Product Requirements Document

Vector + Graph Native Database for Efficient AI Retrieval

Version: 1.0
Date: November 27, 2025
Project: DevForge Hackathon - Problem Statement 2
Author: CodeHashira Team

---

1. Executive Summary

This document outlines the requirements for building a minimal but functional **Vector + Graph Native Database** that supports hybrid retrieval for AI applications. The system combines semantic similarity search through vector embeddings with relational reasoning through graph structures to enable more intelligent and context-aware data retrieval than either approach alone.

1.1 Product Vision

Create a local, high-performance hybrid database that demonstrates how combining vector embeddings with graph relationships produces superior retrieval results for AI applications, including RAG pipelines, knowledge assistants, and enterprise search systems.

1.2 Success Criteria

- Functional hybrid retrieval system running locally
- Real-time query performance (<500ms for typical queries)
- Demonstrated improvement over vector-only or graph-only approaches
- Clean, well-documented API
- Passing score of 35+ in Round 1 Technical Qualifier
- Competitive score in Round 2 Final Demo

---

2. Problem Statement

Modern AI systems require sophisticated retrieval mechanisms for grounding, reasoning, and context assembly. Current solutions face fundamental limitations:

- **Vector databases** excel at semantic similarity but struggle with deep relational queries
- **Graph databases** handle relationships well but lack semantic understanding
- **Real-world applications** need both capabilities simultaneously

2.1 User Needs

AI applications require retrieval systems that can:

- Find semantically similar content across large datasets
- Navigate complex entity relationships
- Answer multi-hop questions requiring reasoning across connections
- Combine similarity and relationship signals for better ranking

2.2 Target Use Cases

- Personal knowledge graphs
- Enterprise RAG (Retrieval-Augmented Generation) pipelines
- Research document management
- Knowledge assistants with relational reasoning
- Context-aware information retrieval

---

3. Core Requirements

3.1 Must-Have Features

3.1.1 Vector Storage & Search

- Store vector embeddings with configurable dimensions
- Implement cosine similarity search
- Support top-k retrieval
- Handle embedding generation or ingestion

3.1.2 Graph Storage & Traversal

- Store nodes with properties and metadata
- Support typed edges with directional relationships
- Enable relationship traversal with depth control
- Maintain edge weights for relationship strength

3.1.3 Hybrid Retrieval

- Merge results from vector similarity and graph adjacency
- Configurable weighting between vector and graph scores
- Unified scoring/ranking mechanism
- Return combined, ranked results

3.1.4 CRUD Operations

Node Operations:

- Create nodes with text, metadata, and embeddings
- Read node details with linked relationships
- Update node properties and regenerate embeddings
- Delete nodes and cascade edge removal

Edge Operations:

- Create relationships between nodes

- Read edge details and properties
- Update edge weights and metadata
- Delete edges

3.1.5 Data Persistence

- Local storage (file-based, SQLite, or in-memory with snapshots)
- Data consistency across restarts
- Efficient indexing for fast retrieval

3.1.6 Embeddings Pipeline

- Integration with open-source embedding models (e.g., Sentence-BERT, MiniLM)
- Support for custom embedding inputs
- Batch embedding generation capability
- Fallback to mocked vectors for testing

3.2 Nice-to-Have Features (Stretch Goals)

- Multi-hop reasoning queries
- Relationship-weighted search algorithms
- Basic schema enforcement
- Pagination and filtering
- Query performance analytics
- Caching layer for frequent queries

---

4. Technical Architecture

4.1 System Components

4.1.1 Storage Layer

- **Vector Store:** In-memory or file-based vector index with cosine similarity search
- **Graph Store:** Adjacency list or graph database structure for nodes and edges
- **Metadata Store:** Key-value store for node properties
- **Persistence Manager:** Handles data snapshots and recovery

4.1.2 Computation Layer

- **Embedding Service:** Generates or accepts vector embeddings
- **Vector Search Engine:** Cosine similarity computation and ranking
- **Graph Traversal Engine:** BFS/DFS for relationship navigation
- **Hybrid Scorer:** Combines vector and graph scores with configurable weights

4.1.3 API Layer

- RESTful endpoints for CRUD and search operations
- Request validation and error handling
- API documentation (OpenAPI/Swagger)

4.1.4 Interface Layer

- CLI tool for queries and data management
- Optional minimal web UI for visualization
- Demo scripts for evaluation

4.2 Technology Stack Recommendations

Backend:

- Language: Python, Node.js, or Go
- Framework: FastAPI, Express, or Gin
- Vector Library: NumPy, Faiss (lightweight mode), or custom implementation
- Graph Library: NetworkX, igraph, or custom adjacency structures

Storage:

- SQLite for structured data
- JSON/Pickle files for vectors
- In-memory caching with periodic snapshots

Embeddings:

- sentence-transformers (Python)
- OpenAI embeddings API (optional)
- Hugging Face models (local)

---

5. API Specification

5.1 Node Operations

POST /nodes

Create a new node with text, metadata, and optional embedding.

