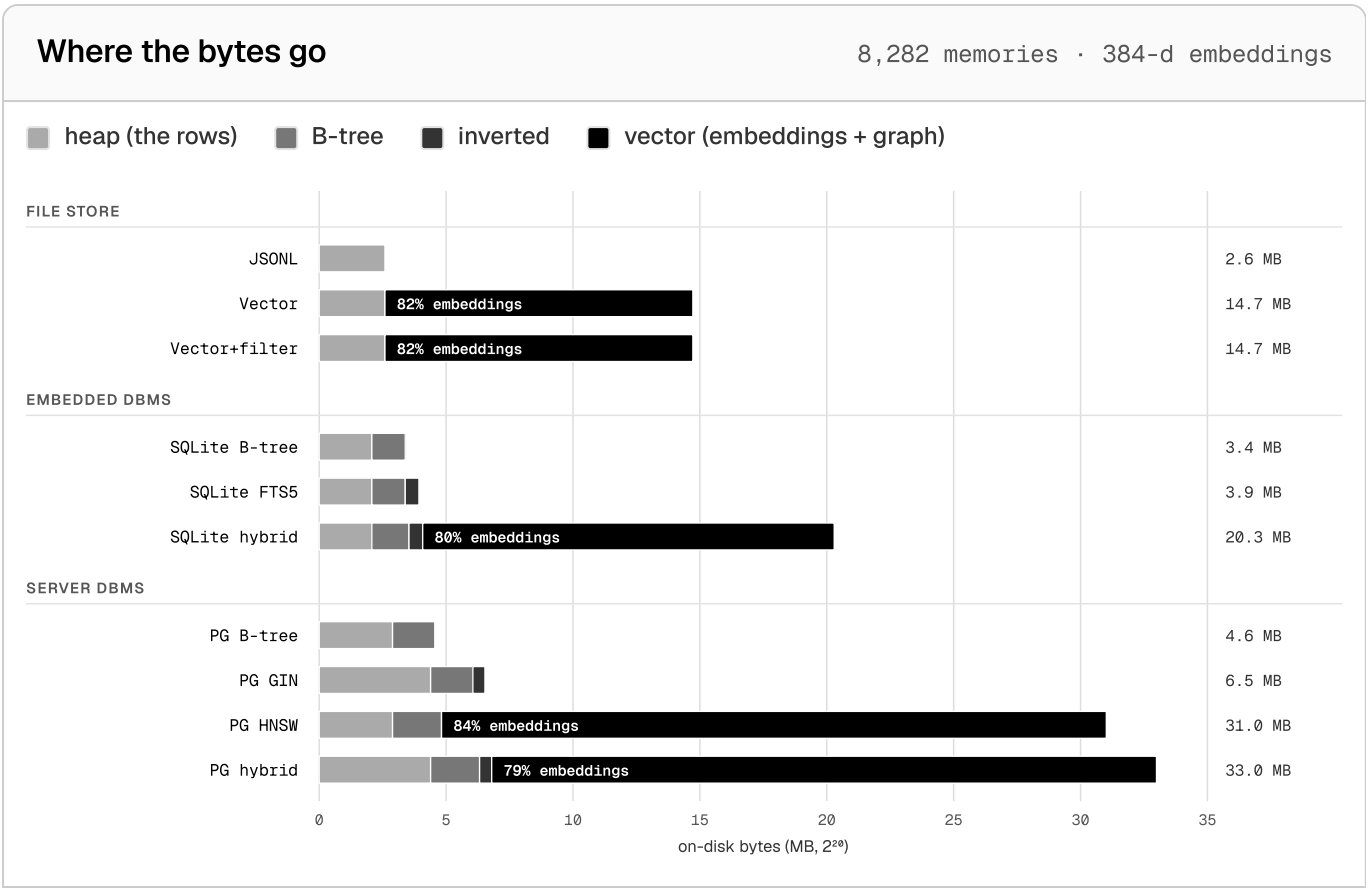
SQLite FTS5 · provenance — EXPLAIN QUERY PLAN

```
SCAN memory_fts VIRTUAL TABLE INDEX 0:M1
SEARCH m USING INTEGER PRIMARY KEY (rowid=?)
USE TEMP B-TREE FOR ORDER BY
```

Figure 9. The same question, two engines. Both resolve the session predicate through a B-tree rather than scanning, which is why both answer a class the similarity arms score near zero on.

6.6 Storage economics

The corpus is 784.4 KB of text. What the architectures cost to store it varies by more than an order of magnitude, and the reason is entirely the embeddings.



One zero-based axis across all 10 arms, so bar length is comparable between rows and not only within one. Sizes are what each engine reports on disk after load — byte-exact for the file arms, page-granular for SQLite and Postgres. Heap segments are not comparable across families, because a JSONL line is JSON text and a table row is typed columns; the index segments are what this figure is about. The result it exists for: on every arm that carries embeddings the vector segment is roughly four fifths of the store, so the cost of semantic recall is paid in storage before a single query runs.

