ChatGPT

Unified Vector+Graph Databases in 2025: State of the Art

Combining vector similarity search with graph databases is an emerging trend, often referred to as **"memory graphs"** [1] [2]. The idea is to pair **vector embeddings** (for semantic similarity) with **explicit graph relationships** (for context and reasoning) in one system. This promises richer retrieval: not just *what* is similar, but *why* and *how items connect* [1]. Below we survey existing systems that implement this unification, discuss whether they truly integrate these capabilities or rely on hybrids, and highlight research on the benefits and trade-offs of unified vs. dual architectures.

Integrated Vector-Graph Database Systems

Many modern databases now offer **integrated vector search and graph storage** within a single architecture – providing a unified query engine, storage layer, and indexing for both embeddings and relationships:

- **Weaviate** – An open-source *"vector-native graph database"* [3]. Weaviate stores objects with vector embeddings and allows **GraphQL** queries that traverse connections between objects. It combines semantic vector search with structured knowledge graph links in one system [3]. (Open source; GitHub)

- **Neo4j** – A popular graph DB that added **native vector indexing** in 2023. Neo4j 5+ supports HNSW indexes on node properties, enabling *approximate nearest-neighbor* search inside Cypher queries [4] [5]. For example, a single Cypher query can find top-*k* similar products by embedding and then **traverse relationships** like `(:Product)-[:HAS_CATEGORY]->(:Category)` in one go [6] [5]. This is a true unification – vector search is **fully integrated** into Neo4j's engine and query language [4] (no external service needed).

- **TigerGraph (TigerVector)** – A distributed graph DB (commercial) that introduced native vector support in v4.2 (Dec 2024) [7]. **TigerVector** extends TigerGraph's schema with an *embedding* data type and integrates an HNSW index with the graph engine [8] [9]. Its GSQL query language was enhanced to support combined queries (pure vector search, vector+graph filtered search, etc.) [10]. This unified design allows *hybrid RAG* queries and showed **performance advantages** over using separate systems – outperforming Neo4j, Amazon Neptune, and even a specialized vector DB (Milvus) in the authors' benchmarks [11]. (TigerVector is described in a 2025 research paper [9] [12].)

- **NebulaGraph** – An open-source distributed graph DB (with an SQL-like **openCypher** query language). In 2025 NebulaGraph added *"native vector processing"*, integrating vector similarity search into its graph model [13]. Vectors become part of nodes' attributes, so one can perform **graph traversals and ANN searches within the same query** [14]. This unified approach enables, for example, finding entities by embedding similarity and immediately exploring their network relationships [15] (all inside Nebula's engine).

- **ArangoDB** – A multi-model database (documents, graphs, etc.) that recently integrated **vector search** directly into its core. ArangoDB uses Facebook's FAISS under the hood but exposes it natively in AQL (its query language) [16]. Vector search in Arango is treated as just another data index, fully **unified with graph traversal** and document queries [16] [17]. For example, one can query for similar embeddings with `APPROX_NEAR_COSINE()` and then traverse graph edges on the resulting documents, all in one AQL query [18] [19]. (Open source)

- **Memgraph** – An in-memory graph database (property graph model) that added **vector indexing** in version 3.x. Memgraph supports vector indices on both nodes *and* edges, enabling semantic similarity searches alongside Cypher-style graph queries [20]. This is a unified feature within Memgraph's engine (available out-of-the-box since v3.2) [21]. (Closed source core, with a community version)

- **Dgraph** – An open-source distributed graph DB, which in its v24 release introduced a vector data type and similarity search in DQL (Dgraph's query language) [22]. Using Dgraph, one can store embeddings as predicates and perform **knn searches directly in graph queries**. This effectively turns Dgraph into a hybrid graph+vector store (with a single system handling both).