Request Body:
{
"text": "string",
"metadata": {
"title": "string",
"tags": ["string"],
"source": "string"
},
"embedding": [0.1, 0.2, ...] // optional
}

Response:
{
"id": "uuid",
"text": "string",
"metadata": {},
"embedding": [...],

```
"created_at": "timestamp"
}
```

GET /nodes/{id}

Retrieve node with properties and linked relationships.

Response:

```
{
"id": "uuid",
"text": "string",
"metadata": {},
"edges": [
{
"edge_id": "uuid",
"target_id": "uuid",
"type": "string",
"weight": 0.8
}
]
}
```

PUT /nodes/{id}

Update node metadata or regenerate embeddings.

Request Body:

```
{
"text": "string",
"metadata": {},
"regenerate_embedding": true
}
```

DELETE /nodes/{id}

Remove node and all associated edges.

Response:

```
{
"deleted": true,
"edges_removed": 5
}
```

5.2 Edge Operations

POST /edges

Create a relationship between nodes.

Request Body:

```
{
"source_id": "uuid",
"target_id": "uuid",
"type": "relates_to",
```

"weight": 0.9,
"metadata": {}
}

Response:
{
"id": "uuid",
"source_id": "uuid",
"target_id": "uuid",
"type": "relates_to",
"weight": 0.9,
"created_at": "timestamp"
}

GET /edges/{id}

Return edge details and properties.

5.3 Search Operations

POST /search/vector

Vector-only search using cosine similarity.

Request Body:
{
"query_text": "string",
"top_k": 10,
"min_score": 0.5
}

Response:
{
"results": [
{
"node_id": "uuid",
"text": "string",
"score": 0.92,
"metadata": {}
}
],
"query_time_ms": 45
}

GET /search/graph

Graph-only traversal from starting node.

Query Parameters:

- `start_id`: Starting node UUID
- `depth`: Maximum traversal depth
- `relationship_type`: Filter by edge type (optional)

Response:
{
"results": [
{
"node_id": "uuid",
"text": "string",
"depth": 2,
"path": ["uuid1", "uuid2", "uuid3"]
}
]
}

POST /search/hybrid

Combined vector + graph search.

Request Body:
{
"query_text": "string",
"vector_weight": 0.6,
"graph_weight": 0.4,
"top_k": 10,
"start_nodes": ["uuid1", "uuid2"], // optional
"max_depth": 2
}

Response:
{
"results": [
{
"node_id": "uuid",
"text": "string",
"vector_score": 0.85,
"graph_score": 0.72,
"combined_score": 0.80,
"metadata": {},
"reasoning": "Found via semantic similarity + 2-hop connection"
}
],
"query_time_ms": 120
}

---

6. Hybrid Scoring Algorithm

6.1 Score Calculation

combined_score = (vector_weight × vector_score) + (graph_weight × graph_score)

Where:

- `vector_score`: Cosine similarity (0 to 1)
- `graph_score`: Normalized graph proximity score

- `vector_weight + graph_weight = 1.0`

6.2 Graph Proximity Scoring

Distance-based:
graph_score = 1 / (1 + shortest_path_distance)

Relationship-weighted:
graph_score = sum(edge_weights_along_path) / path_length

6.3 Result Ranking

1. Calculate individual scores for all candidate nodes
2. Merge results from vector and graph searches
3. Deduplicate nodes
4. Sort by combined score descending
5. Return top-k results

---

7. Data Model

7.1 Node Schema

{
"id": "uuid",
"text": "string", # Original text content
"embedding": [float], # Vector representation
"metadata": {
"title": "string",
"tags": ["string"],
"source": "string",
"created_at": "timestamp",
"updated_at": "timestamp"
},
"edges": ["edge_id"] # References to connected edges
}

