Serverless Knowledge Graph Architecture for Low-Resource Environments: Integrating Structured YAML/JSON Repositories with Health Informatics

Executive Summary

The convergence of educational ontologies and high-frequency biometric data presents a unique architectural challenge when deployed in computationally constrained environments. This report addresses the specific engineering requirements of integrating the "Baby Brains Montessori" curriculum—comprising over 146 YAML-based structured files—with high-volume health and fitness protocols derived from Garmin wearable devices. The defining constraint of this architecture is the unavailability of sufficient Random Access Memory (RAM) to support dedicated, resident graph database management systems (DBMS) such as Memgraph or Neo4j. Consequently, the system must adopt a "serverless" or embedded graph architecture that leverages the file system as the primary storage engine while maintaining the traversal capabilities inherent to knowledge graphs.

This research establishes a comprehensive technical blueprint for a file-based, disk-optimized knowledge graph. Unlike traditional approaches that prioritize in-memory adjacency matrices for microsecond latency, this architecture prioritizes memory efficiency and interoperability through the use of Operating System-level memory mapping (mmap) and offset-based indexing. The analysis proceeds through five critical dimensions: the optimization of file-based graph representations using JSON Lines (JSONL) and byte-level indexing; the implementation of lazy-loading traversal algorithms (BFS/DFS) in Python to bypass the need for Cypher; the integration of disk-based vector search via LanceDB for hybrid semantic-structural querying; the design of a domain-specific schema compliant with IEEE 1752.1 and Open mHealth standards; and a strategic migration path to enterprise-grade systems.

The findings indicate that by decoupling data storage from the compute runtime and employing lazy evaluation strategies, it is possible to replicate the functionality of a property graph database on commodity hardware with minimal RAM. Furthermore, the adoption of strict schema standards today ensures that the data remains portable, enabling a seamless transition to high-performance graph engines should infrastructure constraints ease in the future.

1. Architectural Constraints and the File-Based Paradigm

The design of a knowledge graph (KG) is typically predicated on the assumption of abundance—specifically, the abundance of Random Access Memory (RAM). Graph databases like Neo4j and Memgraph are engineered to optimize traversal speed by loading index-free adjacency lists or pointer-heavy relationship structures entirely into memory. In these systems, traversing an edge is a simple memory pointer dereference, an operation measured in nanoseconds. However, when the hosting environment lacks the RAM to hold the graph topology and its associated properties, these systems performance degrades precipitously due to excessive paging or simply fails to initialize.

1.1 The Memory Hierarchy and the "RAM Wall"

In the context of the Baby Brains Montessori and Health/Fitness integration, the data landscape is bifurcated. The Montessori dataset, consisting of 146+ YAML files, is relatively small in terms of storage bytes but rich in topological complexity (prerequisites, dependencies, categorical relationships). In contrast, the Health/Fitness data, derived from high-frequency Garmin FIT protocols, is voluminous, temporal, and continuously growing. Attempting to load this combined dataset into a Python dictionary or a standard NetworkX graph object would immediately hit the "RAM Wall," triggering MemoryError exceptions or forcing the operating system to swap active memory to disk, which introduces latency orders of magnitude higher than RAM access.

The core architectural requirement, therefore, is to shift the "center of gravity" of the data from the application heap to the disk, specifically leveraging the file system as the database engine. This approach aligns with the "out-of-core" processing paradigm, where algorithms are designed to process data that exceeds available main memory.[1]

1.2 Limitations of Native YAML/JSON Parsers

The naive approach to file-based graphs involves reading data directly from human-readable formats like YAML or JSON. While these formats are excellent for the "Baby Brains" curriculum designers—allowing for easy editing and version control—they are ill-suited for random access retrieval.
Standard Python libraries such as yaml.safe_load or json.load function as eager parsers.2 They read the entire file stream, parse the syntax, and construct a complete Python object graph in memory before returning control to the user. For a single 2GB JSON file containing health records, this operation requires reading the entire 2GB from disk, allocating Python objects for every dictionary and list (which incurs significant overhead per object), and potentially consuming 10GB+ of RAM during the parse phase.

