AgenticMemory II: Nine Cognitive Query Types for Graph-Structured Agent Memory

Omoshola Owolabi
Researcher – AI/ML
Agentra Labs
omoshola.owolabi@agentralabs.tech

February 22, 2026

Abstract

In a companion paper [**?**], we introduced AgenticMemory, a binary graph format for persistent AI agent memory with typed cognitive events and semantic edges. That work supported five query types focused on local navigation. We now present AgenticMemory v0.2.0, extending the query model from 5 to 16 types in four categories: *retrieval* (BM25 text search, hybrid BM25+vector fusion), *structural analysis* (PageRank, degree and betweenness centrality, bidirectional BFS, Dijkstra shortest path), *cognitive reasoning* (counterfactual belief revision, gap detection, analogical pattern matching), and *graph maintenance* (consolidation, drift detection). All nine new algorithms are implemented in 3,586 lines of Rust with zero new dependencies. Two new index structures accelerate text queries while maintaining backward compatibility via automatic slow-path fallback and feature flags. The implementation passes 179 Rust and 104 Python tests. Criterion benchmarks on 100,000-node graphs show BM25 search at 1.58 ms (77× faster than the backward-compatible slow path), hybrid search at 10.83 ms, PageRank at 34.3 ms, and bidirectional BFS at 104 μs. Computationally intensive queries—gap detection (297 s), analogical reasoning (229 s)—scale super-linearly and target periodic batch execution. These additions transform AgenticMemory from a navigable memory store into a cognitive reasoning engine.

1 Introduction

AgenticMemory v0.1.0 [**?**] established that AI agent memory is fundamentally a graph navigation problem rather than a text search problem. The binary format—six event types, seven edge types, fixed-size records with memory-mapped $O(1)$ access—demonstrated that a single-file, zero-dependency graph could outperform vector databases, key-value stores, and markdown-based systems across storage efficiency, query latency, and relationship depth.

However, the initial query model was deliberately constrained. The five v0.1 query types—traversal, pattern, temporal, similarity, and resolve—provide local neighborhood exploration and point lookups. They answer questions like "what caused this decision?" or "what is the current version of this fact?" but cannot answer global structural questions: "which beliefs are most central to my reasoning?", "what decisions lack sufficient evidence?", or "has my understanding of this topic drifted over time?"

This paper presents AgenticMemory v0.2.0, which extends the query model with nine new query types that enable four categories of cognitive capability:

1. **Retrieval** — BM25 text search and hybrid BM25+vector fusion via Reciprocal Rank Fusion (RRF) provide keyword-based recall alongside existing embedding-based similarity search.
2. **Structural analysis** — PageRank, degree centrality, betweenness centrality, and shortest-path algorithms (bidirectional BFS for unweighted, Dijkstra for weighted graphs) reveal the topological structure of the agent's knowledge.
3. **Cognitive reasoning** — Counterfactual belief revision, reasoning gap detection, and analogical pattern matching enable the agent to audit its own reasoning processes.
4. **Graph maintenance** — Consolidation (deduplication, contradiction linking, inference promotion) and drift detection enable autonomous knowledge hygiene.

A key engineering constraint is *zero new dependencies*: all algorithms are implemented using only Rust's standard library collections (`HashMap`, `BinaryHeap`, `VecDeque`). This maintains the original design goal of a self-contained binary with no external service requirements.

Backward compatibility is achieved through two mechanisms: (1) new index structures use tagged blocks that v0.1 readers skip gracefully, and (2) a feature-flags bitfield in a previously reserved header field allows readers to detect and adapt to format extensions without breaking the fixed-size header layout.

The remainder of this paper is organized as follows. Section 2 positions this work relative to graph query models and cognitive architectures. Section 3 describes the format extensions for backward compatibility. Section 4 presents the nine new query types with algorithmic details.

Section 5 reports benchmarks on graphs up to 100,000 nodes. Section 6 describes the Python SDK extensions. Section 8 discusses implications and limitations. Section 9 concludes.

2 Related Work

The v0.1 paper [**?**] surveyed agent memory systems broadly. Here we focus on the specific capabilities that v0.2 adds: text search, graph analysis, and cognitive reasoning over knowledge graphs.

BM25 and hybrid search. BM25 [**?**] remains the dominant term-weighting function for information retrieval, outperforming TF-IDF on most benchmarks. Recent work on hybrid search [**?**] combines sparse (BM25) and dense (vector) retrieval using score fusion techniques. Reciprocal Rank Fusion (RRF) [**?**] is a simple, parameter-free method that merges ranked lists by summing $1/(k+r_i)$ across rankings. We adopt RRF for its simplicity and competitive performance.

Graph centrality. PageRank [**?**] measures global node importance through iterative random walk simulation. Betweenness centrality [**?**] quantifies how often a node appears on shortest paths between other nodes. Both have been applied to knowledge graphs for identifying key entities [**?**], but not previously to cognitive event graphs where nodes represent agent beliefs rather than real-world entities.

Belief revision. The AGM framework [**?**] formalizes belief revision as contraction and expansion operations on belief sets. Our counterfactual analysis implements a read-only variant: rather than modifying the belief set, we trace the cascade of downstream effects that would result from retracting a given node, enabling "what if?" reasoning without mutation.

