

AkasicDB: Demonstrating Omni RAG with a Unified Vector-Graph-Relational DBMS

Geonho Lee Jeongho Park Donghyoung Han Min-Soo Kim*
ghlee5084@kaist.ac.kr jhpark@graphai.io dhhan@graphai.io minsoo.k@kaist.ac.kr
KAIST GraphAI GraphAI KAIST
Daejeon, Republic of Korea Daejeon, Republic of Korea Daejeon, Republic of Korea Daejeon, Republic of Korea

Abstract

Recent Retrieval-Augmented Generation (RAG) systems increasingly combine vector retrieval with structured knowledge, such as Graph RAG and Filtered vector search. However, existing database architectures struggle to support such complex RAG workflows efficiently, as they rely on out-of-DB pipelines or in-DB non-native integration, leading to high overhead. This demo paper presents AkasicDB, a database system that natively supports such RAG workflows by jointly executing vector similarity search, graph traversal, and relational filtering within a single execution framework. AkasicDB extends our prior work, Chimera, with native vector support to enable such unified execution. Based on AkasicDB, we demonstrate the first native integration of Vector-Graph-Relational RAG, which we refer to as *Omni RAG*. Through an interactive chat-style demonstration, users execute and visualize Omni RAG queries, directly experiencing its superior retrieval and reasoning over vector-only approaches while observing the practical limitations of existing database architectures in supporting Omni RAG. A demonstration video is available at https://youtu.be/8d09_dtrEIM

CCS Concepts

• **Information systems** → *Evaluation of retrieval results; Information retrieval query processing.*

Keywords

Retrieval-Augmented Generation, Vector-Graph-Relational DBMS

ACM Reference Format:

Geonho Lee, Jeongho Park, Donghyoung Han, and Min-Soo Kim. 2026. AkasicDB: Demonstrating Omni RAG with a Unified Vector-Graph-Relational DBMS. In *Companion of the International Conference on Management of Data (SIGMOD Companion '26)*, May 31-June 05, 2026, Bengaluru, India. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3788853.3801609>

1 Introduction

Retrieval-Augmented Generation (RAG) has emerged as a key paradigm for grounding Large Language Models (LLMs) with external knowledge to improve factuality and reasoning. In practice, RAG has predominantly been realized through vector-centric retrieval

pipelines, where unstructured text is retrieved via embedding similarity and passed to a language model. Building on this paradigm, recent studies have investigated Graph RAG [2, 8], which leverages graph traversal to improve multi-hop reasoning, as well as Filtered vector search [6], where relational predicates are applied to constrain vector retrieval for improved factual grounding. Despite these advances, Graph RAG and Filtered vector search have largely evolved in isolation, resulting in RAG pipelines that combine vector similarity search with either graph traversal or relational filtering [4], but no single workflow has demonstrated the joint integration of all three. This limitation primarily stems from the absence of a suitable database system architecture that can *natively integrate* these retrieval modalities within one execution framework. As a result, attempts to combine them typically rely on either orchestrating multiple native systems outside the DBMS or stitching the modalities together inside the DBMS through non-native extensions, which hinders unified execution, visualization, and interaction of such RAG workflows as a cohesive whole.

To address this architectural limitation, we leverage our prior work, Chimera [3], which introduced a unified query processing for graph and relational data. Building on this foundation, we extend Chimera with native vector support and develop AkasicDB, a database system that enables vector similarity search to be performed within the same execution framework together with graph traversal and relational filtering for RAG workflows.

Based on AkasicDB, we demonstrate the first native integration of Vector-Graph-Relational RAG, which we refer to as *Omni RAG*. The notion of native integration is explained in detail in Section 3. In the demonstration, users interact with a web-based interface that visualizes retrieved vector, graph, and relational data, along with a chat-style interface that runs the RAG pipeline. Through this demonstration, users can directly experience that *Omni RAG* provides more effective retrieval and reasoning than vector-only approaches. We further contrast this experience with representative *out-of-DB integration of native systems* (e.g., Neo4j, Milvus) as well as *in-DB non-native integration* based on PostgreSQL with graph and vector extensions (e.g., Apache AGE, pgvector), highlighting their practical limitations in supporting *Omni RAG*.

