Vector + Graph Native Database for Efficient AI Retrieval

System Architecture, Libraries & API Documentation

1. System Overview

- This document describes the architecture, libraries, data flow, and API surface of a fully local Vector + Graph Native Retrieval System.
- The system ingests unstructured data (free text, HTML, JSON), stores it as nodes in a SQLite-backed knowledge graph, generates lightweight hash-based embeddings, and exposes search and graph traversal capabilities through a FastAPI backend.
- A separate Streamlit UI can be used to interactively test the system.

.

2. Libraries & Their Roles

Category	Library	Usage / Purpose
Web Framework	FastAPI	Defines REST endpoints for nodes, edges, and search; performs request/respo
Web Server	Uvicorn	ASGI server used to run the FastAPI application locally.
Validation	Pydantic	Defines request/response models (NodeCreate, NodeUpdate, EdgeCreate sea
Database	sqlite3 (stdlib)	Provides a lightweight, file-based database for nodes, edges, and embeddings
Numerics	NumPy	Implements fixed-size hash-based embeddings and cosine-style similarity scor
Keyword Search	rank-bm25	Provides BM25 keyword scoring over node texts for lexical search.
Collections	collections (deque, defaultdi	ct)Used to implement BFS traversal and adjacency lists in the graph service.
Time & JSON	datetime, json	Timestamps for auditing and JSON serialization of metadata.
Web UI (optional)	Streamlit	Provides an interactive UI to upload data, create nodes, and issue search quer
HTTP Client (opti	nal)requests	Used by the UI to call the FastAPI backend endpoints.
Tabular View (opti	ɔnal)pandas	Helps display search results in tables inside the Streamlit UI.

3. End-to-End Pipeline (Text Diagram)

```
[Client / Judge / UI]
| | HTTP (JSON) v +----------
-----------------+
| FastAPI API |
| - /nodes CRUD |
| - /edges CRUD |
| - /search/vector |
| - /search/graph |
| - /search/hybrid |
+-------------+-------------+
| | interacts
with v +--------
----------------
---+
| DatabaseManager |
| - sqlite3 connection |
| - tables: nodes, edges, |
| embeddings |
+------+------+-------------+
| |
| +------------------------+ | | v v +-
-------------+ +--------------------+
| Embedding | | BM25SearchService |
| Service | | - BM25Okapi index |
| - hash-based | | - token-based |
| vectors | | scoring |
+--------------+ +--------------------+
| |
| embeddings |
+---------------+---------------+
| v +-------------------+
| VectorSearch |
| Service |
| - scans stored |
| embeddings |
| - computes |
| similarity |
+---------+--------+
| v +---------------
---+
| GraphService |
| - adjacency list |
| - BFS traversal |
| - graph scoring |
+---------+--------+ |
v +--------------------
-+
| HybridSearchService |
| - vector ranking |
| - BM25 ranking |
| - graph proximity |
| - RRF fusion |
+----------+----------+
| v [Ranked search
results]
```

4. Data Flow: Step-by-Step

Here is the documentation formatted as a professional technical document.

- **1. Ingestion: A client (judge, Postman, or Streamlit UI) sends a POST/nodes request with text, metadata, and auto_embed flag.**

- **The FastAPI layer validates the payload via Pydantic models.**
- **2. Storage: Database Manager writes the node into the SQLite 'nodes' table.**
- **If auto_embed is true, the EmbeddingService generates a hash-based embedding and stores it in the 'embeddings' table.**
- **3. Indexing: VectorSearchService can fetch all embeddings from the DB for vector similarity searches.**
- **BM25Search Service rebuilds a BM25 index over all node texts to support keyword search.**
- **4. Graph Construction: POST/edges creates edges between existing nodes.**
- **GraphService reads the 'edges' table to build an adjacency list for BFS.**
- **5. Retrieval Modes:**
- **/search/vector: encodes the query using the same hash-based embedding scheme and scans all stored vectors.**
- **/search/graph: performs BFS from a start node up to a given depth and returns reachable nodes and edges.**
- **/search/hybrid: runs vector search and BM25 search, fuses rankings via Reciprocal Rank Fusion (RRF), computes graph proximity scores, and combines them using weighted final_score = vector_weight* text_score + graph_weight graph_score.**
- **6. Response: FastAPI returns JSON responses containing node data, scores, and graph context.**

- **The Streamlit UI or judges' test harness consume these APIs to evaluate correctness.**
- **5. API Surface Summary**
- **Nodes:**
- **Core endpoints implemented and tested: Nodes: POST /nodes - Create a node with text, metadata, and optional embedding.**
- **GET/nodes/{id} - Retrieve node details, metadata, embedding status, and relationships.**
- **PUT/nodes/{id} - Update text/metadata and optionally regenerate embeddings.**
- **DELETE /nodes/{id}- Remove a node and its associated embeddings and edges.**
- **Edges:**
- **POST/edges - Create a relationship between two existing nodes.**
- **GET /edges/{id} - Retrieve a specific edge and its properties.**
- **Search:**
- **POST /search/vector - Semantic-style search over hash-based embeddings.**
- **GET /search/graph - BFS graph traversal from a start node up to a given depth.**
- **POST /search/hybrid - Hybrid ranking combining vector, keyword (BM25), and graph proximity.**

- **Utility:**

- **GET / Health check for the service.**

- **GET /stats Returns total node, edge, and embedding counts.**

- **6. Design Advantages & Uniqueness**

- **Fully Local & Deterministic: The system does not depend on external LLMs or remote vector databases.**

- **All computation is done locally using standard Python libraries.**

- **Resilient Embedding Strategy: A custom hash-based embedding scheme avoids heavy ML dependencies and model downloads, while still enabling meaningful similarity search.**

- **True Graph + Vector Hybrid: The architecture natively combines graph traversal (BFS-based proximity) with both semantic-style and keyword-style retrieval, going beyond simple vector-only RAG.**

- **Flexible Data Ingestion: Nodes can store any unstructured text (raw text, HTML, JSON serialized as strings) with arbitrary JSON metadata, allowing many upstream formats to be normalized into a single retrieval engine.**

- **Clean API Contract: The CRUD and search endpoints match the required problem-statement contract, return correct HTTP 200 codes, and are easy to test with Postman or automated judges.**

- **Extensible Frontend: A Streamlit UI (optional) can sit on top of the same APIs, demonstrating how applications and judges can consume the retrieval engine without modifying backend logic.**

- **7. Constraint Compliance**

- **This implementation complies with the hackathon constraints by: Avoiding external LLMs or proprietary embedding APIs.**

- **Running completely on local infrastructure (Python + SQLite).**

- **Implementing all required endpoints for node, edge, and search operations.**

- **Supporting unstructured input (free text, HTML, JSON) via a unified node model.**

- **Demonstrating vector, keyword, graph, and hybrid retrieval modes with clear separation of concerns.**