Figure 10. Bytes by role. At 384 dimensions in float32, one vector is 1536 bytes — larger than the memory it describes, which averages 25.4 tokens.

Table 8. On-disk size of each store after loading 8,282 memories, with every relation and index attributed to one of four kinds and summed. Heap is row storage, B-tree covers primary keys and secondary indexes, inverted is the full-text index, vector is the embedding column plus any nearest-neighbour index over it. The four kinds sum exactly to the total. Vector share is that column as a fraction of the total.

Engine	Heap	B-tree	Inverted	Vector	Total	Vector share	Bytes / memory
JSONL file `file-jsonl`	2.6 MB	—	—	—	2.6 MB	—	328
Flat vector index `file-vec`	2.6 MB	—	—	12.1 MB	14.7 MB	82.4%	1,864
Vector index + post-filter `file-vec-meta`	2.6 MB	—	—	12.1 MB	14.7 MB	82.4%	1,864
SQLite · B-tree only `sqlite-btree`	2.1 MB	1.3 MB	—	—	3.4 MB	—	429
SQLite · FTS5 inverted index `sqlite-fts`	2.1 MB	1.3 MB	552.0 KB	—	3.9 MB	—	498
SQLite · FTS5 + vectors `sqlite-hybrid`	2.1 MB	1.5 MB	552.0 KB	16.2 MB	20.3 MB	79.9%	2,569

Engine	Heap	B-tree	Inverted	Vector	Total	Vector share	Bytes / memory
Postgres · B-tree only `pg-btree`	2.9 MB	1.7 MB	—	—	4.6 MB	—	577
Postgres · GIN inverted index `pg-gin`	4.4 MB	1.7 MB	488.0 KB	—	6.5 MB	—	827
Postgres · pgvector HNSW `pg-hnsw`	2.9 MB	1.9 MB	—	26.2 MB	31.0 MB	84.5%	3,927
Postgres · GIN + HNSW hybrid `pg-hybrid`	4.4 MB	1.9 MB	488.0 KB	26.2 MB	33.0 MB	79.4%	4,177

The total is the sum of the measured relation and index sizes. The Postgres arms also report a whole-cluster figure, which includes the catalogue, the write-ahead log and free space and is therefore not comparable across families: `pg-btree` 42.9 MB, `pg-gin` 60.9 MB, `pg-hnsw` 101.6 MB, `pg-hybrid` 103.6 MB.

SQLite · FTS5 inverted index holds the corpus in 497.5 bytes per memory; Postgres · GIN + HNSW hybrid needs 4177.1, a factor of 8.4. For an agent whose memory grows monotonically, that ratio is the difference between a store that fits on the machine and one that does not — and §6.2 shows the vectors buying quality on one class out of 8.

6.7 Durability under crash

Retrieval quality is the interesting half of the problem; not losing the memory is the necessary half. We killed each writer with an uncatchable signal partway through sustained writes, then reopened the store and asked what survived. The clock starts only after the store holds a durable baseline, so this measures a crash during writing rather than during initialisation.

Table 9. 68 kill trials across 4 store designs. Each trial seeds the store, writes until it reaches steady state, kills the process without warning, then reopens and reads. Caught mid-write counts trials where the kill landed during a write; reopened counts trials where the store opened again at all; unreadable counts trials where it did not. Records durable is the mean number of rows still present after reopening.

Store	Trials	Caught mid-write	Reopened	Unreadable	Records durable	Outcome
`file-append`	20	20	20	`0`	154,030	append-only survived: every line still parses
`file-rewrite`	20	20	17	`3`	17,043	the kill happened between rewrites, so this document parsed
`sqlite`	20	20	20	`0`	70,223	recovered to a committed boundary; integrity_check = ok
`postgres`	8	8	8	`0`	10,719	WAL replayed on open; only committed transactions present

Records durable is not comparable across rows: each design was seeded to its own steady state, so the count reflects write throughput before the kill rather than how much was preserved. What is comparable is the unreadable column, and the outcome text beside it. Where the engine offers its own consistency check the result is recorded: `sqlite` → ok.

The honest result is that **append-only files are fine**. Across 20 kills the JSONL store never lost a parseable line: appending is close to atomic at these sizes, and the failure mode is a missing tail rather than a corrupt file.

The common file pattern is not fine. Holding memory as one JSON document and rewriting it on every change means the file is briefly neither the old state nor the new one — and in 3 of 20 trials the crash landed inside that window and left a document that no longer parses. The loss is not the last record. It is all 17,043 of them, because the store is a single value.

Both DBMS arms recovered to a committed boundary in every trial, with SQLite's `integrity_check` returning `ok` and Postgres replaying its write-ahead log[4] on open. This is not a surprising result; it is a fifty-year-old result[5]. It is included because the architecture that gets it wrong is the one currently in widest use.

6.8 Concurrency

Agents increasingly run as fleets sharing one memory. We ran 8 concurrent writers performing 25 increments each on the same record, in-process for every family so the result isolates the concurrency-control mechanism rather than the deployment model.