2 Omni RAG

Vector RAG embeds both user questions and document chunks into a shared vector space and retrieves based on embedding vector similarity. Figure 1(a) illustrates vector RAG using the example question “What are the essential skills and knowledge needed for new beekeepers to succeed?”. In this setting, the question embedding is used to retrieve a set of top-*k* document chunks that are semantically similar, which are then provided as context for generation. Vector RAG has proven effective for leveraging unstructured

*Corresponding author.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

SIGMOD Companion '26, Bengaluru, India

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2450-3/2026/05

<https://doi.org/10.1145/3788853.3801609>

textual knowledge at scale, and thus serves as the default retrieval paradigm in many contemporary RAG systems.

Meanwhile, graph RAG augments document chunks with an explicitly constructed knowledge graph, as illustrated in Figure 1(b). Rather than retrieving isolated text chunks, Graph RAG aims to provide richer and more comprehensive context by traversing entity relationships, such as those among Beekeeper, Hive, Colony, and Queen. Filtered vector search, illustrated in Figure 1(c), leverages structured metadata (e.g., timestamp or entity type) to constrain the search space before or during vector retrieval. While Graph RAG focuses on expanding the scope of retrieved information through relational structure, filtered vector search emphasizes excluding irrelevant content to improve precision.

Building on these complementary retrieval paradigms, we define *Omni RAG* as a RAG formulation that jointly accesses vector, graph, and relational data, and explicitly specifies what to retrieve from each modality. In Omni RAG, retrieval combines vector similarity search over unstructured text, graph traversal over entity relationships, and relational filtering over structured attributes.

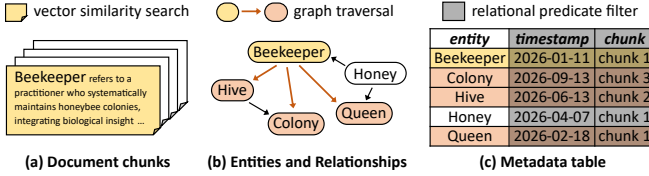


Figure 1: Example of Vector-Graph-Relational RAG.

3 AkasicDB System Overview

This section discusses why existing architectures make Omni RAG difficult in practice and how AkasicDB overcomes these limitations.

3.1 System Design for Omni RAG

A straightforward way to implement Omni RAG is to rely on multiple specialized systems, as illustrated in Figure 3(a). In this setting, a RAG application queries native vector, graph, and relational DBMSs separately and integrates their results outside the database—*out-of-DB integration of native systems*. Although this approach is functionally feasible, it incurs high operational costs to manage multiple systems and often leads to significant inefficiencies. Since the context length of the LLM is limited, RAG ultimately requires only the top- k most relevant results. However, when retrieval is performed across multiple systems, it is carried out in a multi-turn manner, making it difficult to determine how many results each system should produce. For example, the selectivity of graph traversal or relational filtering is not known in advance, and thus each system must conservatively generate more intermediate results than actually needed. These intermediate results are materialized, transferred to the application layer, and only then pruned by a final top- k restriction. As a result, this process often produces excessive intermediate data, leading to redundant data transfer, increased latency, and inefficient resource utilization.

To mitigate these issues, it is desirable to process Omni RAG through an integrated system. In such a setting, retrieval across graph, relational, and vector data can be expressed as a single SQL extended query. Figure 2 shows an example query, inspired by LightRAG [2] and extended with relational filtering, that jointly

retrieves over the three modalities followed by a final LIMIT k . Here, SQL/PGQ [1] extends SQL for graph pattern matching, while the $\leftrightarrow$ operator denotes vector similarity. This formulation enables the query to be planned and executed in a single turn, allowing the top- k constraint to be enforced during execution and avoiding excessive intermediate results.

```
SELECT chunk
FROM GRAPH_TABLE (
  MATCH (src:entity)-[:related_to]->(dst:entity)
  COLUMNS (
    src.embedding AS src_embedding,
    dst.chunk AS chunk,
    dst.timestamp AS timestamp
  )
) WHERE timestamp IN :selected_time_range
ORDER BY src_embedding <-> :question_embedding
LIMIT 5;
```

Figure 2: An SQL extended query example for Omni RAG.

Figure 3(b) illustrates an extension-based approach that enables a single-query formulation of Omni RAG—*in-DB non-native integration*. Highly extensible DBMSs support this approach by allowing graph querying and vector similarity search to be added through extensions. However, these capabilities remain non-native to the core engine. In practice, graph traversal is translated into sequences of relational joins, and vector data are typically stored and processed as generic array types. As a result, both graph traversal and vector search incur substantial overhead, and query optimization remains dominated by a relational planner that lacks awareness of graph- and vector-specific execution characteristics. Consequently, although Figure 3(b) supports the expression of Omni RAG queries, it still suffers from inefficient execution.

