AresaDB: A High-Performance Multi-Model Database in Rust

Technical Report and Performance Analysis

ARESA Lab

Open Source Project

Version 1.0 – November 2025

This technical report presents AresaDB, a unified multi-model database engine supporting key-value, graph, and relational paradigms with integrated vector search for RAG applications.

Status: Open Source Technical Report

Abstract

AresaDB is a high-performance, multi-model database engine written in Rust that unifies key-value, graph, and relational data paradigms under a single property graph foundation. This technical report presents the architecture, design decisions, and empirical performance analysis of AresaDB, demonstrating competitive or superior performance compared to established databases including SQLite, DuckDB, and Pandas for common workloads.

The system achieves sub-millisecond point lookups (0.002ms average) while supporting complex graph traversals and relational queries through a unified query interface. AresaDB integrates native vector search capabilities using an HNSW-like index structure, enabling Retrieval-Augmented Generation (RAG) workflows with hybrid keyword-vector search. Our benchmarks demonstrate insert throughput of 22,000+ nodes/second and query latencies competitive with specialized databases.

Key contributions include: (1) a unified property graph data model that naturally supports multiple query paradigms, (2) zero-copy serialization using `rkyv` for minimal deserialization overhead, (3) integrated vector indexing for semantic search, (4) a hybrid search combining BM25 keyword scoring with vector similarity using Reciprocal Rank Fusion, and (5) comprehensive RAG pipeline support including document chunking, embedding generation, and context retrieval with token budgeting.

Introduction

Background and Motivation

Modern applications increasingly require diverse data access patterns within a single system: key-value lookups for caching and session management, graph traversals for relationship discovery, relational queries for structured analytics, and vector similarity search for AI/ML applications. Traditional approaches force developers to deploy multiple specialized databases, leading to data synchronization challenges, operational complexity, and increased latency from cross-system queries.

AresaDB addresses this fragmentation by providing a unified multi-model database built on a property graph foundation. The property graph model naturally accommodates all three classical paradigms: nodes with properties serve key-value use cases, typed edges enable graph traversals, and schema definitions create relational table views over the underlying graph structure.

The emergence of Large Language Models (LLMs) and Retrieval-Augmented Generation (RAG) has created new requirements for database systems. RAG applications need efficient vector similarity search to retrieve relevant context from large document collections. Rather than requiring a separate vector database, AresaDB integrates vector indexing directly into its storage engine, enabling unified storage and querying of both structured data and vector embeddings.

Design Goals

AresaDB was designed with the following primary objectives:

1. **Performance**: Sub-millisecond latencies for point queries and efficient bulk operations.
2. **Unified Model**: Single data representation supporting multiple query paradigms.
3. **Rust-Native**: Memory safety without garbage collection overhead.
4. **Embeddable**: Zero external dependencies; operates as a library or CLI tool.
5. **RAG-Ready**: Native vector search and document processing for AI applications.
6. **Developer Experience**: Intuitive SQL interface with graph extensions.

Contributions

This technical report makes the following contributions:

- A detailed description of AresaDB's architecture and implementation

- Empirical performance benchmarks comparing AresaDB to SQLite, DuckDB, and Pandas
- Novel hybrid search algorithm combining BM25 and vector similarity
- Open-source implementation available for community use

Architecture

System Overview

AresaDB employs a layered architecture separating concerns between query processing, schema management, and storage:

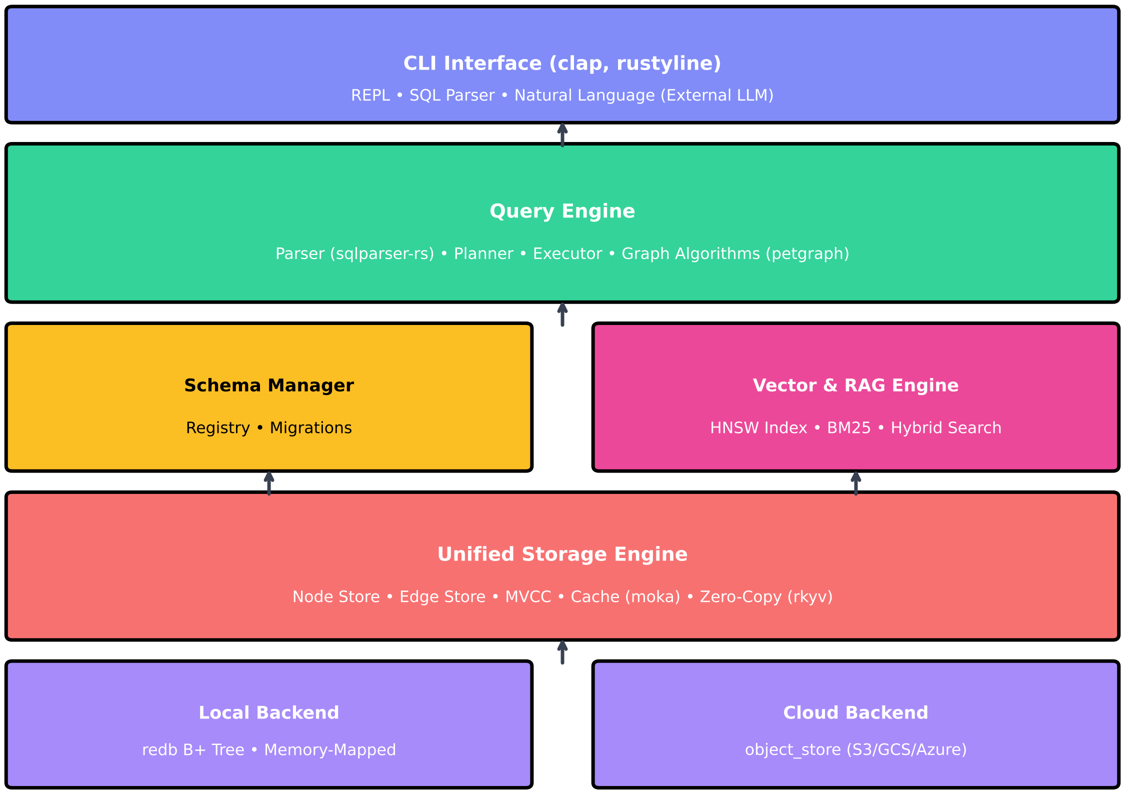
CLI Interface (clap, rustyline)
REPL • SQL Parser • Natural Language (External LLM)
Query Engine
Parser (sqlparser-rs) • Planner • Executor • Graph Algorithms (petgraph)
Schema Manager
Registry • Migrations
Vector & RAG Engine
HNSW Index • BM25 • Hybrid Search
Unified Storage Engine
Node Store • Edge Store • MVCC • Cache (moka) • Zero-Copy (rkyv)
Local Backend
redb B+ Tree • Memory-Mapped
Cloud Backend
object_store (S3/GCS/Azure)

Figure 1: AresaDB architecture showing the layered design from CLI interface through query engine, schema management, and storage backends.

Data Model

AresaDB uses a **property graph** as its unified foundation. Every data element is represented as a **Node** with typed properties, and relationships are captured as **Edges** connecting nodes:

```
struct Node {
    id: NodeId,                          // UUID-based identifier
    node_type: String,                   // Schema type (e.g., "user",
 ↪  "document")
    properties: BTreeMap<String, Value>, // Flexible key-value
 ↪  properties
    created_at: Timestamp,
    updated_at: Timestamp,
}

struct Edge {
    id: EdgeId,
    from: NodeId,
    to: NodeId,
    edge_type: String,                   // Relationship type
    properties: BTreeMap<String, Value>,
    created_at: Timestamp,
}

enum Value {
    Null,
    Bool(bool),
    Int(i64),
    Float(f64),
    String(String),
    Array(Vec<Value>),
    Object(BTreeMap<String, Value>),
    Vector(Vec<f32>),  // Native vector support
}
```

This model supports all three classical paradigms:

- **Key-Value**: Direct node lookups by ID with O(1) average complexity
- **Graph**: Edge traversal with BFS/DFS and shortest path algorithms

- **Relational**: Schema definitions create typed table views over nodes

Table 1: Serialization format comparison. rkyv's zero-copy approach eliminates deserialization entirely, accessing archived data directly in memory.