Table 10. 8 concurrent writers, 25 increments each, against a single counter in each of 3 stores. Expected is writers × rounds; observed is the value read back after every writer finished; lost is the difference. The mechanism column names what did — or did not — order the two writers.

Store	Expected	Observed	Lost	Lost share	Mechanism
`file`	200	25	175	87.5%	read-modify-write of the whole document; last write wins
`sqlite`	200	200	0	0.0%	UPDATE … SET hits = hits + 1, serialised by the engine
`postgres`	200	200	0	0.0%	UPDATE under MVCC; the row lock orders the two writers

The file store lost 175 of 200 updates — 87.5%. It is the textbook lost-update anomaly[3], and it arrives here for the textbook reason: read the document, modify a field, write the document back, and whoever writes last erases everyone else. Both DBMS arms lost nothing, expressing the same edit as a single statement whose atomicity the engine guarantees.

6.9 The normalisation anomaly, as contradiction

This is where the database-theory chapter stops being theoretical. In our corpus a fact is restated across 3.69 memories on average and up to 15. When the fact changes, a normalised schema updates one row in `FACT`. A denormalised store must find and rewrite every restatement.

An agent does not rewrite every restatement. It rewrites what it retrieved — at most 10 rows. Across 120 probed facts a top-10 repair reached 1.53 of 4.2 restatements, leaving **63.5% still asserting the old value**.

Those rows do not sit inertly on disk. They match the same queries they always did, so the agent retrieves them, reads them as confident statements of fact, and acts on them. An update anomaly in an agent's memory does not present as a data-quality metric. It presents as an agent that contradicts itself.

Table 11. The cost of a corrected fact, over the 2,500 facts and 8,282 memories of the corpus. A restatement is a later memory that supersedes an earlier one about the same fact. The third block assumes a repair pass that rewrites every row a top-k retrieval returns and gets every one of them right; what it leaves stale is what retrieval never showed it.

Measure	Value
Restatements per fact — 1,695 facts restated at least once	
Mean	3.69
Median	3
90th percentile	8
Maximum	15
Rows a single correction must update	
Normalised — the fact is stored once	1
Denormalised — mean over restated facts	3.69
Denormalised — worst case observed	15
Repair through top-k retrieval — k = 10, 120 facts probed	
Restatements a fact has, mean	4.20
Restatements top-k reaches, mean	1.53

Measure	Value
Left stale after a perfect repair	63.5%

Table 11, continued. 12 sampled facts from the probe, showing how much of each fact's history a single top-10 retrieval reaches. Stale is restatements minus reached — rows a repair driven by that retrieval would never see.

Fact	Restatements	Reached by top-10	Left stale	Stale share
F-inc-INC-2356	4	2	2	50.0%
F-inc-INC-3268	3	3	0	—
F-adr-ADR-385	3	2	1	33.3%
F-run-RUN-481	2	2	0	—
F-run-RUN-455	4	2	2	50.0%
F-cfg-CFG-1566	2	1	1	50.0%
F-inc-INC-2809	2	2	0	—
F-inc-INC-4135	2	2	0	—
F-cfg-CFG-1637	2	1	1	50.0%
F-cfg-CFG-1222	3	1	2	66.7%
F-own-PER-231	8	4	4	50.0%
F-cfg-CFG-713	4	1	3	75.0%

A normalised store updates 1 row and the correction is complete. Copying the fact into every memory that mentions it turns the same correction into 3.69 rows on average and 15 at worst — and since top-10 retrieval reaches only 1.53 of the 4.20 rows a fact occupies, 63.5% of them stay wrong however good the repair is.

6.10 The context-window budget

Retrieval is not free at the point of use. Everything returned is pasted into a context window and paid for per token, so the right question is not “which arm scores highest” but “which arm scores highest per token spent”.