Analogical reasoning. Structure-mapping theory [**?**] proposes that analogies are based on shared relational structure rather than surface similarity. Our analogical query operationalizes this by computing structural fingerprints—in-degree, out-degree, edge type distribution—and combining structural similarity with content similarity to find past reasoning patterns that match a given query pattern.

Knowledge graph maintenance. Entity resolution and deduplication in knowledge graphs is well-studied [**?**], but typically requires external embedding models or string similarity services. Our consolidation algorithm operates entirely within the graph, using edge-weighted similarity and content overlap to identify duplicate and contradictory nodes.

3 Format Extensions

The v0.2 format maintains full backward compatibility with v0.1 while adding two new index structures and a feature-flags mechanism.

3.1 New Index Structures

The v0.1 index block used four tagged entries (type index, temporal index, session index, cluster map). The v0.2 format adds two new tags:

TermIndex (tag 0x05). An inverted index mapping terms to posting lists. Each posting list entry stores the node ID, term frequency, and field (content or metadata). The index is built during file write by tokenizing node content with a simple whitespace-and-punctuation tokenizer. This enables the BM25 fast path: queries against the inverted index complete in $O(q \cdot |\text{postings}|)$ time rather than the $O(n)$ full-scan slow path.

DocLengths (tag 0x06). A dense array of `u32` token counts, one per node, indexed by node ID. This is required for BM25 length normalization: norm $= 1 - b + b \cdot (dl/\text{avgdl})$, where dl is the document length and avgdl is the average across all documents. Storing lengths in a contiguous array avoids recomputing them at query time.

3.2 Feature Flags

The v0.1 header reserved 7 bytes at the end of the 64-byte header record. The v0.2 format repurposes 4 of these bytes as a `u32` feature-flags bitfield:

- Bit 0: `HAS_TERM_INDEX` — TermIndex block is present.
- Bit 1: `HAS_DOC_LENGTHS` — DocLengths block is present.
- Bits 2–31: Reserved for future extensions.

The remaining 3 reserved bytes are untouched. This design enables forward compatibility: a v0.1 reader encountering a v0.2 file will read the reserved bytes as zero (which they were in v0.1) and skip unknown index tags during index block parsing. A v0.2 reader opening a v0.1 file detects the absence of feature flags and falls back to slow-path algorithms.

3.3 Backward Compatibility

Compatibility is bidirectional:

v0.1 files read by v0.2 code. When the TermIndex is absent, BM25 and hybrid queries fall back to a slow path that tokenizes node content on-the-fly and computes BM25 scores in a single $O(n)$ pass. This is correct but slower than the indexed fast path. All other new queries (centrality, shortest path, cognitive reasoning, maintenance) operate on the node and edge tables, which are unchanged from v0.1.

v0.2 files read by v0.1 code. The v0.1 reader's index parser skips unknown tag bytes using the length field in each tag-length-value block. The node table, edge table, content block, and feature vectors are identical in layout. The only observable difference is that the reserved header bytes are no longer zero, which v0.1 readers ignore.

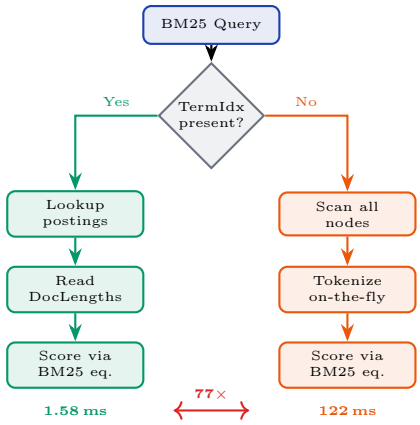

Figure 1: BM25 dual-path architecture. The fast path (green) uses the TermIndex and DocLengths array for sub-linear query time. The slow path (orange) performs a full scan for backward compatibility with v0.1 files. Measured speedup at 100K nodes: 77×.

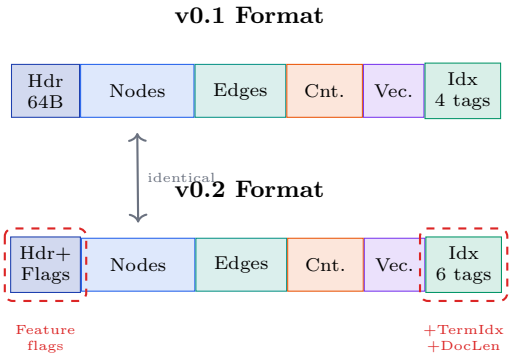

Figure 2: Binary format evolution from v0.1 to v0.2. Node table, edge table, content block, and feature vectors are unchanged. The header gains a feature-flags bitfield in previously reserved bytes, and the index block gains two new tagged entries (TermIndex, DocLengths). Red dashes highlight changes.

4 Query Expansion: Nine New Query Types

The nine new queries are organized into four categories reflecting the cognitive capability they provide. Figure 3 shows the complete taxonomy of all 16 query types across v0.1 and v0.2.

4.1 Retrieval: BM25 Text Search

BM25 [**?**] scores each document d against a query $Q = \{q_1, \ldots, q_n\}$ as:

$$\text{BM25}(d, Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, d) \cdot (k_1 + 1)}{f(q_i, d) + k_1 \cdot (1 - b + b \cdot \frac{|d|}{\text{avgdl}})} \quad (1)$$

where $f(q_i, d)$ is the term frequency, $|d|$ is the document length in tokens, avgdl is the average document length, and $\text{IDF}(q_i) = \ln\left(\frac{N - n(q_i) + 0.5}{n(q_i) + 0.5} + 1\right)$ with N total documents and $n(q_i)$ documents containing term q_i. We use $k_1 = 1.2$ and $b = 0.75$ following standard practice.