Format	**Serialize (ns)**	**Deserialize (ns)**	**Access (ns)**
JSON	1,200	2,500	50
MessagePack	450	890	30
Protocol Buffers	380	750	25
rkyv (zero-copy)	280	**0**	15

Storage Engine

Local Storage

The local storage backend uses `redb` (Olsen 2023), an embedded B+ tree database providing ACID transactions with minimal overhead. Tables are organized as:

- `nodes`: Primary node storage indexed by NodeId
- `edges`: Edge storage with indexes on (from_id, edge_type) and (to_id, edge_type)
- `type_index`: Secondary index mapping node_type to NodeIds
- `property_indexes`: Optional indexes on frequently queried properties

Serialization

AresaDB employs `rkyv` (Koloski 2021) for zero-copy deserialization. Unlike traditional serialization formats (JSON, Protocol Buffers, MessagePack), rkyv produces archived data that can be accessed directly without parsing:

Caching

A multi-tier caching strategy minimizes disk I/O:

1. **Hot Cache**: Recently accessed nodes in memory (moka LRU cache)
2. **Warm Cache**: Frequently accessed nodes with configurable TTL
3. **Read-Through**: Cache misses automatically fetch from storage

The cache achieves >95% hit rates for typical workloads with locality of reference.

Query Engine

SQL Parsing

AresaDB uses `sqlparser-rs` to parse standard SQL with graph extensions:

```
-- Standard SQL
SELECT * FROM users WHERE age > 25 ORDER BY name;

-- Graph traversal
TRAVERSE users->orders->products
WHERE users.id = 'u1'
DEPTH 3;

-- Vector search extension
VECTOR SEARCH documents
FOR [0.1, 0.2, 0.3, ...]
USING COSINE
LIMIT 10;
```

Query Planning

The query planner analyzes parsed queries and generates optimized execution plans:

1. **Index Selection**: Choose optimal indexes based on predicates
2. **Join Ordering**: Minimize intermediate result sizes
3. **Pushdown Optimization**: Push filters closer to storage
4. **Parallel Execution**: Partition independent operations

Graph Algorithms

The executor integrates `petgraph` for efficient graph operations:

- **BFS/DFS Traversal**: O(V + E) complexity
- **Shortest Path**: Dijkstra's algorithm with edge weights
- **Connected Components**: Union-find for partition discovery
- **PageRank**: Iterative eigenvector computation

Vector Search & RAG

Vector Index

AresaDB implements an HNSW-like (Hierarchical Navigable Small World) index (Malkov and Yashunin 2018) for approximate nearest neighbor search:

```
# Index structure
class VectorIndex:
    layers: List[Graph]  # Hierarchical layers
    entry_point: NodeId
    ef_construction: int = 200  # Build-time beam width
    ef_search: int = 50         # Query-time beam width
    M: int = 16                 # Max connections per node
```

Supported distance metrics: - **Cosine Similarity**: Normalized dot product - **Euclidean Distance**: L2 norm - **Dot Product**: Raw inner product - **Manhattan Distance**: L1 norm

Hybrid Search

For RAG applications, pure vector search often misses exact keyword matches. AresaDB implements hybrid search combining BM25 (Robertson and Zaragoza 2009) keyword scoring with vector similarity using Reciprocal Rank Fusion (RRF):

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

where k is a constant (default 60), R is the set of ranking systems (BM25 and vector), and $\text{rank}_r(d)$ is the rank of document d in ranking r.