To address these limitations, we adopt the architecture shown in Figure 3(c) and develop AkasicDB, which realizes *in-DB native integration* for Omni RAG. AkasicDB consists of (i) dedicated storage components for vector, graph, and relational data, and (ii) a unified query processing layer that coordinates traversal, join, and similarity search within the same execution framework. This architecture provides the foundation for efficient Omni RAG processing, which we detail in the following subsections.

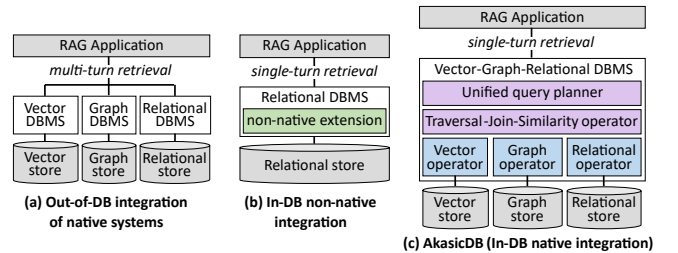


Figure 3: Comparison of architectures.

3.2 Vector-Graph-Relational Storage

AkasicDB extends the dual-store architecture of Chimera into a triple-store design by incorporating a native vector store alongside the graph and relational stores. The graph store maintains graph topology using adjacency-list, while the relational store is built on PostgreSQL tables. Vectors are stored separately in a dedicated

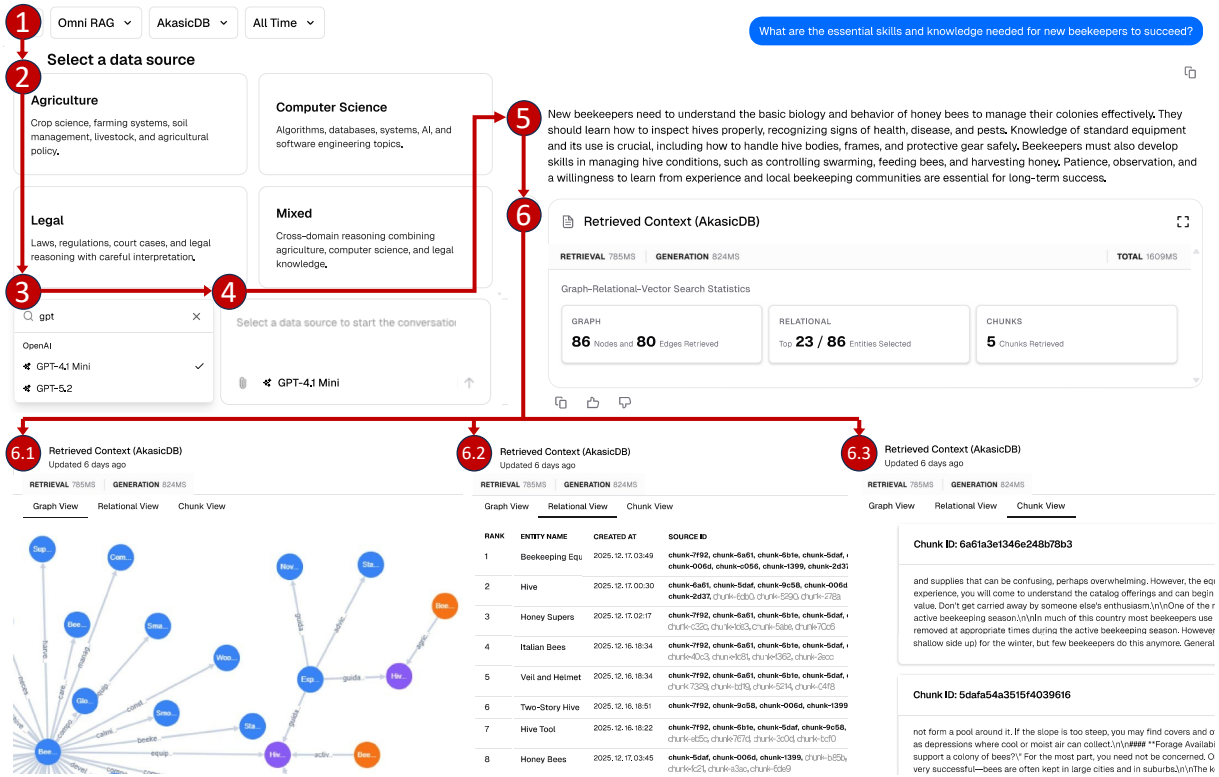


Figure 4: End-to-end demonstration workflow of Omni RAG.

disk-based storage called segments to support efficient similarity search. These three stores are co-located within a single system and share a common transaction manager, ensuring consistent and up-to-date access across graph, relational, and vector data.