7.2 Edge Schema

{
"id": "uuid",
"source_id": "uuid",
"target_id": "uuid",
"type": "string", # e.g., "cites", "related_to", "follows"
"weight": float, # 0.0 to 1.0
"metadata": {},
"created_at": "timestamp"
}

---

8. Demo Requirements

8.1 Dataset

Use a real-world dataset demonstrating practical utility:

- **Option 1:** Research papers with citations (arxiv abstracts)
- **Option 2:** Personal notes with topic connections
- **Option 3:** Wikipedia snippets with hyperlink relationships
- **Option 4:** Project documentation with cross-references

Minimum dataset size: 100+ nodes, 200+ edges

8.2 Demo Scenarios

Scenario 1: Vector-Only vs Hybrid

Query: "machine learning optimization techniques"

- Show vector-only results (pure semantic similarity)
- Show hybrid results (semantic + related papers through citations)
- Demonstrate improved relevance with hybrid approach

Scenario 2: Multi-Hop Reasoning

Query: "Find papers related to papers cited by [specific paper]"

- Demonstrate 2-hop graph traversal
- Show how graph structure enables reasoning
- Compare with vector-only inability to traverse relationships

Scenario 3: Relationship-Aware Ranking

- Show how edge weights influence result ranking
- Demonstrate filtering by relationship type
- Highlight graph proximity scoring

---

9. Evaluation Alignment

9.1 Round 1: Technical Qualifier (50 points)

Core Functionality (20 pts)

- ✓ All CRUD endpoints functional
- ✓ Vector search working with cosine similarity
- ✓ Graph traversal with depth control
- ✓ Data persistence across restarts

Hybrid Retrieval Logic (10 pts)

- ✓ Combined scoring implementation
- ✓ Configurable weight parameters
- ✓ Demonstrable output relevance improvement

API Quality (10 pts)

- ✓ Clean endpoint structure
- ✓ Comprehensive API documentation
- ✓ Proper error handling and validation
- ✓ Clear request/response schemas

Performance & Stability (10 pts)

- ✓ Query response time <500ms for typical datasets
- ✓ No crashes during demo
- ✓ Handles edge cases gracefully

Target: 40+ points to advance comfortably

9.2 Round 2: Final Demo (100 points)

Real-World Demo (30 pts)

- Use-case clarity and practical applicability
- End-to-end workflow demonstration
- Data quality and relevance
- User experience polish

Hybrid Search Effectiveness (25 pts)

- Clear improvement over single-mode search
- Quantitative metrics (precision, relevance)
- Multiple example queries showing benefits
- Explanation of why hybrid works better

System Design Depth (20 pts)

- Architecture justification
- Indexing strategy explanation
- Scoring algorithm rationale
- Scalability considerations
- Trade-off discussions

Code Quality & Maintainability (15 pts)

- Clean, modular code structure
- Meaningful variable and function names
- Appropriate comments and documentation
- Separation of concerns
- Testing coverage (bonus)

Presentation & Storytelling (10 pts)

- Clear problem explanation
- Confident delivery
- User perspective and benefits
- Technical depth balanced with accessibility

Target: 75+ points for competitive positioning

---

10. Development Roadmap

Phase 1: Foundation (Days 1-2)

- Set up project structure and repository
- Implement basic node and edge storage
- Create simple CRUD API endpoints
- Set up persistence layer

Phase 2: Vector Search (Days 3-4)

- Integrate embedding model
- Implement vector storage and indexing
- Build cosine similarity search
- Test with sample data

Phase 3: Graph Capabilities (Days 5-6)

- Implement graph traversal algorithms
- Add relationship-based queries
- Create graph proximity scoring
- Test multi-hop queries

Phase 4: Hybrid Integration (Days 7-8)

- Build hybrid scoring logic
- Implement combined search endpoint
- Fine-tune weighting mechanisms
- Optimize performance

Phase 5: Demo & Polish (Days 9-10)

- Prepare dataset and use cases
- Build CLI/UI interface
- Create API documentation
- Prepare presentation materials
- Conduct internal testing

---

11. Technical Constraints

11.1 System Requirements

- **Must run locally** (no cloud dependencies for core functionality)
- **Real-time performance** (<500ms query latency target)
- **No existing hybrid solutions** (build from scratch)
- **Open source only** (for embedding models and dependencies)

11.2 Resource Constraints

- Memory-efficient for typical datasets (1000+ nodes)
- Disk space <500MB for code and sample data
- CPU-bound operations optimized