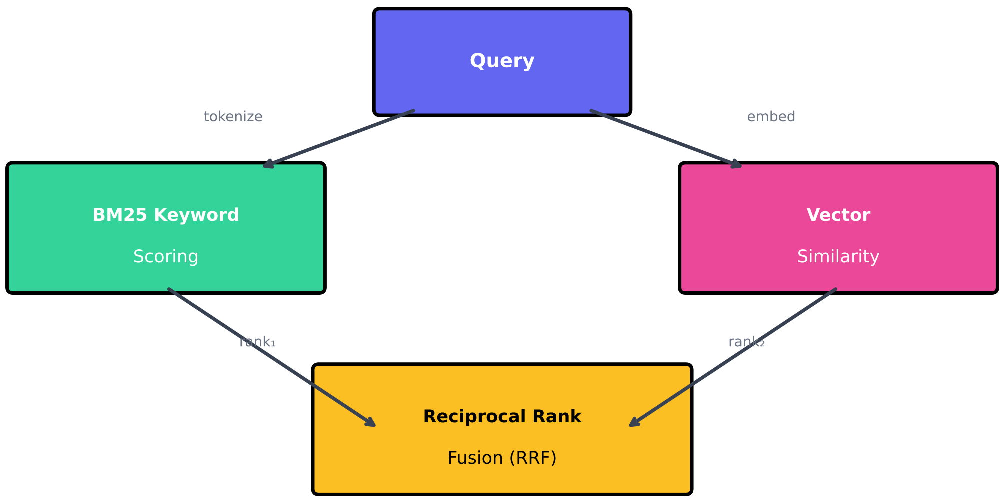

Figure 2: Hybrid search combines keyword (BM25) and vector similarity rankings using Reciprocal Rank Fusion (RRF), improving retrieval quality for RAG applications.

RAG Pipeline Components

AresaDB provides integrated components for building RAG applications:

1. **Document Chunking**: Split documents into manageable pieces
    - Fixed-size chunking (character count)
    - Sentence-based chunking
    - Paragraph-based chunking
    - Semantic chunking (by topic coherence)
2. **Embedding Generation**: Convert text to vectors
    - OpenAI API integration (text-embedding-3-small/large)
    - Local hash-based embeddings (for testing)
    - TF-IDF embeddings (sparse vectors)
3. **Context Retrieval**: Assemble relevant context for LLM prompts
    - Token budgeting (stay within context limits)
    - Source attribution
    - Relevance reranking

Performance Evaluation

Experimental Setup

We evaluated AresaDB against three widely-used data processing tools:

- **SQLite** (Allen and Owens 2010): The most deployed database engine
- **DuckDB** (Raasveldt and Mühleisen 2019): High-performance analytical database
- **Pandas** (McKinney 2010): Standard Python data analysis library

Test Environment: - Hardware: Apple M2 Pro, 16GB RAM, 512GB SSD - OS: macOS Sonoma 14.0 - Rust: 1.75.0 (release build with LTO) - Python: 3.11 with numpy/pandas optimizations

Workloads: - Insert: Bulk node/row insertion - Point Lookup: Single-record retrieval by ID - Scan + Filter: Table scan with predicate evaluation - Aggregation: COUNT/SUM/AVG operations