AkasicDB provides indexing mechanisms for each store according to its data representation. Both the graph store and the relational store support B-tree indexes to accelerate attribute-based filtering and access to graph elements. For vector data, AkasicDB supports multiple indexes, including HNSW, IVF, and Vamana, which perform approximate nearest neighbor search to efficiently retrieve similar vectors at scale.

3.3 Unified Query Processing

AkasicDB builds on Chimera's Traversal-Join (TJ) operator, which enables graph traversal and relational joins to be executed within a single query plan by taking both topology and property operands as input. To support Omni RAG workloads, AkasicDB extends this operator by integrating vector similarity search following the VBASE [6] design. Specifically, AkasicDB exposes ANN index traversal as an iterator-based operator with an Open/Next/Close interface, allowing similarity search to be embedded into a Volcano-style execution framework. Under this design, all vector, graph, and relational operations incrementally produces records via Next, while global top- k semantics are enforced by upstream ORDER BY and LIMIT operators. As a result, operations across the three modalities are seamlessly composed within a single execution plan, enabling unified and efficient query processing for Omni RAG.

4 Demonstration Scenarios

4.1 End-to-End Experience

We demonstrate Omni RAG in six steps, as illustrated in Figure 4.

Step 1. Configuration. The demonstration begins by selecting the RAG mode and target system. Depending on this selection, users are guided into different demonstration scenarios: one focusing on comparing Vector RAG and Omni RAG, and another focusing on experiencing performance differences across systems.

Step 2. Data Source Selection. Users select a knowledge domain (e.g., Agriculture, Computer Science, Legal, or Mixed), enabling cross-domain Omni RAG over vector-graph-relational data.

Step 3. Model Selection. Users then select an LLM for answer generation from a list of supported models, allowing seamless switching between models within the same retrieval pipeline.

Step 4. Input Question. Users either select a suggested question or freely input their own natural language question. In the running example, the user asks: "What are the essential skills and knowledge needed for new beekeepers to succeed?".

Step 5. LLM Response. Based on the selected configuration, the system retrieves relevant context and presents the generated answer in a conversational chat interface using the chosen LLM.

Step 6. Retrieval Statistics and Inspection. Users inspect detailed retrieval statistics, including retrieval counts across modalities such as the number of vertices and edges traversed, relational entities selected, and text chunks retrieved. The interface reports retrieval time and generation time separately. Users can further expand this view to examine modality-specific results in detail:

Step 6.1. Graph view. visualizes the retrieved subgraph as an interactive Cytoscape layout, where vertex colors distinguish different entity roles and edges represent retrieved relationships.

Step 6.2. Relational view. shows retrieved entities in a tabular layout, with relevant records highlighted to indicate entities that contributed to the retrieved text chunks.

Step 6.3. Chunk view. displays the retrieved text chunks in a document-style layout, allowing users to directly inspect the textual evidence used for generation.

4.2 Scenario 1 – Answer Quality Comparison

In this scenario, users configure the system to generate answers using both Vector RAG and Omni RAG for the same question and data source. The interface presents the resulting answers side by side, allowing users to directly compare their quality and vote which response better. Alongside human voting, we also assess the same answers using an LLM-as-a-judge, which has been shown to correlate well with human judgments in natural language generation evaluation [5, 7]. This enables users to observe when automated evaluation agrees with or diverges from human preferences. Through this interaction, the scenario highlights how Omni RAG produces more informative and useful responses and engages users in understanding both answer quality and evaluation practices.

4.3 Scenario 2 – System Efficiency Comparison

In this scenario, users vary the backend system and adjust the date range to control retrieval selectivity. The interface reports retrieval time and generation time separately, allowing direct observation of how different system architectures respond. By issuing the same question under different configurations, users can experience how retrieval efficiency improves chat responsiveness, highlighting the impact of database system architecture on RAG performance.