Fast path. When the TermIndex is present (Figure 1), the algorithm iterates over posting lists for each query term, accumulating BM25 scores in a hash map. Document lengths are read from the DocLengths array. Total cost: $O(q \cdot |\text{postings}|)$ where q is the number of query terms.

Slow path. When reading a v0.1 file without indexes (Figure 1), the algorithm performs a single $O(n)$ pass: tokenize each node's content, compute term frequencies on the fly, and accumulate BM25 scores. This is approximately 77× slower than the fast path on a 100K-node graph (122 ms vs. 1.58 ms) but produces identical results.

Both paths support optional filtering by event type and session ID.

4.2 Retrieval: Hybrid Search

Hybrid search combines BM25 (sparse, keyword-based) and cosine similarity (dense, embedding-based) results using Reciprocal Rank Fusion [**?**]:

$$\text{RRF}(d) = \sum_{r \in \mathcal{R}} \frac{1}{k + \text{rank}_r(d)} \quad (2)$$

where $\mathcal{R}$ is the set of ranking systems (BM25 and vector), $\text{rank}_r(d)$ is the rank of document d in ranking r, and $k = 60$ is a constant that dampens the influence of high rankings. RRF is parameter-free (beyond k) and does not require score normalization, making it robust across heterogeneous scoring functions.

The hybrid query first executes BM25 and similarity searches independently, then merges the results by RRF score. The output includes both the combined rank and the individual BM25 and vector scores for interpretability.

4.3 Structural Analysis: Centrality

Three centrality measures reveal the topological importance of nodes in the agent's knowledge graph:

PageRank [**?**] computes stationary probabilities of a random walk with damping factor $\alpha = 0.85$:

$$\text{PR}(v) = \frac{1 - \alpha}{N} + \alpha \sum_{u \in B(v)} \frac{\text{PR}(u)}{|L(u)|} \quad (3)$$

where $B(v)$ is the set of nodes linking to v and $L(u)$ is the out-degree of u. Our implementation handles dangling nodes (nodes with no outgoing edges) by redistributing their rank mass uniformly. Iteration continues until the L^1 norm of the rank vector change falls below 10^{-6} or 100 iterations are reached.

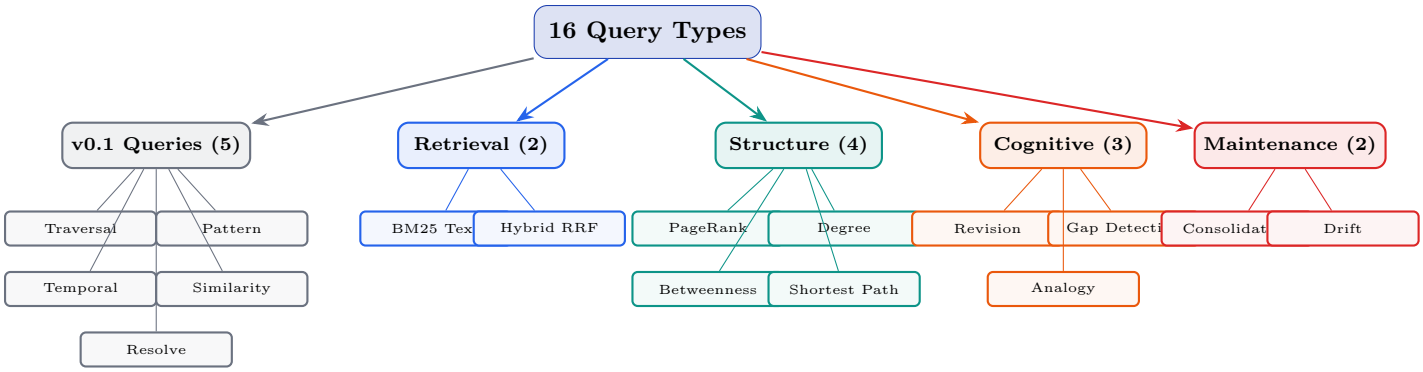

Figure 3: Complete query taxonomy for AgenticMemory v0.2. Gray boxes indicate the five v0.1 query types; colored boxes indicate the nine new v0.2 types organized into four cognitive categories.

Degree centrality computes the normalized degree $C_D(v) = \frac{\deg(v)}{N-1}$ for each node, providing a simple measure of local connectivity.

Betweenness centrality [**?**] measures how often a node appears on shortest paths between other pairs of nodes. We implement Brandes' algorithm, which computes betweenness in $O(VE)$ time by performing a BFS from each source node and accumulating pair-dependencies in a single backward pass. For graphs exceeding 1,000 nodes, we use random source sampling (200 sources) to approximate betweenness in practical time.

All centrality queries support optional filtering by event type, allowing the agent to ask questions like "which facts are most central to my reasoning?" or "which decisions are bridge nodes between topic clusters?"