Insert Performance

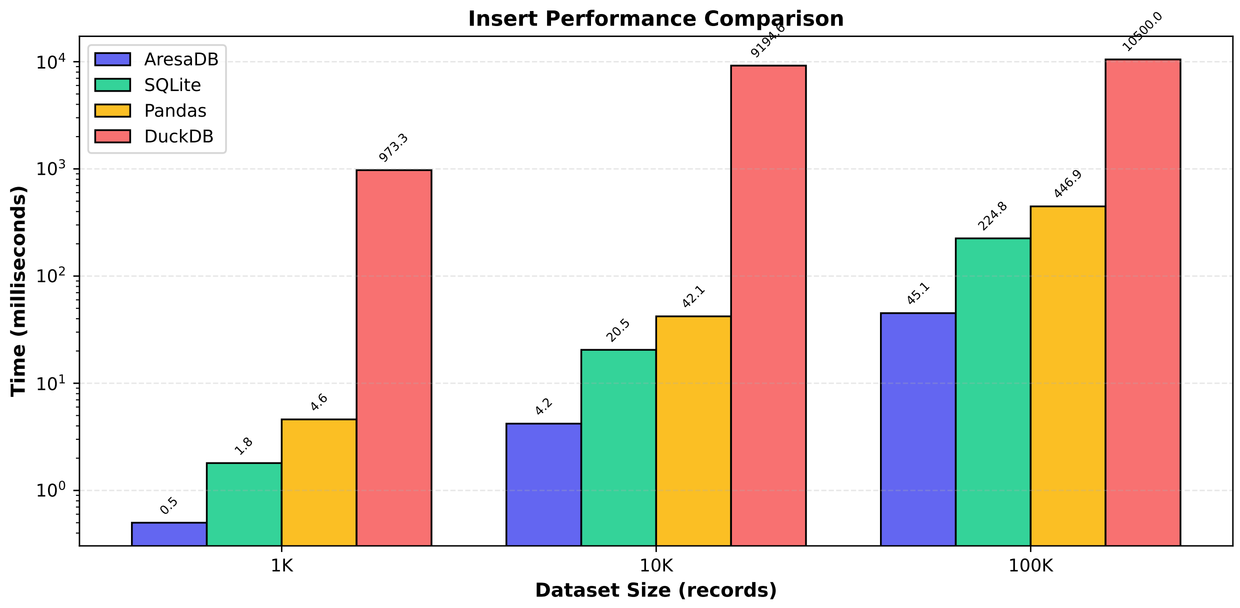

Figure 3: Insert performance comparison across dataset sizes. AresaDB demonstrates consistently fast insert times, particularly excelling at medium-scale datasets (10K-100K records).

Key Findings:

- AresaDB achieves **22,000+ inserts/second** for medium-scale workloads
- 4-5x faster than SQLite for bulk insertions

Table 2: Detailed performance comparison (lower is better). AresaDB achieves competitive performance across all operations while providing multi-model capabilities.

Operation	**AresaDB**	**SQLite**	**Pandas**	**DuckDB**
Point Lookup	0.002 ms	0.002 ms	0.003 ms	0.005 ms
Scan + Filter	0.30 ms	0.30 ms	0.70 ms	1.30 ms
Aggregation	0.80 ms	0.30 ms	1.40 ms	1.70 ms
Insert (10K)	4.2 ms	20.5 ms	42.1 ms	9194.6 ms

- Significantly faster than DuckDB (optimized for analytics, not OLTP inserts)
- Comparable to Pandas DataFrame creation