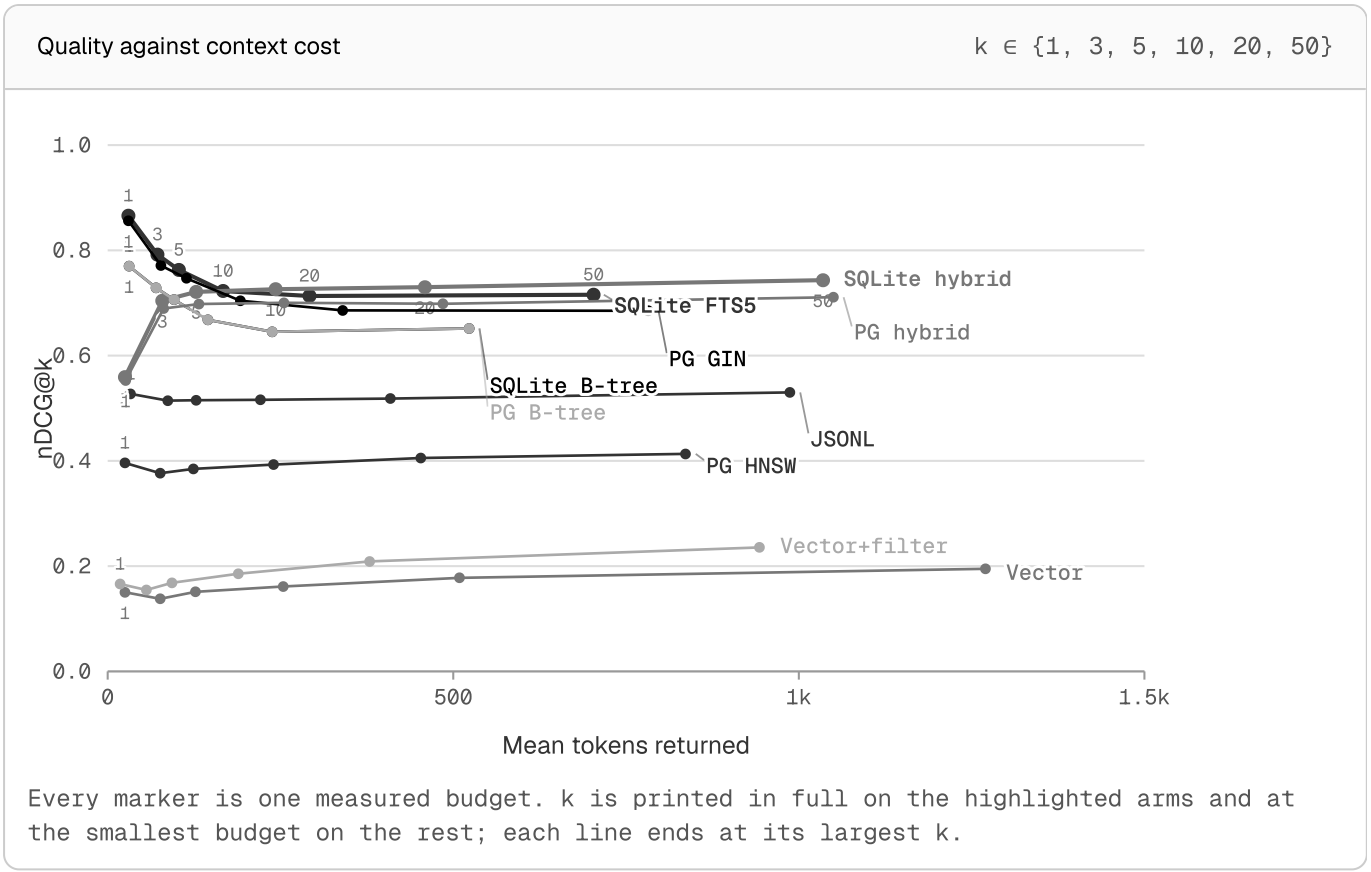


Figure 11. Each line is one architecture swept over k. Up and to the left is better: more quality for fewer tokens.

6.11 Scaling

Every architecture was rebuilt and re-measured at 609, 1,995, 8,282, 33,197 memories. The unindexed arms degrade linearly, as they must; the indexed arms do not.