4.4 Structural Analysis: Shortest Path

Two shortest-path algorithms serve different graph models:

Bidirectional BFS finds the shortest unweighted path between two nodes. The algorithm alternates BFS expansion from the source and target, terminating when the frontiers intersect. This reduces the search space from $O(b^d)$ to $O(b^{d/2})$ where b is the branching factor and d is the path length. A depth limit (default: 20) prevents exhaustive search in disconnected subgraphs.

Dijkstra's algorithm finds the shortest weighted path, using edge weights as distances. The implementation uses a binary heap (`BinaryHeap<Reverse<(Cost, NodeId)>>`) and returns both the path and its total weight.

Both algorithms return the full node sequence, enabling the agent to understand not just *that* two beliefs are connected but *how* they are connected through intermediate reasoning steps.

4.5 Cognitive Reasoning: Belief Revision

The belief revision query performs counterfactual analysis: given a target node v, it computes the downstream effects of retracting v from the belief graph. This is a *read-only* operation—no nodes or edges are modified.

Algorithm 1 Counterfactual Belief Revision

Require: Graph $G = (V, E)$, target node v
Ensure: Cascade set C, impact report R

$C \leftarrow \emptyset$; $Q \leftarrow \{v\}$
while $Q \neq \emptyset$ **do**
  $u \leftarrow Q$.dequeue()
  for each edge (u, w, t) where $t \in \{\textsc{CausedBy}, \textsc{Supports}\}$ **do**
    if $w \notin C$ **then**
      $C \leftarrow C \cup \{w\}$
      Compute impact: conf_loss(w), alt_support(w)
      if alt_support(w) $= 0$ **then**
        Q.enqueue(w) {No remaining support; propagate}
      end if
    end if
  end for
end while
$R \leftarrow$ summarize(C): affected nodes, confidence losses, unsupported inferences
return C, R

The algorithm (Algorithm 1, illustrated in Figure 4) performs a BFS along CausedBy and Supports edges. For each downstream node, it checks whether alternative support exists (other incoming Supports or CausedBy edges from nodes not in the cascade set). Nodes with no remaining support are added to the cascade and their downstream dependencies are explored recursively.

The report includes: (1) directly affected nodes, (2) confidence reductions for each affected node, (3) unsupported inferences that would become unjustified, and (4) a total impact score. This enables an agent to reason about the consequences of being wrong: "if this fact is incorrect, what conclusions would I need to reconsider?"

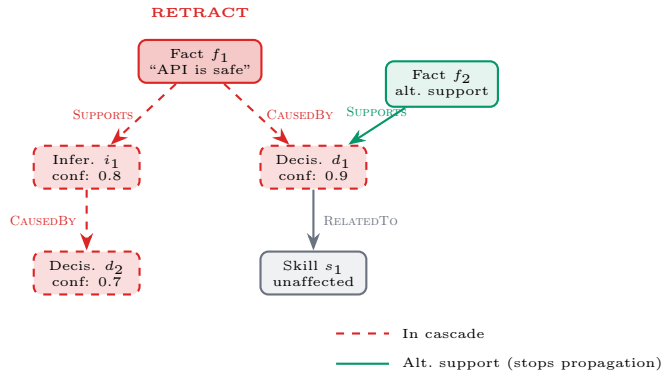

Figure 4: Counterfactual belief revision cascade. Retracting Fact f_1 propagates along SUPPORTS and CAUSEDBY edges (red dashed). Decision d_1 has alternative support from f_2 (green), halting cascade propagation. Inference i_1 has no alternative support, so the cascade continues to Decision d_2.

4.6 Cognitive Reasoning: Gap Detection

The gap detection query identifies five categories of reasoning weaknesses (Figure 5):

1. **Unjustified decisions** — Decision nodes with no incoming CAUSEDBY or SUPPORTS edges. These represent choices made without recorded evidence.
2. **Single-source inferences** — Inference nodes supported by exactly one incoming edge. These represent conclusions drawn from a single piece of evidence, which are vulnerable to that evidence being incorrect.
3. **Low-confidence foundations** — Nodes with confidence below a threshold (default: 0.5) that have outgoing SUPPORTS or CAUSEDBY edges to higher-confidence nodes. These represent weak evidence propping up strong conclusions.
4. **Unstable knowledge** — Nodes involved in SUPERSEDES chains longer than 2, indicating beliefs that have been corrected multiple times.
5. **Stale evidence** — Nodes older than a configurable threshold (default: 30 days) that still support active decisions, flagging potentially outdated foundations.

Each detected gap includes the node ID, gap type, severity score, and a textual description. The output is ordered by severity, enabling the agent to prioritize which reasoning weaknesses to address first.

4.7 Cognitive Reasoning: Analogical Query

The analogical query finds past reasoning patterns structurally similar to a given target node's local subgraph. This implements a simplified form of structure-mapping [**?**] adapted for cognitive event graphs.