- **SurrealDB** – A recently popular multi-model DB (document+graph) that is *"purpose-built for AI systems"* and supports **vector search** alongside graph relations [23]. SurrealDB's unified query engine can combine structured SQL-like queries, graph traversals, full-text search, and vector similarity in one interface [24]. (Open source)

- **FalkorDB** – An open-source graph database designed for **GraphRAG** and AI use-cases. FalkorDB natively integrates vector indexes with a low-latency graph engine [25]. It provides *"unified data storage"* – vectors stored alongside graph entities – and allows querying both in one system [26]. Advanced query optimizations in Falkor aim to efficiently run *combined* similarity + multi-hop graph queries [27]. In other words, FalkorDB eliminates the need for separate vector and graph stores [28]. (Open source, built on a Redis-based backend)

- **ApertureDB** – A specialized open-source database for **multimodal data** that combines a graph store for metadata with an integrated vector search engine. ApertureDB uses an in-memory property graph to store relationships, and FAISS for embedding similarity – all accessed through a unified **JSON query API** [29] [30]. This lets users store images/videos with embeddings and query by vector similarity **and** graph filters in one place [31] [30].

- **HelixDB** – A new open-source **graph-vector database** (Rust-based) designed explicitly for unified vector+graph operations. HelixDB's query language allows *hybrid traversals*: e.g. do a `SearchV(query_vector)` to get the top-*k* nearest nodes, then traverse their edges (like `similar_nodes->OUT<Friends>`), all in one query [32]. The engine is built from scratch to support both modes with high performance (~2ms vector search, sub-millisecond graph hops, per their benchmarks) [33] [34]. By **combining** graph and vector in a single type-safe engine, HelixDB claims to halve the complexity and cost compared to maintaining separate systems [33] [35]. (Open source, AGPL-3.0 [36])

- **Commercial cloud graph databases** have also joined the trend. For example, *Amazon Neptune* (AWS's managed graph DB) introduced vector embeddings support in its "Neptune ML/Analytics" features, allowing similarity queries on graph data [12]. Likewise, **Azure Cosmos DB** (multi-model NoSQL with graph API) added **hybrid indexing** for vector search and graph queries under one service [37]. Even enterprise platforms like SAP HANA are integrating a vector engine alongside their graph engine for unified querying [38]. These are essentially single-system solutions (albeit proprietary) for graph+vector retrieval.

Bottom line: There is a *broad landscape* of systems – both open-source and commercial – that natively unify vector similarity search with graph/knowledge graph functionality. In these examples, the integration is "true" in the sense that you **don't need two separate databases** – a single engine handles storage and querying of both embedding vectors and graph relationships. The query languages have been extended (e.g. Cypher, GSQL, AQL, GraphQL, etc.) so that a developer can perform hybrid searches in one query and transaction.

Unified vs. Hybrid Architectures

The alternative to a unified architecture is a **hybrid setup** where a vector database and a graph database are used side by side. In a hybrid approach, one might store embeddings in a dedicated vector index (like Pinecone, Milvus, etc.) and maintain connections in a separate graph store (like Neo4j or Neptune), then combine results at the application layer. This was common in early Retrieval-Augmented Generation pipelines: e.g. use a vector DB to fetch relevant documents by similarity, then use a graph DB to re-rank or add knowledge graph context. While this works, it requires **orchestrating two systems**, duplicating data (or storing cross-references), and handling consistency manually. Queries typically need to be split: first call the vector search API, then use the IDs to query the graph DB – adding latency and complexity.

Trade-offs: A *unified system* avoids those issues by keeping all data in one place with one query interface. Benefits of true unification include:

- **Integrated Querying & Indexing:** You can issue one query that mixes vector similarity and graph traversal without writing glue code or moving data between systems [32]. The index structures can be optimized together (e.g. TigerGraph's vector index interoperates with its MPP graph engine for parallel search [39]).

- **Consistency and Simplicity:** There's a single source of truth – no need to sync updates across a graph store and a separate vector store [40]. This reduces "data silo" problems and simplifies application logic [40].

- **Transactional Support:** Some unified databases support transactions that span both vector and graph data. For instance, TigerVector allows atomic updates of nodes that include both normal properties and embedding fields [41]. In a dual system, it's hard to get *ACID guarantees* when updating an item's embedding and its relationships across two services.

- **Lower Operational Overhead:** Maintaining one system (scaling, backups, security, etc.) is easier than two. The HelixDB team, for example, claims up to *50% reduction in infrastructure cost* by eliminating the separate vector DB tier [42] [35].

However, unified architectures can have **trade-offs** too:

- **Maturity & Specialization:** Vector search is a new feature in many graph databases, so performance might initially lag specialized vector-only solutions. Early implementations (e.g. Neo4j's first vector index releases, or Neptune's support) were sometimes less feature-rich – lacking advanced ANN algorithms or filtering capabilities that standalone vector DBs offer [12]. That said, newer systems like TigerVector have worked to close this gap, showing *comparable or superior performance* to a pure vector DB in their tests [11].