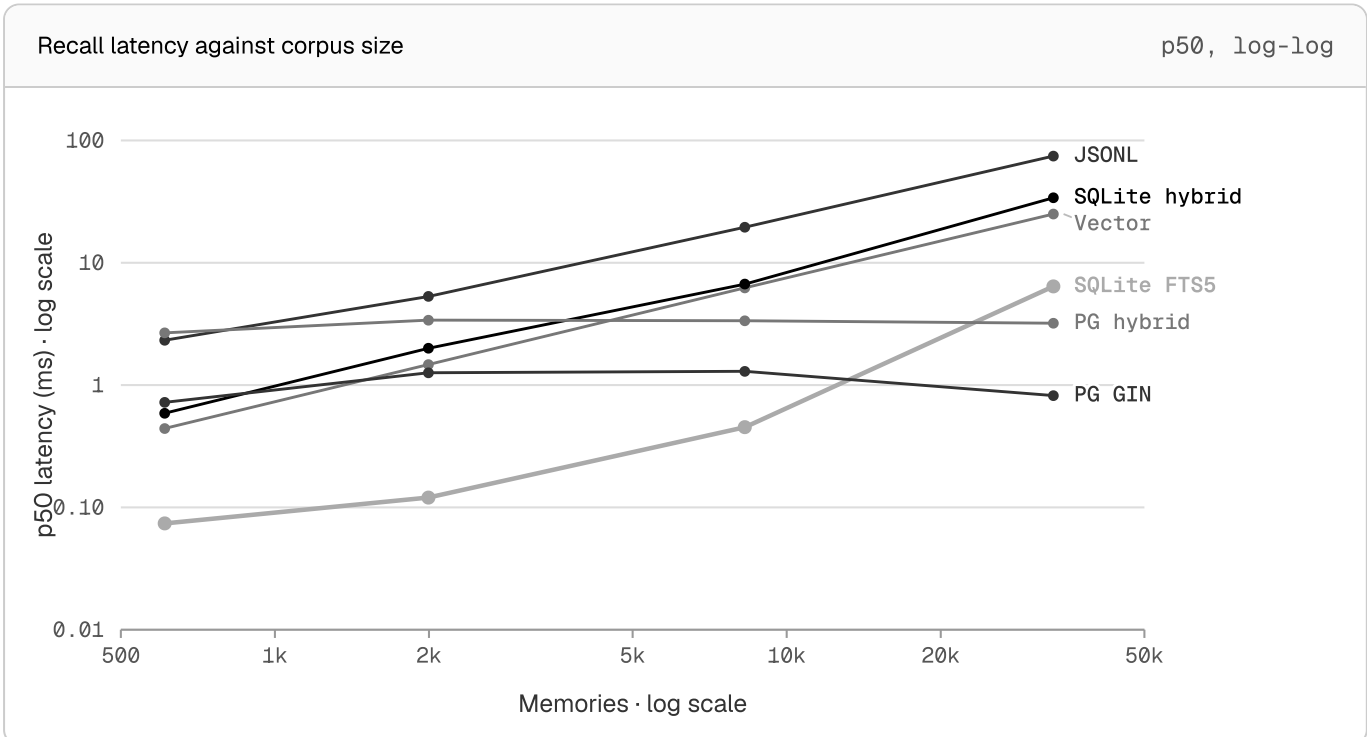


Figure 12. A scan is linear in the corpus. An index is not.

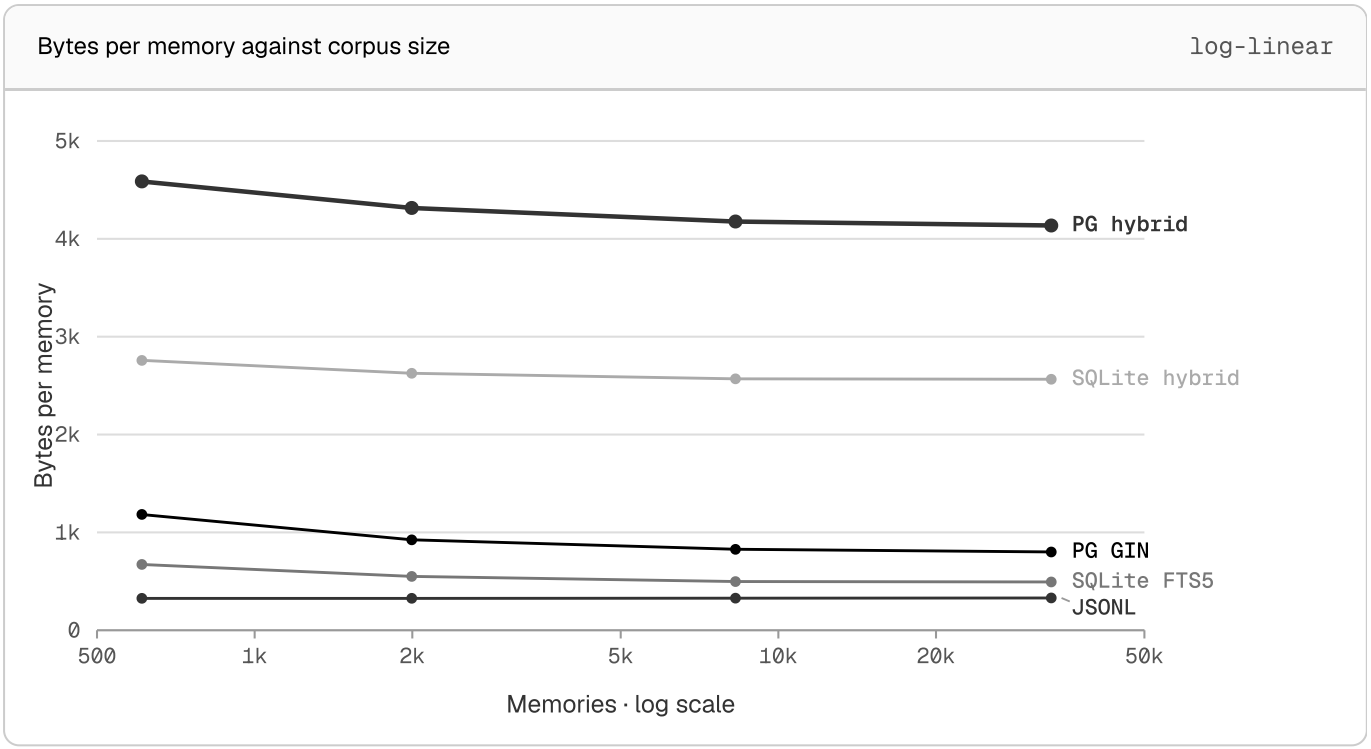


Figure 13. Per-memory storage is roughly constant; the gap between arms is the embedding, not the data.

Table 12. 5 architectures re-measured at 4 corpus sizes, each a fresh load and a fresh query set drawn from the same generator, chosen so that each column is a different design rather than a different tuning of the same one. Rows are scales, labelled by the memories actually produced; columns are engines. The upper block is median query latency, the lower block is store size per memory — the two costs that grow for different reasons.