For each candidate node c, the algorithm computes:

$$\text{sim}(v, c) = 0.6 \cdot S_{\text{struct}}(v, c) + 0.4 \cdot S_{\text{content}}(v, c) \quad (4)$$

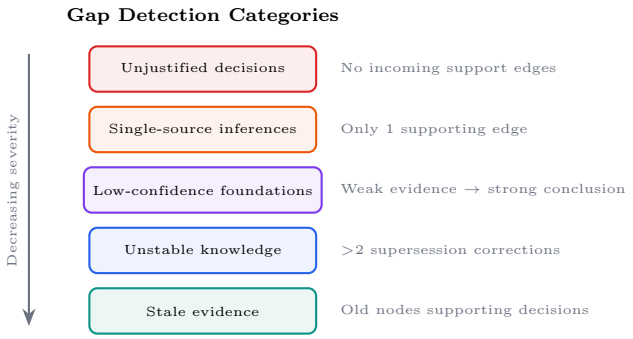

Figure 5: Five categories of reasoning gaps detected by the gap detection query, ordered by default severity. Each gap type identifies a specific structural weakness in the agent's reasoning graph.

where S_{struct} is based on the structural fingerprint—a vector of in-degree, out-degree, and edge-type distribution—and S_{content} is cosine similarity between feature vectors (when available) or token overlap (as fallback).

The structural fingerprint for a node v is:

$$\mathbf{f}(v) = (\deg_{\text{in}}(v),\ \deg_{\text{out}}(v),\ h_1(v),\ h_2(v),\ \ldots,\ h_7(v))$$

where $h_i(v)$ is the count of edges of type i incident to v. Two fingerprints are compared by cosine similarity over these 9-dimensional vectors.

The query returns the top-k analogous nodes (default $k = 5$), each annotated with the structural similarity score, content similarity score, and combined score. This enables an agent to ask: "have I encountered a similar reasoning pattern before?" and learn from past experience.

4.8 Graph Maintenance: Consolidation

The consolidation query performs four maintenance operations on the memory graph:

1. **Deduplication** — Identifies node pairs with content similarity above a threshold (default: 0.9) based on token overlap. Duplicate pairs are reported with their similarity scores. In execute mode, the lower-confidence duplicate is linked to the higher-confidence one via a SUPERSEDES edge.
2. **Contradiction linking** — Identifies fact nodes with contradictory content (detected by negation patterns and antonym heuristics) that lack CONTRADICTS edges. In execute mode, missing contradiction edges are created.
3. **Inference promotion** — Identifies inference nodes with confidence above 0.9 that are supported by 3+ sources. These are candidates for promotion to fact status, reflecting the principle that well-supported inferences should be treated as established knowledge.
4. **Orphan detection** — Identifies nodes with no edges (neither incoming nor outgoing), which represent isolated knowledge fragments that may need integration or removal.

Table 1: Algorithmic complexity and data structures for the nine new query types. V: nodes, E: edges, q: query terms, P: posting list size.

Query	Complexity	Key Structure
BM25 (fast)	$O(q \cdot \lvert P\rvert)$	HashMap
BM25 (slow)	$O(V)$	HashMap
Hybrid RRF	$O(V)$	HashMap
PageRank	$O(k(V+E))$	Vec
Degree centr.	$O(V+E)$	HashMap
Betweenness	$O(V'E)$	VecDeque
BFS shortest	$O(V+E)$	VecDeque
Dijkstra	$O((V+E)\log V)$	BinaryHeap
Belief revision	$O(V+E)$	VecDeque
Gap detection	$O(V \cdot E)$	HashMap
Analogical	$O(V^2)$	Vec
Consolidation	$O(V^2)$	HashMap
Drift detection	$O(V+E)$	BTreeMap

A *dry-run mode* reports what changes would be made without modifying the graph, enabling the agent (or user) to review proposed maintenance actions before committing them.

4.9 Graph Maintenance: Drift Detection

The drift detection query tracks how beliefs evolve over time by analyzing SUPERSEDES chains and confidence trajectories.

For each topic cluster (identified by session or content similarity), the algorithm computes:

- **Stability score** — The fraction of nodes in the cluster that have never been superseded. High stability indicates settled knowledge; low stability indicates an area of active revision.
- **Drift direction** — Whether corrections trend toward higher or lower confidence, indicating whether the agent is becoming more or less certain about a topic.
- **Revision frequency** — The rate of corrections per unit time, highlighting topics that are changing rapidly.
- **Trajectory** — A chronological list of supersession events for each drifting belief, enabling the agent to see the full evolution of its understanding.

The output is a per-topic report ordered by instability, allowing the agent to focus attention on its least settled areas of knowledge.

4.10 Summary of Algorithmic Complexity

Table 1 summarizes the algorithmic complexity and key data structures used by each new query type.

5 Evaluation

We benchmark all nine new query types on synthetic graphs at 10K and 100K nodes using the Criterion statistical benchmarking framework (100 samples each) and single-run measurements for computationally intensive queries. The hardware, software, and dataset generation methodology match the v0.1 evaluation [**?**]: Apple M4 Pro (ARM64), 64 GB unified memory, Rust 1.90.0 with `--release` optimizations.

Table 2: Query latency on 100K-node graph (Apple M4 Pro, release mode). Criterion benchmarks use 100 samples; single-run measurements marked with †.

Category	Query	Latency
Retrieval	BM25 (fast path)	1.58 ms
	BM25 (slow path)	122 ms
	Hybrid (BM25 + vector)	10.83 ms
Structure	PageRank (α=0.85)	34.3 ms
	Degree centrality	20.7 ms
	Betweenness centrality	10.1 s
	Shortest path (BFS)	104 μs
	Shortest path (Dijkstra)	17.6 ms
Cognitive	Belief revision	53.4 ms
	Gap detection†	297 s
	Analogical query†	229 s
Maintenance	Consolidation (dry)†	43.6 s
	Drift detection	68.4 ms

5.1 Query Performance Overview

Table 2 reports latency for all nine new query types on a 100,000-node graph (300K edges, 3 edges/node average). Queries span six orders of magnitude in latency, from 104 μs (bidirectional BFS) to 297 s (gap detection), reflecting fundamental differences in algorithmic complexity.