5 Experiments

Settings. We evaluate Omni RAG queries on four datasets derived from the UltraDomain [2] benchmark: Agriculture (2.0M tokens), CS (2.3M tokens), Legal (5.1M tokens), and Mix (0.6M tokens). We compare AkasicDB against three alternative system configurations. PGVector+AGE denotes an in-DB non-native integration based on PostgreSQL, where vector and graph functionalities are provided via extensions (pgvector and Apache AGE). Neo4j+Milvus represents an out-of-DB integration of native systems, where graph data are managed by Neo4j, relational data are stored as key-value properties in Neo4j, and vector data are handled by Milvus. In this configuration, we use a fixed multi-turn retrieval that performs vector search in Milvus first (to leverage its vector index), followed by graph traversal and relational filtering in Neo4j. Neo4j Vector denotes an in-DB non-native integration that uses a fixed query plan, executing graph traversal and filtering in Neo4j first and then performing vector search via vector extension. We measure latency using time-to-first-token (TTFT), which includes both retrieval and generation time. All experiments were conducted on a single server with two Intel Xeon Gold 6326 CPUs and 1024 GB RAM.

Answer Quality Comparison. Table 1 (upper part) reports the answer quality of Vector RAG and Omni RAG using the winning ratio evaluated by an LLM-as-a-judge following [8]. Across all

Table 1: Winning Ratio (%) and Latency (msec) of Omni RAG.

Metric	Configuration	Datasets			
		Agriculture	CS	Legal	Mix
Winning Ratio	Vector RAG	45%	48%	47%	36%
	Omni RAG	55%	52%	53%	64%
Retrieval Time for Omni RAG	PGVector+AGE	18,573	21,278	11,079	4,728
	Milvus+Neo4j	7,099	3,001	4,783	1,536
	Neo4j Vector	4,042	4,497	2,409	1,323
	AkasicDB	802	942	697	583
Generation Time for Omni RAG	gpt-4.1-mini	883	876	882	881

datasets, Omni RAG consistently outperforms Vector RAG by 4–28 percentage points, with the largest gain on Mix. This indicates that jointly leveraging graph, relational, and vector information enables more accurate answers than vector-only retrieval.

System Efficiency Comparison. Table 1 (lower part) presents the system efficiency of Omni RAG in terms of latency under a selectivity setting of 20%. AkasicDB achieves the lowest retrieval time across all datasets, reducing latency by orders of magnitude compared to PGVector+AGE and by 2–9× compared to Neo4j and Milvus configuration. Generation time, which is dominated by LLM inference and thus incurs substantial latency, is reported for completeness; however, under complex Omni RAG retrieval settings, retrieval time can become a primary bottleneck.

6 Conclusions

This demonstration presented AkasicDB, a vector-graph-relational DBMS that enables efficient Omni RAG within a single system. We showed that existing architectures make Omni RAG difficult to realize in practice, and that AkasicDB addresses these challenges. Through interactive scenarios, users can compare the quality of Vector RAG and Omni RAG responses and directly experience performance differences across systems. Overall, this demonstration highlights the practical benefits of unified vector-graph-relational processing for emerging RAG workloads.

References

- [1] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindacker, Victor Marsault, Wim Martens, Jan Michels, et al. 2022. Graph pattern matching in GQL and SQL/PGQ. In *SIGMOD*.
- [2] Zirui Guo, Lianghao Xia, Yanhua Yu, Tu Ao, and Chao Huang. 2025. LightRAG: Simple and Fast Retrieval-Augmented Generation. In *EMNLP Findings*.
- [3] Geonho Lee, Jeongho Park, and Min-Soo Kim. 2024. Chimera: A system design of dual storage and traversal-join unified query processing for SQL/PGQ. *Proc. VLDB Endow.* (2024).
- [4] Peizheng Li, Chaoyi Chen, Hao Yuan, Zhenbo Fu, Hang Shen, Xinbo Yang, Qiang Wang, Xin Ai, Yanfeng Zhang, et al. 2025. NeutronRAG: Towards Understanding the Effectiveness of RAG from a Data Retrieval Perspective. In *SIGMOD*.
- [5] Jiaan Wang, Yunlong Liang, Fandong Meng, Zengkui Sun, Haoxiang Shi, Zhixu Li, Jinan Xu, Jianfeng Qu, and Jie Zhou. 2023. Is ChatGPT a Good NLG Evaluator? A Preliminary Study. In *Proceedings of EMNLP Workshop*.
- [6] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiaodong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, et al. 2023. VBASE: Unifying online vector similarity search and relational queries via relaxed monotonicity. In *OSDI*.
- [7] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *NeurIPS* (2023).
- [8] Yingli Zhou, Yaodong Su, Youran Sun, Shu Wang, Taotao Wang, Runyuan He, Yongwei Zhang, Sicong Liang, Xilin Liu, Yuchi Ma, et al. 2026. In-Depth Analysis of Graph-Based RAG in a Unified Framework. *Proc. VLDB Endow.* (2026).