- **Complexity in the Engine:** Supporting heterogeneous workloads (graph traversals vs. high-dim vector math) in one engine is non-trivial. It demands careful engineering to optimize both. Some solutions essentially embed an existing vector library (FAISS, HNSWlib) inside the graph DB. Others (like HelixDB or FalkorDB) build from scratch for this dual purpose. It's an active area of optimization and may increase the complexity of the database internals.

- **Use-Case Fit:** If your application heavily uses one paradigm and not the other, a unified DB might add overhead. For example, if you only need vector search and hardly use relationships, a pure vector store might be simpler. Conversely, if you only need graph queries on structured data, adding a vector index might be unnecessary. Unified systems shine most for **hybrid use cases** where both semantic similarity and relational queries are essential (e.g. complex question answering, recommendation with content+collaborative signals, AI agent memory, etc.).

In summary, truly unified architectures are increasingly viable and help avoid the latency and complexity of dual setups [9] [40]. Most new entrants in this space emphasize that they remove the need for "juggling two databases" and enable one-step hybrid queries [33] [35]. The trade-off is that the unified solutions must match the performance of specialized systems on their respective tasks – a challenge that ongoing development and research are addressing.

Research & Literature on Vector-Graph Fusion

The fusion of vector and graph databases has also attracted attention in academic and industry research, especially due to the rise of **Retrieval-Augmented Generation (RAG)** for LLMs. Key themes include architecture design, query performance, and emergent retrieval capabilities from unification:

- **TigerGraph's TigerVector Paper (SIGMOD 2025)** – This paper explicitly discusses integrating vector search into a graph database [9]. The authors argue for a unified system (which they call **VectorGraphRAG**) instead of two separate DBs, listing the benefits we noted: reduced data movement, direct linking of embeddings to source data, unified query language for hybrid search, and single-point access control [9] [40]. They also identify challenges like maintaining high performance and supporting filtered or joined queries in a graph context [43]. TigerVector introduces techniques (e.g. decoupling vector storage from other properties, parallel query execution) to overcome these and demonstrates improved *hybrid query performance* over both other graph DBs and a specialized vector DB [11]. In short, this research validates that a well-designed unified system can achieve efficiency without sacrificing capability.

- **Comparative Analyses:** The community is actively comparing **Graph RAG vs. Vector RAG** approaches. For example, one benchmark (from FalkorDB) found that adding knowledge graph context to vector queries can improve retrieval **accuracy ~2.8×** for complex queries [44]. Knowledge graphs help disambiguate and filter vector results, reducing the "noise" of purely similarity-based recall [44]. Academic case studies (e.g. on personal memory chatbots or domain-specific QA) also report that graph-based retrieval reduces LLM hallucinations and raises precision/recall compared to vector-alone RAG [45] [46]. In a *TOBUGraph* experiment for personal photo memories, a dynamic memory graph outperformed standard vector RAG in both precision and recall, yielding more accurate answers and fewer errors [45] [46]. These results point to **emergent benefits** when unifying vectors with relationships: the system can retrieve not just semantically similar items, but also ensure they are *contextually relevant and connected*, leading to more reliable downstream use (like LLM responses).

- **Hybrid Query Languages:** Research is also exploring how to make query languages that naturally express hybrid searches. Neo4j's approach was to extend Cypher with a procedure call (`db.index.vector.queryNodes`) for ANN search [6]. TigerGraph extended GSQL with new syntax for vector similarity joins [10]. Emerging systems like HelixDB even created a custom query DSL (`.hx` queries) with built-in vector search operators and graph traversal in one step [32]. The design challenge is to allow **composable queries** that mix modalities without awkward workarounds. Early evidence suggests that a unified query layer can indeed execute complex combinations efficiently, especially if the planner/optimizer is aware of both vector indexes and graph topology.