11.3 Code Access Requirements

- Repository accessible to OSC and AI/ML Club evaluators
- Read and clone permissions required
- Failure to provide access = disqualification

---

12. Success Metrics

12.1 Functional Metrics

- 100% of required API endpoints implemented
- 100% uptime during demo presentations
- <500ms average query response time
- Zero critical bugs during evaluation

12.2 Quality Metrics

- API documentation completeness: 100%
- Code test coverage: >60%
- Demonstration queries successful: 100%
- Improvement over single-mode: >20% relevance

12.3 Competition Metrics

- Round 1 score: 40+ / 50 (qualifying threshold: 35+)
- Round 2 score: 75+ / 100 (target for top 3)

---

13. Risk Assessment

13.1 Technical Risks

Risk	Impact	Mitigation
Embedding model integration complexity	High	Use simple models like MiniLM; fallback to mock vectors
Query performance issues	High	Implement caching; optimize indexing early
Hybrid scoring produces poor results	High	Test multiple algorithms; adjust weights dynamically
Data persistence bugs	Medium	Thorough testing; use proven libraries (SQLite)

13.2 Timeline Risks

Risk	Impact	Mitigation
Scope creep with stretch goals	Medium	Focus on core requirements first; add stretch goals if time permits
Integration delays	Medium	Build modular components; use interfaces for swappable parts
Demo preparation time shortage	High	Parallel development of demo alongside core features

---

14. Deliverables Checklist

14.1 Code Deliverables

- [ ] Backend service (fully functional)
- [ ] All required API endpoints
- [ ] Data persistence implementation
- [ ] Embedding pipeline integration
- [ ] Hybrid scoring algorithm

14.2 Documentation Deliverables

- [ ] API documentation (OpenAPI/Swagger)
- [ ] README with setup instructions
- [ ] Architecture diagram
- [ ] Algorithm explanations
- [ ] Dataset description

14.3 Demo Deliverables

- [ ] CLI or minimal UI for queries
- [ ] Sample dataset loaded
- [ ] Demo script/walkthrough
- [ ] Comparison visualizations
- [ ] Presentation slides

14.4 Repository Requirements

- [ ] Clean commit history
- [ ] Proper .gitignore
- [ ] Dependencies listed (requirements.txt/package.json)
- [ ] Access granted to evaluators
- [ ] License file (if applicable)

---

15. Appendix

15.1 Reference Resources

Embedding Models:

- Sentence-BERT: https://www.sbert.net/
- all-MiniLM-L6-v2 (lightweight, fast)
- Hugging Face Transformers

Graph Algorithms:

- NetworkX documentation
- BFS/DFS implementations
- Shortest path algorithms

Vector Search:

- Cosine similarity computation
- Approximate nearest neighbors (ANN)

API Design:

- REST API best practices
- OpenAPI specification

15.2 Example Queries

Vector Search:
curl -X POST http://localhost:8000/search/vector
-H "Content-Type: application/json"
-d '{"query_text": "neural networks", "top_k": 5}'

Hybrid Search:
curl -X POST http://localhost:8000/search/hybrid
-H "Content-Type: application/json"
-d '{

"query_text": "deep learning optimization",
"vector_weight": 0.6,
"graph_weight": 0.4,
"top_k": 10
}'

15.3 Glossary

- **Vector Embedding:** Numerical representation of text in high-dimensional space
- **Cosine Similarity:** Measure of similarity between two vectors (range: -1 to 1)
- **Graph Adjacency:** Direct connections between nodes
- **Multi-hop Query:** Query requiring traversal through multiple edges
- **RAG:** Retrieval-Augmented Generation (AI technique using external knowledge)
- **Hybrid Retrieval:** Combining vector similarity and graph relationships
- **Top-K:** Returning the top K highest-scoring results

---

16. Team Communication Plan

16.1 Repository Structure

```
/
├── src/
│ ├── api/ # API endpoints
│ ├── core/ # Core logic (vector, graph, hybrid)
│ ├── storage/ # Persistence layer
│ └── models/ # Data models
├── tests/ # Unit and integration tests
├── docs/ # Documentation
├── data/ # Sample datasets
├── demo/ # Demo scripts and UI
└── README.md
```

16.2 Development Workflow

- Daily standups (async updates)
- Feature branches with PR reviews
- Continuous integration for testing
- Demo dry-runs before evaluation

Document Status: Draft for Team Review
Next Review: Before implementation kickoff
Approval Required: Team Lead + Technical Architect