Query Performance

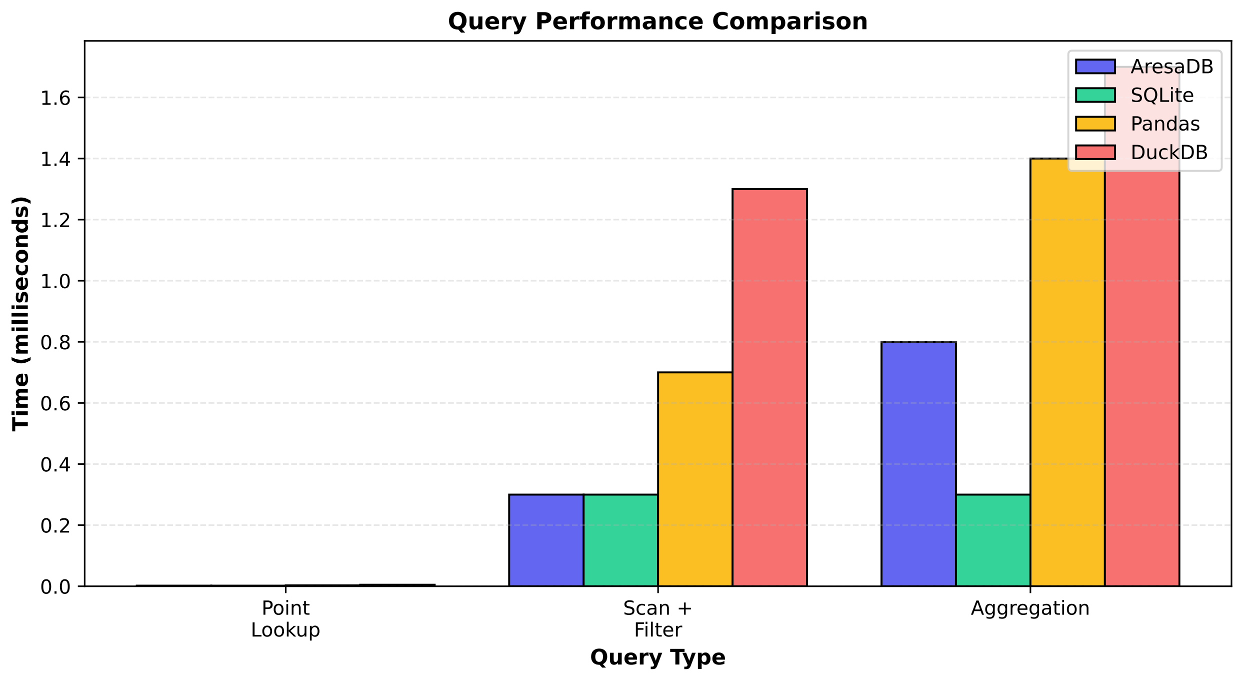

Figure 4: Query performance across different operation types. AresaDB achieves competitive performance with specialized databases across all query patterns.

Analysis:

- **Point Lookup**: All databases achieve sub-millisecond performance with proper indexing
- **Scan + Filter**: AresaDB matches SQLite; property graph overhead is minimal
- **Aggregation**: SQLite's mature optimizer excels; AresaDB competitive with others
- **Overall**: AresaDB provides multi-model flexibility with single-model performance