Overall, the literature (and whitepapers/blogs from vendors) concur that *unifying* vector and graph capabilities yields richer querying and can improve retrieval outcomes for advanced applications [47] [9]. There is recognition that neither vectors nor graphs alone solve all problems: **similarity search is great for recall**, but needs graphs for reasoning and precision; graphs are great for structured knowledge, but need vectors to handle unstructured semantics [1] [2]. This complementary nature is driving both practical systems and theoretical frameworks (e.g. proposals for "Unified Index Layers" in AI stacks [48] and **HybridRAG** techniques that combine VectorRAG and GraphRAG [49] [50]).

Is the *MemoryGraph* Concept Novel?

The **MemoryGraph** concept – described as a *"vector-native graph database" with a hybrid query engine and unified indexing – is very much in line with the state of the art above. In fact, multiple systems already embody this concept. For example, Weaviate explicitly markets itself as a vector-native graph DB combining semantic search with explicit relationship modeling [3]. HelixDB, FalkorDB, and others are being built from the ground up with the same philosophy. Even legacy graph databases (Neo4j, TigerGraph, Dgraph, etc.) are converging toward this unified model by adding native vector support.

Novelty: In a broad sense, MemoryGraph is part of a *broader industry trend* rather than a completely novel one-off idea. The vision of blending vectors and graphs is shared by many in 2024–2025. That said, each implementation can have **meaningful differentiations**. MemoryGraph could distinguish itself via specifics like:

- **Unified Indexing Approach:** Perhaps a unique index structure that handles both vector similarity and graph connectivity in one engine (reducing the typical overhead of maintaining separate indexes). For instance, if MemoryGraph uses a single storage layout to serve both ANN queries and graph traversals efficiently, that would be an innovation. (Most current systems still maintain separate index structures internally – e.g. an HNSW index plus adjacency lists – and “orchestrate” between them [51]).

- **Hybrid Query Optimizer:** A query planner that intelligently decides when to use vector search vs. graph search or both, possibly yielding *emergent behaviors* like reasoning shortcuts. Neo4j and others require the user to explicitly call the vector search and then match graph patterns [6]. A more novel system might automatically combine these based on query intent.

- **Performance/Scalability**: If MemoryGraph is tuned for extremely low-latency or large-scale scenarios beyond what existing open solutions handle, it could stand out. For example, HelixDB boasts certain latency metrics [34]; MemoryGraph might aim even higher or handle larger graphs with millions of embeddings.

- **Use-case Focus:** “Memory graph” often implies use in AI agent memory or dynamic context storage. If MemoryGraph is tailored for those patterns (e.g. time-decay of memories, episodic grouping, etc.), it could offer features beyond a generic vector-graph DB. Some systems like **Graphiti (Neo4j’s experimental project)** and others in the agent memory space are exploring how structured memories + vectors can yield **emergent reasoning** for AI agents [52] [53]. MemoryGraph might have novel features on that front.

In summary, the concept of a unified vector+graph database is **not wholly unprecedented** – there are already several implementations and even a nascent product category around it. The *MemoryGraph* idea as described (hybrid engine, unified index) is a natural progression of database technology responding to AI needs [1] [2]. Its differentiation will depend on execution details. If MemoryGraph can demonstrate significantly better query performance, easier scaling, or smarter retrieval outcomes than existing unified systems, it would indeed advance the state of the art. Otherwise, it should be seen as one player in a rapidly evolving landscape of “graph-enhanced vector databases” (or conversely, vector-augmented graph databases). The trajectory is clear: **future AI-centric databases will likely be *both* vector and graph at their core**, blurring the line between the two. MemoryGraph’s vision aligns with this trajectory, and its novelty lies in how effectively it realizes the promise compared to peers.