Scale	JSONL	SQLite FTS5	SQLite hybrid	PG GIN	PG hybrid
Median query latency, p50 — 10 results per query					
609 memories target 500 · 94 queries	2.3 ms	0.07 ms	0.59 ms	0.72 ms	2.7 ms
1,995 memories target 2,000 · 91 queries	5.3 ms	0.12 ms	2.0 ms	1.3 ms	3.4 ms
8,282 memories target 8,000 · 313 queries	19.5 ms	0.45 ms	6.7 ms	1.3 ms	3.4 ms
33,197 memories target 32,000 · 92 queries	74.5 ms	6.4 ms	34.0 ms	0.82 ms	3.2 ms
Store size, bytes per memory					
609 memories target 500 · 94 queries	326	673	2,758	1,184	4,587
1,995 memories target 2,000 · 91 queries	327	550	2,626	924	4,316
8,282 memories target 8,000 · 313 queries	328	498	2,569	827	4,177
33,197 memories target 32,000 · 92 queries	330	494	2,565	800	4,136

Every arm completed every scale; no cell is marked failed. Latency and size are therefore comparable down each column as well as across each row.

The ingest side carries the opposite lesson. Building an inverted index is cheap; building an HNSW graph is not, and neither is writing 384-dimensional vectors through a query protocol. That cost is paid once per memory rather than once per recall, which is the right trade for a store written far less often than it is read — but it is not free, and for a memory that is written on every turn it is the dominant term.

7. Retrieval as a query, not as code

One result deserves separating from the measurements, because it is about what the architecture makes *expressible* rather than what it makes fast.

Hybrid retrieval is normally application code: run the keyword search, run the vector search, merge the two ranked lists, apply the filters, return the top *k*. In the Postgres arm none of that code exists. The entire strategy — both indexes, the structural predicate, reciprocal rank fusion[8] and the truncation — is one statement the planner optimises as a unit.

```
Postgres · GIN + HNSW hybrid — the whole retrieval strategy        generated by engines/postgres.mjs
```

```
SELECT m.memory_id FROM memory m WHERE m.ts @@ to_tsquery('english', $1) AND m.created_day <= $2
         ORDER BY ts_rank_cd(m.ts, to_tsquery('english', $3)) DESC LIMIT 10
```

This matters for a reason beyond elegance. Retrieval strategy is the part of an agent that changes most often — a new filter, a different weighting, a recency term. Expressed as a query it is data the engine re-plans against current statistics. Expressed as application code it is a merge loop that has to be rewritten, re-tested, and kept consistent with whatever the store is doing.

Section 8 is interactive

The query-class explorer, the dense-retrieval forensics and the schema browser are things you operate rather than read. Scan the code, or visit **dbms-memory.khe.money**.

9. Discussion

9.1 A vector index is not a memory system

The strongest reading of our results is not that vector search is bad. On paraphrase it is the best tool available, and it is the only arm that finds a memory sharing no words with the query. The error is one of scope: a vector index is *one access method*, and agent memory needs several.

61.7% of our workload is decided by a predicate, a join or an aggregate. Those are not exotic queries — they are “what did we decide before the migration”, “what came out of that session”, “what is still true”, “how many times did this come up”. A system whose only operation is top-*k* similarity cannot express them, and the failure is silent: it returns ten plausible memories and no indication that the question was not the one it answered.

9.2 What a DBMS actually contributes

Our results separate into two kinds, and they are worth keeping apart.

The **retrieval** results are contingent. They depend on our corpus, our embedding model and our query mix; a different domain would move them. A reader is entitled to discount them.

The **integrity** results are not contingent in the same way. A store with no transaction will lose concurrent updates; a store that rewrites a single document has no commit boundary to recover to; a store that repeats a fact in 3.69 places cannot correct it atomically. These follow from the architecture, not from the workload. They are also the results that current practice most consistently ignores.

9.3 The recommendation is smaller than “use Postgres”

SQLite · FTS5 + vectors scored highest overall at 0.726. We would not recommend it as a default. SQLite · FTS5 inverted index reached 0.723 — 0.5% lower — at 0.44 ms median latency, 8.4× less storage, no embedding model and no server. For a single agent on one machine that is the better engineering.

The case for the server arm is the case for concurrency, for a real planner, and for expressing retrieval as a query rather than as code. The case for the vectors is narrower still: they earn their 8.4× storage on paraphrase recall and, on our corpus, nowhere else.

The general lesson is the one the normalisation ladder in §4.2 already states. Agent memory has a schema whether or not anyone writes it down. Writing it down is what makes the questions answerable.

---

10. Threats to validity

The corpus is synthetic. This is the most important caveat. Real agent transcripts are private, and non-circular relevance labels require knowing which fact each memory restates — which means generating the memories from the facts. We accept the trade and mitigate it by validating the corpus before using it (§5.3, Table 5) rather than assuming its properties. Absolute scores should not be read as predictions for any deployed system; the comparison between arms on identical data is what we claim.