Furthermore, these formats enforce sequential access. To access the last record in a standard JSON array, the parser must process every preceding byte to ensure syntactic validity (matching braces, commas, etc.). This $O(N)$ access time complexity makes graph traversal—which relies on rapid, non-sequential hopping between nodes—computationally infeasible. A query checking 5 levels of prerequisites would require scanning the entire dataset 5 times.

1.3 The Solution: Memory Mapping (mmap)

To circumvent the limitations of eager parsing and sequential I/O, the architecture leverages the mmap system call. Memory mapping allows the application to map a file descriptor directly into the process's virtual memory address space.[3] This creates the illusion that the entire file is loaded into RAM, while the Operating System (OS) kernel manages the actual data loading behind the scenes.

- **Virtual Memory Abstraction:** The OS divides the file into "pages" (typically 4KB). When the Python code accesses a specific byte offset in the mapped region (e.g., data[1024:2048]), the OS triggers a "page fault" if that page is not currently in physical RAM. The kernel then fetches only that specific page from the disk.
- **Lazy I/O:** This mechanism essentially provides "lazy I/O." The application effectively random-accesses the disk with the syntax of array slicing. If a graph traversal only touches 5% of the nodes, only those 5% of the file pages are ever loaded from the disk.[4]
- **Zero-Copy Access:** For binary formats, mmap allows zero-copy access where the data is read directly from the kernel's page cache without being copied into a user-space buffer. For text formats like JSON, deserialization is still required, but the I/O overhead is drastically reduced.[3]

1.4 Trade-offs of Serverless Graphs

Adopting this file-based, serverless approach entails specific trade-offs compared to running a dedicated DBMS:

- **Concurrency:** A file-based graph is primarily optimized for read-heavy workloads. Concurrent writes (e.g., updating health stats while querying) require careful file locking (using fcntl or similar) to prevent corruption, whereas a DBMS handles ACID transactions automatically.[6]
- **Query Language:** Without a DBMS, there is no Cypher or SQL engine. All query logic (filtering, joining, traversing) must be implemented in the application code (Python), effectively moving the "query planner" logic into the developer's domain.
- **Indexing:** Indices must be manually built and maintained. Unlike Neo4j, which automatically updates indices upon node insertion, a file-based architecture requires explicit index regeneration or append-only log structures.

Despite these trade-offs, the serverless approach provides the only viable path for running complex knowledge graph operations on hardware that cannot support the JVM (for Neo4j)

or the memory requirements of in-memory engines.

---

2. Optimal File-Based Graph Representation

To enable efficient graph operations on disk, the raw data—specifically the heterogeneous mix of Montessori YAML files and Garmin JSON dumps—must be transformed into a structure that supports $O(1)$ random access. The recommended structure combines **JSON Lines (JSONL)** for data storage with a **binary byte-offset index** for retrieval.

2.1 The JSON Lines (JSONL) Format

JSON Lines (also known as NDJSON) stores a single graph entity (Node or Edge) on each line of the file. This format offers a critical advantage over monolithic JSON: **line independence**.

- **Structure:** Each line is a valid, independent JSON object. There are no enclosing brackets `` or separating commas between lines.
- **Append-Only capability:** New health records can be appended to the end of the file without rewriting or parsing the existing data, a crucial feature for logging high-frequency fitness data.[7]
- **Stream Processing:** The file can be processed line-by-line using a generator, keeping memory usage constant regardless of file size.

Transformation Pipeline:
The Montessori YAML files act as the "Source of Truth" for human editing. A build process should traverse these files and compile them into a single montessori_graph.jsonl file. Similarly, processed Garmin FIT data is appended to health_data.jsonl.
Example JSONL Structure:

JSON


```
{"id": "act_001", "type": "Activity", "label": "Pouring Water", "props": {"age": "18m"}, "edges": ["skill_05", "skill_09"]}
{"id": "skill_05", "type": "Skill", "label": "Pincer Grasp", "props": {"domain": "motor"}, "edges":}
```


2.2 The Sidecar Byte-Offset Index

While JSONL allows line-by-line reading, finding a specific node (e.g., "act_001") still requires a sequential scan ($O(N)$). To achieve the $O(1)$ lookups required for graph traversal, we must implement a **Sidecar Index**.