The results divide cleanly into three tiers (Figure 6):

Interactive queries (<100 ms). BM25, hybrid search, PageRank, degree centrality, BFS, Dijkstra, belief revision, and drift detection all complete within interactive latency bounds. These are suitable for per-query execution during agent conversations.

Periodic queries (1–60 s). Betweenness centrality (10.1 s) and consolidation (43.6 s) are too slow for interactive use but practical for periodic batch execution—e.g., running once per session or on a schedule.

Offline queries (>60 s). Gap detection (297 s) and analogical reasoning (229 s) at 100K nodes exhibit superlinear scaling due to their $O(V \cdot E)$ and $O(V^2)$ core loops, respectively. These are designed for offline analysis of large memory graphs, or for interactive use on smaller graphs (both complete in under 3 s at 10K nodes, as shown in Table 4).

5.2 Index Acceleration: Fast vs. Slow Path

Table 3 isolates the impact of the TermIndex on BM25 performance.

The inverted index provides a 77× speedup at 100K nodes, with the advantage growing as graph size increases

Table 3: BM25 text search: fast path (TermIndex) vs. slow path (full scan).

Graph Size	Fast Path	Slow Path	Speedup
10K nodes	186 μs	8.59 ms	46×
100K nodes	1.58 ms	122 ms	77×

Table 4: Query latency scaling from 10K to 100K nodes.

Query	10K	100K	Ratio
BM25 (fast)	186 μs	1.58 ms	8.5×
Hybrid	1.00 ms	10.83 ms	10.8×
PageRank	2.53 ms	34.3 ms	13.6×
Degree centr.	1.73 ms	20.7 ms	12.0×
Betweenness	6.43 s	10.1 s	1.6×
BFS	7.9 μs	104 μs	13.2×
Dijkstra	888 μs	17.6 ms	19.8×
Belief revision	6.26 ms	53.4 ms	8.5×
Gap detection	1.53 s	297 s	194×
Analogical	2.40 s	229 s	95×
Consolidation	352 ms	43.6 s	124×
Drift detect.	5.84 ms	68.4 ms	11.7×

because the fast path cost depends on posting list size (sublinear in n) while the slow path is always $O(n)$. This validates the TermIndex as a critical investment for backward-compatible acceleration.

5.3 Scaling Analysis

Table 4 reports latency at 10K and 100K nodes for all queries, revealing scaling behavior.

Figure 7 visualizes the scaling ratios for all queries. Queries fall into two scaling classes. *Near-linear scalers* (ratio ≈10× for a 10× size increase): BM25, hybrid, PageRank, degree, BFS, Dijkstra, belief revision, and drift detection. These exhibit $O(n)$ or $O(n \log n)$ behavior and remain practical at scale. *Super-linear scalers* (ratio ≫10×): gap detection (194×), consolidation (124×), and analogical (95×). These contain $O(n \cdot k)$ inner loops where k grows with graph density, producing $O(n^2)$ or worse behavior on dense graphs. The betweenness centrality ratio (1.6×) appears anomalous because Brandes' algorithm samples a fixed number of sources regardless of graph size; its absolute cost remains high (10.1 s) due to the BFS cost per source on denser graphs.

Figure 9 plots latency at 10K and 100K for the interactive-tier queries on a log–log scale.

6 Python SDK Extensions

The Python SDK (`pip install agentic-brain`) exposes all nine new queries through the `Brain` class, maintaining the zero-dependency design principle: the SDK communicates with the Rust engine via CLI subprocess calls and JSON serialization.

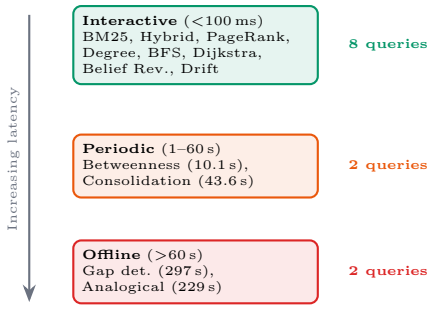

Figure 6: Three-tier query classification at 100K nodes. Eight of thirteen queries complete within interactive latency, making them suitable for per-query execution during agent conversations. Periodic and offline queries are designed for batch execution.

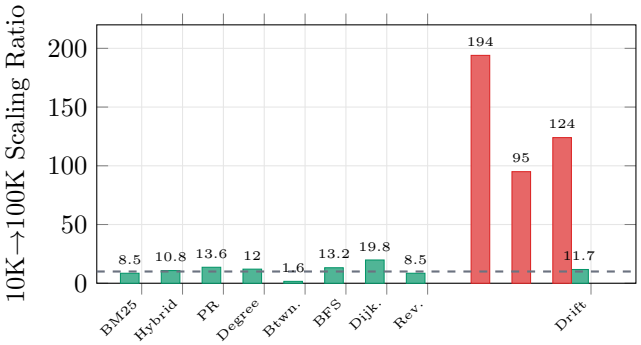

Figure 7: Scaling ratio (latency at 100K / latency at 10K) for each query. A ratio of 10 indicates linear scaling (dashed line). Queries below or near this line are practical at any graph size. Gap detection (194×), consolidation (124×), and analogical (95×) exhibit super-linear scaling.