The distractors are adversarial by construction. 15% of the corpus mentions an identifier while asserting nothing about it, and those records share the grammatical shape of a lookup question. That design is why dense retrieval scores as low as it does on the lexical class. We think the pattern is realistic — agents write “checked X, unrelated” constantly — but a corpus without it would narrow the gap, and the 0.002 figure should be read as the behaviour under adversarial-but-plausible distractors rather than a universal constant.

One embedding model, one dimensionality. All similarity results use Xenova/all-MiniLM-L6-v2 at 384 dimensions. A larger model, or one trained with identifier-aware objectives, would score better on the lexical class. The structural classes would not move at all, because their failure is representational rather than a matter of embedding quality.

PGlite is Postgres in WebAssembly. The SQL semantics, planner, MVCC and WAL are genuine PostgreSQL 18.3, which is what our correctness claims rest on. The absolute latencies are not those of a native server: WASM is slower and single-threaded, so the Postgres arms are penalised on timing relative to a real deployment. Where we compare engines on latency we say so; the structural results do not depend on it.

Concurrency was measured in-process. Running the file store across processes and PGlite in one would have confounded concurrency control with the deployment model. The lost-update result therefore isolates the mechanism, and says nothing about throughput under real multi-process contention.

Recency happens to substitute for currency. The file arm scores well on the currency class, but not because it models supersession — it breaks ties by recency, and in our corpus the correction is always the newest memory. That coincidence is a property of the generator. It would fail the moment a superseded memory were touched again, and the arm has no way to express the constraint that would make it robust.

Single machine, single run. All measurements come from one 16-core machine on win32/x64. Latency medians are over three timed runs per query; they are not a substitute for a benchmarking harness with isolation and repetition across machines.

---

11. Related work

The database side of this paper is textbook and deliberately so. The relational model[1], the entity-relationship model[2], normalisation and the transaction[3] are settled results; our contribution is applying them to a workload that has grown up without them, and measuring what their absence costs. Recovery follows ARIES[5]; the ranking baseline is BM25[14]; fusion uses reciprocal rank fusion with the original constant rather than a tuned one[8], since a tuned constant would let the hybrid arm win by fitting our corpus.

On the agent side, retrieval-augmented generation[15] established the pattern of fetching text into a context window, and the systems that followed — MemGPT[16] with its paged memory hierarchy, and the generative-agent architecture[13] with its retrieval scored on recency, importance and relevance — both treat memory as a storage problem. Neither is evaluated as a database: we are not aware of prior work measuring agent memory for durability under crash, for lost updates under concurrent writers, or for the update anomaly that follows from storing an unnormalised fact.

Approximate nearest-neighbour search over HNSW[9] and its integration into a relational engine through pgvector[10] are what make the hybrid arm possible at all; the observation that filtered vector search interacts badly with post-filtering is

known in that literature, and §6.4 quantifies it for this workload.

12. Conclusion

An agent's memory is a database, and building it without one costs more than performance. Across 10 architectures on 8,282 memories and 313 labelled queries we find that 61.7% of a realistic recall workload cannot be expressed as similarity search at all; that dense retrieval is near-random on the identifier queries agents ask most (0.002 against 0.790 nDCG for an inverted index); that eight concurrent writers cost a file store 87.5% of its updates while costing both DBMS arms nothing; and that a fact restated 3.69 times cannot be corrected by retrieval, leaving 63.5% of its restatements contradicting the agent's current belief.

None of the database results are new. Codd[1], Chen[2] and Gray[3] settled them decades ago. What is new is the setting: a class of system that writes records, indexes them, queries them under a budget and must survive a crash — and that has largely been built as though none of that work had happened. The useful contribution of this paper is not a new architecture. It is a measurement of how much the old one is still worth.

References

E. F. Codd. *A Relational Model of Data for Large Shared Data Banks.* Communications of the ACM 13(6), 1970. dl.acm.org

P. P.-S. Chen. *The Entity-Relationship Model — Toward a Unified View of Data.* ACM TODS 1(1), 1976. dl.acm.org

J. Gray, A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993. dl.acm.org

The PostgreSQL Global Development Group. *Write-Ahead Logging (WAL).* PostgreSQL 18 documentation. postgresql.org

C. Mohan et al. *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging.* ACM TODS 17(1), 1992. dl.acm.org

The PostgreSQL Global Development Group. *PostgreSQL 18 documentation.* postgresql.org/docs/18. Measured build: `PostgreSQL 18.3 (PGlite 0.5.4) on wasm32-unknown-linux-gnu, compiled by emcc (Emscripten gcc/clang-like replacement + linker emulating GNU ld) 3.1.74 (1092ec30a3fb1d46b1782ff1b4db5094d3d06ae5), 32-bit`

SQLite Consortium. *SQLite 3.53.0*, and FTS5 full-text search. sqlite.org

G. V. Cormack, C. L. A. Clarke, S. Buettcher. *Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods.* SIGIR 2009. dl.acm.org

Y. A. Malkov, D. A. Yashunin. *Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs.* 2016. arXiv:1603.09320

A. Kane et al. *pgvector — open-source vector similarity search for Postgres.* github.com/pgvector/pgvector. Build 0.0.5.