This index maps a unique Node ID to its specific **Byte Offset** and **Byte Length** in the JSONL file.

2.2.1 Index Structure

The index is a lightweight lookup table. For a graph with 1 million nodes, this index might be 20-30MB, which easily fits into even very constrained RAM, or can itself be stored on disk using a B-Tree structure (like SQLite or dbm).

Node ID (Key)	Offset (Bytes)	Length (Bytes)
act_001	0	128
skill_05	129	94

2.2.2 Building the Index

The index is built during the data ingestion phase. As the builder writes the JSONL file, it records the file pointer position (file.tell()) before writing each line.

1. Open graph.jsonl for writing.
2. Open graph.idx (or a SQLite index.db) for storing the map.
3. For each entity:
    - Get current offset: pos = f_jsonl.tell()
    - Write JSON line.
    - Get length: length = f_jsonl.tell() - pos
    - Store (id, pos, length) in the index.

2.2.3 Retrieval Logic

To retrieve "skill_05":

1. Query the Index for "skill_05" $\rightarrow$ Returns Offset: 129, Length: 94.
2. Seek to the offset: f_jsonl.seek(129).
3. Read exactly *Length* bytes: data = f_jsonl.read(94).
4. Deserialize: node = json.loads(data).

This operation requires exactly one disk seek and one read, fulfilling the performance requirement for graph traversals.[8]

2.3 Alternative Storage Considerations

While SQLite is a robust option, the JSONL+Index approach is often superior for *graph* workloads in Python for specific reasons:

- **Serialization Overhead:** Storing arbitrary, schema-less properties (common in Montessori and Health data) in SQLite requires either Entity-Attribute-Value (EAV) tables (slow joins) or storing JSON blobs in a text column. If storing JSON blobs, SQLite adds its own page-management overhead.
- **Direct Mapping:** The JSONL approach maps 1:1 with the Python dictionary structure used during traversal.
- **Compression:** JSONL files can be compressed (e.g., gzip), and with blocked compression formats (like bgzf), random access is still possible, though complex. For this low-resource constraint, uncompressed text on SSD is usually preferred for speed, or IndexedGzipFile if space is tight.[10]

2.4 Optimizing Adjacency Lists

To facilitate traversal, the "Edges" should be stored *within* the Node object in the JSONL file (Adjacency List pattern).

- **Directed vs. Undirected:** Since JSONL is sequential, storing edges on both the Source and Target nodes (doubling the storage) enables bidirectional traversal without a separate edge index.
- **Edge Properties:** If edges have properties (e.g., "Confidence Score" for health data), the adjacency list should store objects, not just IDs.
    - *Simple:* "edges": ["node_b", "node_c"]
    - *Property Graph:* "edges":

This denormalization increases file size but drastically reduces seek time during traversal, as fetching a node also fetches its immediate topology.[11]

---

3. Query Patterns Without Cypher: Algorithmic Implementation

In the absence of a query engine like Cypher, the application logic must implement graph algorithms directly. To respect the memory constraints, these algorithms must utilize **Lazy Evaluation** and **Generators**.

3.1 Lazy Loading and The Proxy Pattern

Standard Python graph libraries (like NetworkX) often load the entire graph structure into memory. For the Health/Fitness integration, where nodes could represent thousands of daily heart rate readings, this is impossible. We must implement a custom Graph class that uses a **Proxy Pattern**.

3.1.1 The Node Proxy

Instead of a Node object containing a list of full Node objects as neighbors, it contains a list of **Node IDs**. The neighbor objects are only instantiated when iterated over.