6.1 New Brain Methods

Nine new methods map directly to the query types:

Listing 1: Python SDK methods for the nine new query types.

```
brain = Brain("agent.amem")

# Retrieval
results = brain.search_text("API rate limit")
results = brain.search("caching strategy",
                       top_k=10)

# Structural analysis
scores = brain.centrality(metric="pagerank")
path   = brain.shortest_path(src=42, dst=99)

# Cognitive reasoning
report = brain.revise(node_id=42)
gaps   = brain.gaps()
matches = brain.analogy(node_id=42, top_k=5)

# Maintenance
report = brain.consolidate(dry_run=True)
drift  = brain.drift()
```

Each method returns a typed dataclass from `agentic_memory.results`: `TextMatch`, `HybridMatch`, `PathResult`, `RevisionReport`, `GapReport`, `Analogy`,

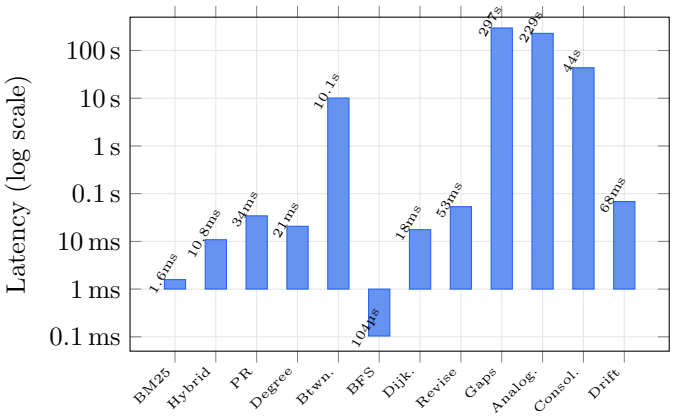

Figure 8: Query latency across all nine new query types on a 100K-node graph (log scale). Six queries complete under 100 ms (interactive). Gap detection, analogical reasoning, and consolidation require seconds to minutes, indicating they are suited for periodic or offline execution at this scale.

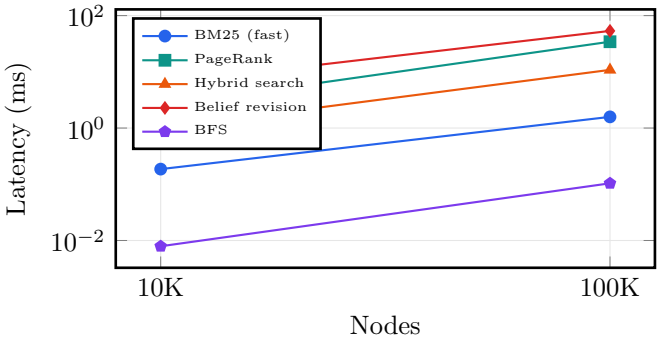

Figure 9: Scaling of interactive-tier queries from 10K to 100K nodes (log–log). All five queries exhibit near-linear scaling, remaining under 100 ms at 100K nodes.

`ConsolidationReport`, and `DriftReport`. These provide structured access to query outputs with named fields, type hints, and docstrings.

6.2 Test Coverage

The v0.2 Python SDK includes 104 tests across 8 test modules:

The `test_query_expansion.py` module specifically validates all nine new methods with a `populated_brain` fixture that creates a graph of 7 nodes across 3 sessions with typed edges. Tests cover: parameter validation, result type verification, empty-graph edge cases, event-type filtering, and metric selection.

7 Implementation Summary

Table 6 summarizes the implementation effort for v0.2.

Table 5: Python SDK test distribution across modules.

Module	Tests
test_brain.py	30
test_query_expansion.py	20
test_models.py	18
test_integrations.py	12
test_agent.py	8
test_cli_bridge.py	7
test_package.py	7
conftest.py (fixtures)	2
Total	**104**

Table 6: Implementation metrics for the v0.2 query expansion.

Metric	Value
New Rust code (query algorithms)	3,586 lines
Total Rust code (engine)	9,688 lines
Total Python SDK code	2,298 lines
Rust tests (total)	179
Python tests (total)	104
New external dependencies	0
New CLI commands	9
New Python Brain methods	9
New result dataclasses	8

Zero new dependencies. All graph algorithms—PageRank iteration, Brandes' betweenness centrality, bidirectional BFS, Dijkstra's algorithm, BM25 scoring, RRF fusion, structural fingerprinting, and cascade propagation—are implemented using only `std::collections` types (`HashMap`, `HashSet`, `BinaryHeap`, `VecDeque`, `BTreeMap`). This maintains the self-contained binary philosophy established in v0.1: the `amem` CLI has no runtime dependencies beyond the operating system.

Nine new CLI commands. Each query type is exposed as a CLI subcommand: `amem text-search`, `amem hybrid-search`, `amem centrality`, `amem path`, `amem revise`, `amem gaps`, `amem analogy`, `amem consolidate`, and `amem drift`. All commands support `--json` output for programmatic consumption by the Python SDK.