ElectricSQL. *PGlite — PostgreSQL packaged as WebAssembly.* pglite.dev. Version 0.5.4.

E. Tulving. *Episodic and Semantic Memory.* In *Organization of Memory*, Academic Press, 1972. alicekim.ca

J. S. Park et al. *Generative Agents: Interactive Simulacra of Human Behavior.* UIST 2023. arXiv:2304.03442

S. Robertson, H. Zaragoza. *The Probabilistic Relevance Framework: BM25 and Beyond.* Foundations and Trends in Information Retrieval, 2009. city.ac.uk

P. Lewis et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.* NeurIPS 2020. arXiv:2005.11401

C. Packer et al. *MemGPT: Towards LLMs as Operating Systems.* 2023. arXiv:2310.08560

R. Elmasri, S. B. Navathe. *Fundamentals of Database Systems.* 7th ed., Pearson, 2016. pearson.com · R. Ramakrishnan, J. Gehrke. *Database Management Systems.* 3rd ed., McGraw-Hill, 2003. cs.wisc.edu

---

Appendix A. Reproduction

Everything regenerates from one command. No database server, no API key and no network access at measurement time — the embedding model and both engines are local.

```
regenerate everything
```

```
git clone https://github.com/HKTITAN/dbms-agent-memory
cd dbms-agent-memory && npm install

npm run paper   # corpus -> embed -> capture -> qr -> build -> pdf -> epub
```

`npm run capture` alone rebuilds the dataset every figure reads. It runs the 10 architectures over the main corpus, the post-filter sweep, the scaling sweep at 4 sizes, the crash trials, the lost-update test and the anomaly probe, and writes `data/capture.json`. Total runtime is dominated by the Postgres arms.

Appendix B. Schema DDL

Generated from `engines/schema.mjs` — the same declaration the ER diagram in Figure 2 renders from, so the diagram and the executed schema cannot disagree.

PostgreSQL 7 relations, 5 secondary indexes

```
CREATE TABLE agent (
  agent_id TEXT NOT NULL,
  model TEXT NOT NULL,
  family TEXT NOT NULL,
  PRIMARY KEY (agent_id)
);

CREATE TABLE session (
  session_id TEXT NOT NULL,
  agent_id TEXT NOT NULL,
  started_day INTEGER NOT NULL,
  PRIMARY KEY (session_id),
  FOREIGN KEY (agent_id) REFERENCES agent(agent_id)
);

CREATE TABLE entity (
  entity_id TEXT NOT NULL,
  entity_kind TEXT NOT NULL,
  display_name TEXT NOT NULL,
  PRIMARY KEY (entity_id)
);

CREATE TABLE fact (
  fact_id TEXT NOT NULL,
  subject_id TEXT NOT NULL,
  predicate TEXT NOT NULL,
  object_value TEXT NOT NULL,
  memory_kind TEXT NOT NULL,
  valid_from_day INTEGER NOT NULL,
  PRIMARY KEY (fact_id),
  FOREIGN KEY (subject_id) REFERENCES entity(entity_id)
);

CREATE TABLE memory (
  memory_id TEXT NOT NULL,
  session_id TEXT NOT NULL,
  turn_no INTEGER NOT NULL,
  seq INTEGER NOT NULL,
  kind TEXT NOT NULL,
  body TEXT NOT NULL,
  fact_id TEXT,
  created_day INTEGER NOT NULL,
  source TEXT NOT NULL,
  confidence REAL NOT NULL,
  superseded_by TEXT,
  PRIMARY KEY (memory_id),
  FOREIGN KEY (session_id) REFERENCES session(session_id),
  FOREIGN KEY (fact_id) REFERENCES fact(fact_id),
  FOREIGN KEY (superseded_by) REFERENCES memory(memory_id)
);

CREATE TABLE memory_entity (
  memory_id TEXT NOT NULL,
  entity_id TEXT NOT NULL,
  PRIMARY KEY (memory_id, entity_id),
  FOREIGN KEY (memory_id) REFERENCES memory(memory_id),
  FOREIGN KEY (entity_id) REFERENCES entity(entity_id)
);

CREATE TABLE embedding (
  memory_id TEXT NOT NULL,
  model TEXT NOT NULL,
  dim INTEGER NOT NULL,
  vec VECTOR(384) NOT NULL,
  PRIMARY KEY (memory_id),
  FOREIGN KEY (memory_id) REFERENCES memory(memory_id)
);

CREATE INDEX ix_memory_session ON memory(session_id);  -- serves: provenance
CREATE INDEX ix_memory_day ON memory(created_day);  -- serves: temporal
CREATE INDEX ix_memory_fact ON memory(fact_id);  -- serves: aggregate
CREATE INDEX ix_memory_superseded ON memory(superseded_by);  -- serves: currency, negation
CREATE INDEX ix_mement_entity ON memory_entity(entity_id);  -- serves: entity lookup
```

Harshit Khemani, Kush Ahuja, Madhav Bassi, Kushagra Agrawal · dbms-memory.khe.money

PostgreSQL 18.3 · SQLite 3.53.0 · 2026-08-09