Python

```
class LazyNode:
    def __init__(self, node_id, data_loader):
        self.id = node_id
        self._loader = data_loader
        self._data_cache = None

    @property
    def data(self):
        if self._data_cache is None:
            # Only loads from disk when accessed
            self._data_cache = self._loader.fetch(self.id)
        return self._data_cache

    @property
    def neighbors(self):
        # Generator: yields neighbors one by one
        neighbor_ids = self.data['edges']
        for nid in neighbor_ids:
            yield LazyNode(nid, self._loader)
```

This ensures that at any point in a traversal, only the nodes in the current path or "frontier" are held in memory. As the traversal moves on, Python's reference counting garbage collector frees the memory of visited nodes.[13]

3.2 Breadth-First Search (BFS) Implementation

BFS is the primary algorithm for "Reachability" (e.g., "Is the child ready for this activity?") and "Shortest Path" (e.g., "What is the quickest path to learning division?").

Memory-Efficient BFS Logic:

1. **Queue:** Use collections.deque for $O(1)$ pops.
2. **Visited Set:** Maintain a set of visited IDs (strings). A set of strings is memory efficient (millions of IDs fit in MBs). *Do not store Node objects in the visited set.*
3. **Generator Pipeline:** The BFS function should yield nodes as it finds them, allowing the consumer to stop the search early (e.g., "Find the *first* prerequisite not met").

Implementation Nuance: For multi-hop queries (e.g., "Find all nodes within 3 hops"), the queue should store tuples: (node_id, depth). When depth exceeds the limit, stop expanding that branch. This effectively prunes the search tree without loading unnecessary data.[15]

3.3 Depth-First Search (DFS) and Recursion Limits

DFS is useful for exploring complete prerequisite chains. However, Python has a strict recursion limit (default 1000). For deep graphs (like a multi-year curriculum dependency tree), a recursive DFS will crash.
Solution: Implement Iterative DFS using an explicit stack (Python list).

- **Cycle Detection:** Critical in file-based graphs where you cannot "see" the whole structure. Maintain a path_stack set (nodes currently in the recursion stack) to detect back-edges. If a neighbor is in path_stack, a cycle exists.

3.4 The "NoGraphs" Library

Rather than re-inventing these algorithms, the **NoGraphs** library is explicitly designed for this "lazy graph" use case. It allows you to define a graph by a function next_nodes(node).

- **Integration:** You simply pass your LazyNode.neighbors generator to NoGraphs.
- **Benefits:** It implements BFS, DFS, Dijkstra, A*, and MST traversals that are completely lazy. It creates the internal bookkeeping structures (visited sets, heaps) efficiently.
- **Recommendation:** Use NoGraphs to handle the algorithmic complexity (e.g., bidirectional search, path reconstruction) while your code focuses on the JSONL reading optimization.[14]

3.5 Handling Bidirectional Queries on Directed Storage

The JSONL adjacency list is typically directed (Source $\rightarrow$ Target). However, queries often require reverse lookups (e.g., "What activities *require* this skill?").

- **Index-Time Inversion:** When building the JSONL file, you can build a secondary "Reverse Index" file.
- **Double-Entry:** For every relationship $A \rightarrow B$, write an entry in A's record ({"edges_out":}) and update B's record ({"edges_in": ["A"]}).
- **Runtime Implication:** This requires the build process to be 2-pass (or store the reverse map in memory during build), but it makes runtime traversal strictly $O(1)$ in both directions.

---

4. Hybrid Intelligence: Integrating Vector Search

The user requirement includes "Hybrid Vector + Graph Search." This capability is essential for queries that combine semantic intent with structural constraints, such as "Find activities

similar to 'Pouring' (Vector) that are suitable for a child who has mastered 'Grasping' (Graph)."

4.1 The Role of Vector Embeddings

In this architecture, text descriptions of Montessori activities and Health protocols are converted into high-dimensional vectors (embeddings) using models like all-MiniLM-L6-v2 or OpenAI's embeddings. These vectors capture the *meaning* of the content.

- *Example:* "Pouring Water" and "Transferring Beans" are semantically close in vector space (both are "transferring" tasks), even if they are not explicitly linked in the graph topology.

4.2 Vector Database Selection: LanceDB

Given the "Cannot run Memgraph/Neo4j due to RAM" constraint, typical vector stores like Milvus or Weaviate are likely too heavy. **LanceDB** is the optimal choice for this persona.