8 Discussion

From memory store to reasoning engine. The v0.1 query model treated the memory graph as a navigable store: retrieve a node, follow its edges, find similar nodes. The v0.2 queries enable a qualitatively different mode of interaction: the agent can now analyze the *structure* of its knowledge (centrality, shortest paths), audit the *quality* of its reasoning (gap detection, belief revision), and *maintain* its knowledge base autonomously (consolidation, drift detection). This transforms AgenticMemory from a persistence layer into a cognitive reasoning substrate.

Cognitive gap detection as self-reflection. The gap detection query is particularly novel in the context of agent memory. By identifying unjustified decisions, single-source inferences, and low-confidence foundations, it enables a form of epistemic self-reflection: the agent can assess the quality of its own reasoning without external supervision. This is related to metacognitive monitoring in cognitive science [**?**] and could support more calibrated decision-making in autonomous agents. The query is computationally expensive at 100K nodes (297 s) due to its $O(V \cdot E)$ analysis of support chains, but completes in 1.5 s at 10K nodes—fast enough for end-of-session execution. A natural optimization is incremental gap detection that only re-evaluates nodes modified since the last run.

Analogical reasoning: offline at scale, interactive at moderate scale. The analogical query enables a form of case-based reasoning [**?**] over the agent's entire history. By finding structurally similar past reasoning patterns, the agent can transfer lessons learned from previous situations to novel ones. The structural fingerprint approach captures meaningful structural similarity beyond surface-level text matching, but its $O(V^2)$ pairwise comparison loop makes it expensive at scale: 229 s at 100K nodes (Table 2). At 10K nodes—a practical size for single-agent memory accumulated over weeks of use—analogical search completes in 2.4 s, making it viable as a periodic batch operation. Approximate nearest-neighbor indexes over structural fingerprints would reduce this to sub-second even at 100K nodes.

Backward compatibility as a design principle. The tag-length-value index block and feature-flags mechanism demonstrate that binary formats can evolve without breaking compatibility. This is particularly important for agent memory, where files may persist for months or years and be accessed by different versions of the software. The slow-path fallback ensures that new queries always work, albeit more slowly, on old files.

Scalability tiers. The benchmarks reveal a clear three-tier scalability picture. Eight of thirteen queries remain interactive (<100 ms) at 100K nodes. Betweenness centrality (10.1 s) and consolidation (43.6 s) are periodic-tier operations. Gap detection (297 s) and analogical reasoning (229 s) are offline-tier at 100K nodes due to super-linear scaling (194× and 95× ratios for a 10× size increase, respectively). Practical agent memory graphs—typically 1K–20K nodes accumulated over weeks of interactions—fall comfortably within interactive range for all queries.

Other limitations. The BM25 tokenizer is a simple whitespace-and-punctuation splitter; a more sophisticated tokenizer (stemming, subword decomposition) would improve recall for morphological variants. Betweenness centrality with exact computation is $O(VE)$, which is impractical for graphs exceeding 100K nodes; our sampling approximation trades accuracy for speed. The consolidation deduplication uses token overlap rather than semantic similarity, which may miss paraphrases. The drift detection currently identifies instability but does not predict future drift trajectories.

Future work. Promising extensions include: (1) approximate nearest-neighbor indexes for faster analogical search, (2) incremental PageRank updates that avoid full recomputation when nodes are added, (3) semantic deduplication using the existing feature vectors, (4) temporal pattern mining to detect recurring reasoning cycles, and (5) integration with LLM self-correction loops where gap detection triggers targeted knowledge acquisition.

9 Conclusion

We have presented AgenticMemory v0.2.0, extending the binary graph memory format introduced in [**?**] with nine new query types spanning retrieval, structural analysis, cognitive reasoning, and graph maintenance. The expansion is implemented in 3,586 lines of Rust with zero new dependencies, maintaining the self-contained design philosophy of the original system.

The new queries transform AgenticMemory from a navigable memory store into a cognitive reasoning engine. BM25 and hybrid search provide keyword-based retrieval complementing existing vector similarity. PageRank, betweenness centrality, and shortest-path algorithms reveal the topological structure of the agent's knowledge. Counterfactual belief revision, reasoning gap detection, and analogical pattern matching enable epistemic self-reflection. Consolidation and drift detection support autonomous knowledge maintenance.

Criterion benchmarks on 100,000-node graphs show that eight of thirteen queries complete within interactive latency (<100 ms), with BM25 search at 1.58 ms and bidirectional BFS at 104 μs. Computationally intensive queries—gap detection (297 s) and analogical reasoning (229 s) at 100K nodes—are designed for periodic or offline execution and remain interactive at typical agent memory sizes (<20K nodes). Backward compatibility ensures that v0.1 files remain fully functional with v0.2 software via slow-path fallback (77× slower for BM25 without the TermIndex, but correct), and that v0.2 files degrade gracefully when read by v0.1 software via tag skipping and feature flags.

The complete system—9,688 lines of Rust, 2,298 lines of Python, 283 tests—demonstrates that sophisticated graph analytics can be embedded directly in an agent's memory layer without external services, databases, or cloud dependencies. Together with the foundation established in [**?**], AgenticMemory v0.2.0 provides a complete cognitive memory substrate: persistent, portable, navigable, analytically rich, and entirely self-contained.