Comparison of Unified vs. Dual Architectures

Approach	Description	Pros	Cons
Unified Vector+Graph DB	Single system handles graph storage *and* vector indexing/query. One integrated query engine (e.g. Cypher, GSQL, GraphQL) can combine nearest-neighbor search with graph pattern matching. Examples: Neo4j (with native vectors), Weaviate, TigerGraph, HelixDB, etc.	- One query to get both semantic and relational context (no app-level orchestration) [32] . - Consistent data and transactions across vectors and edges [40] . - Lower latency (in-memory joins of results) and no network hops between DBs. [47] - Simplified devops (only one system to deploy/scale).	- Feature maturity still catching up to dedicated solutions (e.g. vector index options, tooling). - Engine complexity: must optimize two different workloads together. - Possibly resource-intensive: handling high-dim vectors and graph operations might require more memory/CPU tuning in one engine.
Hybrid Two-DB Setup	Separate vector database and graph database, used in tandem. The application or a middleware handles combining results (e.g. find top-*k* IDs via vector search, then fetch/ connect them via graph DB). Often used in early RAG pipelines.	- Allows **best-of-breed** choice for each component (use the fastest vector DB and the most powerful graph DB independently). - Each DB can be scaled or tuned for its specific workload (e.g. vector index in GPU, graph on CPU). - Can be quicker to implement initially using existing platforms and orchestrating with code or tools like LlamaIndex.	- **Higher complexity**: two query languages, two data stores, integration code needed. - Data duplication or synchronization issues (need to keep embedding store and graph store in sync manually). - Cross-database queries incur extra latency (network calls, data transfer) and lack native join optimization [9] . - No unified transactional guarantees; harder to ensure consistency in updates across both.

Table: High-level comparison of unified vs. dual-system architectures for vector+graph use cases.

Sources:

1. Observability Guy, *"Why Vector Databases Will Soon Be Replaced by Memory Graphs," Medium*, Nov. 30, 2025 – explains how *memory graphs* combine vector similarity with relationships and reasoning [1] [2].

2. *Neo4j Blog*, Aug. 22, 2023 – announcement of **Neo4j's native vector search** integration (HNSW index) into its graph database [4] [6].

3. Sudhir Hasbe (Neo4j CPO), *Neo4j AuraDB Vector Index intro* – example Cypher query doing vector search + graph pattern in one step [6].

4. NebulaGraph v5.1 Release Notes (May 2025) – describes **native vector support** in NebulaGraph enabling unified graph traversal and similarity search in the same query [13] [15].

5. ArangoDB Blog, *"Vector Search in ArangoDB: Practical Insights…"* (2023) – details ArangoDB's integration of FAISS for vector search, **fully integrated** with its multi-model query engine [16] [17].

6. Memgraph Docs – *"Vector search – Memgraph"* (2024) – notes that Memgraph supports vector indexes on nodes/edges natively (available from v3.2) [20] [21].

7. TigerGraph TigerVector Paper (Li et al., 2025) – SIGMOD'25 paper introducing *TigerVector*, with unified vector+graph query support in TigerGraph. Discusses benefits of a unified system [9] [40] and limitations of prior partial solutions (Neo4j, Neptune) [12]. Shows TigerVector's performance beating Neo4j, Neptune, and Milvus in hybrid search benchmarks [54].

8. TigerGraph Blog, *"Vector Embeddings Reveal Hidden Layers in AI"* (Rajeev S., 2023) – emphasizes TigerGraph's **hybrid approach** (graph + vector in one platform) [55] and how it yields better accuracy/explainability for use cases like fraud detection and recommendations [56] [57].

9. FalkorDB Blog, *"Knowledge Graph vs Vector Database: Which to Choose?"* (2025) – highlights how FalkorDB unifies knowledge graphs and vector DBs. Reports a **2.8× improvement in query accuracy** when graph context is added to complex vector searches [44].

10. HelixDB Website – describes **HelixDB** as an open-source Rust-based *graph-vector database* unifying vector search and graph traversal in one engine [58]. Provides an example of a hybrid query mixing `SearchV` (vector search) and graph edge traversal [32].

11. HelixDB GitHub Repository – confirms HelixDB is open-source ("graph-vector database built from scratch in Rust") [36].

12. *TOBUGraph Paper* (Kashmira et al., 2024) – demonstrates a graph-based approach for personal memory retrieval. Combining an LLM with a **memory graph** yielded higher precision/recall and reduced hallucinations versus a pure vector RAG baseline [45] [46].

13. HybridRAG Paper (Chakraborty et al., 2024) – proposes **HybridRAG** that uses both vector and knowledge graph retrieval. Shows that using *both* a vector DB and a KG outperforms either alone for question answering [49] [50], reinforcing the value of hybrid retrieval.

14. Weaviate Documentation – positions Weaviate as a *vector database with knowledge graph capabilities*, i.e. a **vector-native graph DB** allowing semantic search with structured relations [3].

15. ApertureDB Docs – *"What is ApertureDB?"* – describes a unified multimodal DB with an in-memory **graph database for metadata** and integrated vector search using FAISS [29] [59], all accessible via a single query interface [30].