- **Disk-Based Architecture:** LanceDB is built on the **Lance** file format, a columnar format designed for AI workloads. It uses an **IVF-PQ** (Inverted File with Product Quantization) index that resides on disk. It does not require loading the entire index into RAM, unlike HNSW-based stores (like ChromaDB or Faiss in default modes).[16]
- **Zero-Copy:** It utilizes Apache Arrow for data interchange, allowing for extremely fast, zero-copy reads from disk to Python memory.
- **Embedded:** LanceDB runs in-process (like SQLite), requiring no separate server.

4.3 The "Graph-Based Vector Search" Pattern

To integrate the Vector Search with the Knowledge Graph, we utilize a **Foreign Key** pattern.

4.3.1 Schema Integration

In LanceDB, the schema for the vector table should include the node_id from the file-based graph.

Python

```
# LanceDB Schema
import pyarrow as pa
schema = pa.schema()
```

4.3.2 Query Execution Workflow

The hybrid search is executed as a multi-stage pipeline:

1. **Stage 1: Semantic Retrieval (LanceDB)**
    - Query: "Activities for fine motor control."
    - LanceDB scans the disk-based index and returns the Top-K candidates (e.g., 50 activities).
    - Output: A list of node_ids.
2. **Stage 2: Structural Filtering (Graph)**
    - For each candidate node_id, the system uses the **Offset Index** to load the LazyNode from the JSONL file.
    - **Graph Check:** A BFS traversal checks the prerequisites subgraph for each candidate.
    - *Condition:* "Is the child's 'completed_skills' set a superset of the candidate's 'prerequisite' set?"
3. **Stage 3: Ranking and Result**
    - The system filters out candidates where the structural check fails (i.e., the child is not ready for the activity).
    - The remaining valid nodes are returned to the user, ranked by their vector similarity score.

This architecture ensures that recommendations are both *relevant* (Semantic) and *actionable* (Structural).[18]

---

5. Domain-Specific Schema Design: Health & Education

A Knowledge Graph is only as good as its schema. For this project, the schema must bridge the gap between the structured pedagogical data of Montessori and the quantitative physiological data from Garmin. To ensure validity and interoperability, the health portion of the schema must adhere to **IEEE 1752.1** and **Open mHealth** standards.

5.1 The Montessori Ontology

The Montessori curriculum data (from the 146 YAML files) represents the "Knowledge" layer of the graph.

- **Node Types:**
    - Activity: A specific task (e.g., "Scrubbing a Table").
    - Skill: A developmental milestone (e.g., "Crossing the Midline").
    - Material: Physical objects used (e.g., "Scrubbing Brush").
    - Area: Curriculum area (e.g., "Practical Life", "Sensorial").
- **Relationships:**
    - (:Activity)-->(:Skill)
    - (:Activity)-->(:Material)

- (:Activity)-->(:Area)
- (:Skill)-->(:Skill) (Developmental progression)

5.2 The Health & Fitness Ontology (IEEE 1752.1)

The Garmin data represents the "Observation" layer. Instead of creating a custom schema, the system maps Garmin data to the **IEEE 1752.1** standard. This standard defines JSON schemas for metadata, sleep, and physical activity.[20]

5.2.1 Garmin FIT to IEEE 1752.1 Mapping

Garmin devices generate .FIT files containing binary messages. The fitdecode library is essential for extracting this data without data loss. Crucially, fitdecode supports "chained" FIT files (where multiple sessions are concatenated) and does not discard header/footer data, ensuring provenance is maintained.[22]

Mapping Strategy:

Garmin FIT Message	Field	IEEE 1752.1 / OpenmHealth Schema	Graph Property (JSONL Body)
Record	heart_rate	omh:heart-rate	{"type": "HeartRate", "value": 120, "unit": "bpm"}
Record	steps	omh:step-count (or physical-activity)	{"type": "StepCount", "value": 15, "unit": "steps"}
Session	total_calories	omh:calories-burned	{"type": "Calories", "value": 450, "unit": "kcal"}
Session	timestamp	effective_time_frame	{"timestamp": "2023-10-27T08:00:00Z"}
DeviceInfo	manufacturer	header.acquisition_provenance	{"source": "Garmin Vivosmart 5"}