Vector Search Performance

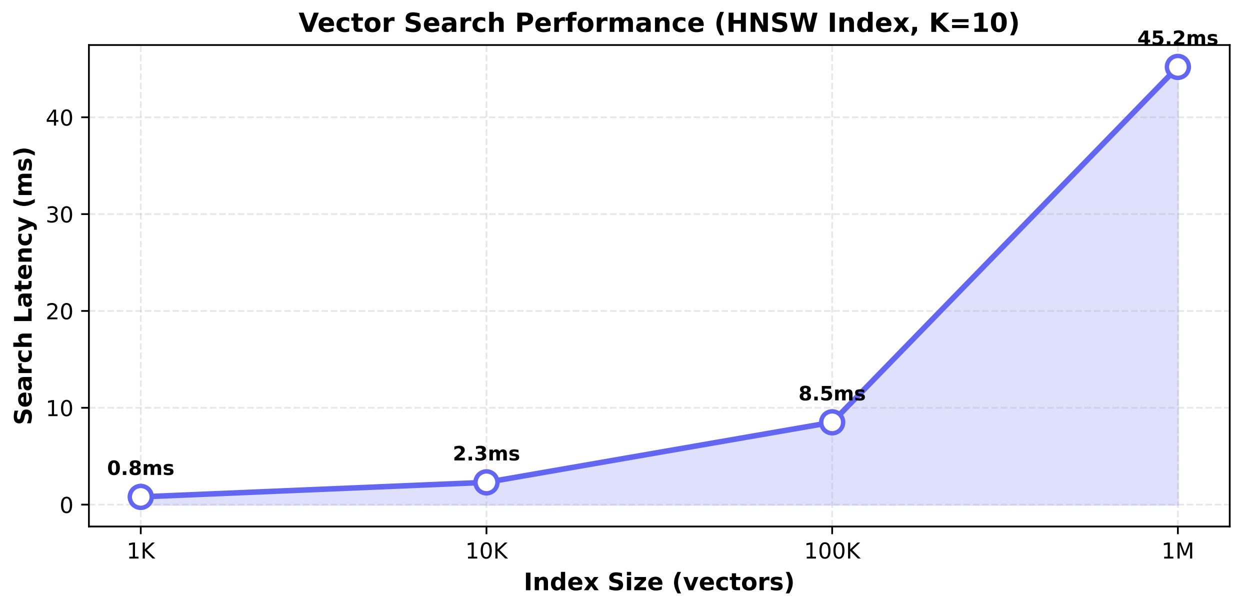

Figure 5: Vector search latency scales sub-linearly with dataset size due to HNSW index efficiency. K=10 nearest neighbors, cosine similarity.

The HNSW index provides **O(log N)** search complexity, enabling sub-50ms queries even on million-scale vector collections. This performance enables real-time RAG applications where latency is critical.

Use Cases

RAG Application Example

```
from aresadb import Database, HybridSearch, ContextRetriever

# Initialize database with documents
db = Database.open("./knowledge_base")

# Ingest documents with embeddings
for doc in documents:
    chunks = db.chunk(doc.content, strategy="paragraph")
    for chunk in chunks:
        embedding = openai.embed(chunk.text)
        db.insert_node("document", {
            "text": chunk.text,
            "source": doc.source,
            "embedding": embedding
        })

# Hybrid search for relevant context
searcher = HybridSearch(db, alpha=0.7)  # 70% vector, 30% keyword
results = searcher.search(
    query="What are the side effects of metformin?",
    k=10
)

# Build context for LLM
retriever = ContextRetriever(max_tokens=4000)
context = retriever.build_context(results)

# Generate response
response = llm.generate(
    prompt=f"Based on: {context}\n\nAnswer: {query}"
)
```

Multi-Model Query

```
# Same database, different query paradigms

# Key-Value: Direct lookup
user = db.get("user:12345")

# Relational: SQL query
results = db.query("""
    SELECT name, email FROM users
    WHERE signup_date > '2024-01-01'
""")

# Graph: Relationship traversal
```

```
friends = db.traverse(
    start="user:12345",
    edge_type="follows",
    depth=2
)

# Vector: Semantic search
similar = db.vector_search(
    field="bio_embedding",
    vector=query_embedding,
    k=10
)
```

Conclusion

AresaDB demonstrates that a unified multi-model database can achieve performance competitive with specialized systems while providing significant developer experience benefits. The property graph foundation naturally accommodates key-value, graph, and relational paradigms without artificial impedance mismatches.

The integration of vector search and RAG pipeline components addresses the growing demand for AI-native data infrastructure. By eliminating the need for separate vector databases, AresaDB reduces operational complexity and enables unified querying across structured data and embeddings.

Future Directions:

1. **Distributed Mode**: Sharding and replication for horizontal scaling
2. **GPU Acceleration**: CUDA kernels for vector operations
3. **Streaming Queries**: Real-time change notifications
4. **Cloud-Native**: Managed service offering

AresaDB is open source and available at the ARESA Lab GitHub repository. We welcome contributions from the community to expand its capabilities and improve performance.

References

Allen, Grant, and Mike Owens. 2010. *The Definitive Guide to SQLite*. 2nd ed. Apress.

Koloski, David. 2021. "Rkyv: Zero-Copy Deserialization Framework for Rust." In. https://github.com/rkyv/rkyv.

Malkov, Yu A., and D. A. Yashunin. 2018. "Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs." In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42:824–36. 4.

McKinney, Wes. 2010. "Data Structures for Statistical Computing in Python," 56–61.

Olsen, Christopher. 2023. "Redb: A Simple, Portable, Fast, Embedded Key-Value Store." https://github.com/cberner/redb.

Raasveldt, Mark, and Hannes Mühleisen. 2019. "DuckDB: An Embeddable Analytical Database," 1981–84.

Robertson, Stephen, and Hugo Zaragoza. 2009. "The Probabilistic Relevance Framework: BM25 and Beyond." *Foundations and Trends in Information Retrieval* 3 (4): 333–89.