16. Arxiv preprint, *"Supporting Vector Search in Graph Databases for Advanced RAGs"* (TigerGraph team, 2025) – further details TigerVector's architecture: embedding as a new attribute type, decoupled storage, vector-search-in-graph query extensions, etc., enabling *hybrid vector+graph searches for RAG* [8] [10].

---

[1] [2] Why Vector Databases Will Soon Be Replaced by Memory Graphs | by Observability Guy | Nov, 2025 | Medium
https://observabilityguy.medium.com/why-vector-databases-will-soon-be-replaced-by-memory-graphs-82abd8d9eec3

[3] Vector Databases and Unstructured Data: Building Your Knowledge Graph | Datasumi
https://en.datasumi.com/vector-databases-and-unstructured-data-building-your-knowledge-graph

[4] [5] [6] Vector Search: Unlock Deep Insights for AI-Powered Apps
https://neo4j.com/blog/genai/vector-search-deeper-insights/

[7] [8] [9] [10] [11] [12] [39] [40] [41] [43] [47] [54] TigerVector: Supporting Vector Search in Graph Databases for Advanced RAGs
https://arxiv.org/html/2501.11216v1

[13] [14] [15] NebulaGraph Enterprise v5.1 Embeds Vector Search for AI-Grade Data Fusion
https://www.nebula-graph.io/posts/NebulaGraph_v5.1_Embeds_Vector_Search

[16] [17] [18] [19] Vector Search in ArangoDB: Insights & Hands-on Examples
https://arango.ai/blog/vector-search-in-arangodb-practical-insights-and-hands-on-examples/

[20] Vector search - Memgraph
https://memgraph.com/docs/querying/vector-search

[21] Vector Search Demo: Turning Unstructured Text into Queryable ...
https://memgraph.com/blog/vector-search-memgraph-knowledge-graph-demo

[22] Vector Similarity Search in DQL - Dgraph Documentation
https://docs.dgraph.io/howto/similarity-search/

[23] [24] SurrealDBVectorStore - Docs by LangChain
https://docs.langchain.com/oss/python/integrations/vectorstores/surrealdb

[25] [26] [27] [28] [44] Knowledge graph vs vector database: Which one to choose?
https://www.falkordb.com/blog/knowledge-graph-vs-vector-database/

[29] [30] [31] [51] [59] What is ApertureDB? | ApertureDB
https://docs.aperturedata.io/Introduction/WhatIsAperture

[32] [33] [34] [35] [42] [58] HelixDB | Native Graph-Vector Database
https://www.helix-db.com/

[36] GitHub - HelixDB/helix-db: HelixDB is an open-source graph-vector database built from scratch in Rust.
https://github.com/HelixDB/helix-db

[37] Why use Azure Cosmos DB for NoSQL for your AI applications?
https://learn.microsoft.com/en-us/azure/cosmos-db/gen-ai/why-cosmos-ai

[38] Unifying AI Workloads with SAP HANA Cloud
https://news.sap.com/2025/07/unifying-ai-workloads-sap-hana-cloud-one-database/

[45] [46] A Graph-Based Approach for Conversational AI-Driven Personal Memory Capture and Retrieval in a Real-world Application
https://arxiv.org/html/2412.05447v1

[48] RAG is evolving: Graph-RAG, Unified Index Layers, SELF ... - LinkedIn
https://www.linkedin.com/posts/sairam-sundaresan_rag-as-we-know-it-is-dying-the-next-wave-activity-7392157574638247936-KMsc

[49] [50] HybridRAG: Integrating Knowledge Graphs and Vector Retrieval Augmented Generation for Efficient Information Extraction
https://arxiv.org/html/2408.04948v1

[52] The AI-Native GraphDB + GraphRAG + Graph Memory Landscape ...
https://dev.to/yigit-konur/the-ai-native-graphdb-graphrag-graph-memory-landscape-market-catalog-2198

[53] Graphiti: Knowledge Graph Memory for an Agentic World - Neo4j
https://neo4j.com/blog/developer/graphiti-knowledge-graph-memory/

[55] [56] [57] Vector Embeddings Reveal Hidden Layers in AI - TigerGraph
https://www.tigergraph.com/blog/vector-embeddings-reveal-hidden-layers-in-ai/