Graph Node Structure (Health Observation):

JSON

```
{
  "id": "obs_hr_20231027_0800",
  "type": "HealthObservation",
  "schema": "ieee:heart-rate:1.0",
  "payload": {
    "heart_rate": {"value": 75, "unit": "beats/min"},
    "effective_time_frame": {"date_time": "2023-10-27T08:00:00Z"}
  },
  "edges":
}
```

5.3 Representing Evidence and Confidence

In the health domain, data often carries a degree of uncertainty (e.g., "Smartwatch detected stress with 80% confidence"). The Property Graph model excels here by allowing properties on edges.

- **Pattern:** (:HealthObservation)-->(:HealthState)
- **JSONL Representation:**
  JSON
  "edges":

This allows traversal algorithms to filter edges based on confidence thresholds (e.g., "Only consider indicators with confidence > 0.7").[24]

5.4 Linking the Domains

The power of this Knowledge Graph lies in the intersection.

- **Cross-Domain Query:** "Does 'High Stress' (Health) impact 'Fine Motor' performance (Montessori)?"
- **Linkage:**
  - (:HealthState {label: "Stress"})-->(:Skill {domain: "Fine Motor"})
  - This relationship might be inferred from literature or user logs.
  - Query Logic: If User -> HAS_STATE -> Stress, then User effectively has a blocker on Skill. The BFS algorithm can take this blocker into account when calculating "Available

Activities."

---

6. Migration Path to Memgraph/Neo4j

While the file-based architecture solves the immediate resource constraints, it should be viewed as a "bootstrap" phase. As the dataset grows or if hardware becomes available (e.g., migrating to a cloud instance), the system should be able to transition to a dedicated Graph DBMS like Memgraph or Neo4j without data re-engineering.

6.1 Data Isomorphism Strategy

The **JSONL** format chosen for the file-based graph is highly compatible with the import tools of enterprise graph databases.

- **Neo4j:** The APOC library (apoc.load.json) can ingest JSONL lines directly.
- **Memgraph:** Memgraph provides the json_util module for loading JSON data.

Design for Migration:

1. **Strict Typing:** Ensure every node in the JSONL file has a "labels" (list) and "properties" (dict) field. This maps 1:1 to the Property Graph data model.
2. **ID Consistency:** Use UUIDs or strict naming conventions for IDs in the JSONL file. These will become the internal IDs or indexed property IDs in the DBMS.

6.2 The Migration Pipeline

When ready to migrate, the process is:

1. **Export:** No export needed; the runtime graph.jsonl *is* the import artifact.
2. **Bulk Import (Memgraph):**
   ```
   Cypher
   CALL json_util.load_from_path("/var/lib/memgraph/import/graph.jsonl")
   YIELD objects
   UNWIND objects AS o
   MERGE (n:Entity {id: o.id})
   SET n += o.props
   WITH n, o
   UNWIND o.edges AS e
   MATCH (t:Entity {id: e.target})
   MERGE (n)-->(t)
   SET r += e.props;
   ```

   Note: For massive datasets, converting JSONL to CSV and using the LOAD CSV clause or the neo4j-admin import tool is significantly faster (orders of magnitude) than Cypher-based JSON loading. A simple Python script using pandas can convert the

JSONL to nodes.csv and edges.csv.[26]

6.3 Query Translation

The algorithmic logic (Python BFS/DFS) will need to be rewritten in **Cypher**.

- *Python:* for neighbor in node.neighbors: if neighbor.id == target...
- Cypher: MATCH (start:Activity {id: $id})-->(prereq) RETURN prereq
  The schema design (Section 5) ensures that this translation is syntactically direct. The "Lazy Node" logic in Python is effectively what the Cypher engine performs internally during query execution.

6.4 Memgraph MAGE Integration

Memgraph offers **MAGE** (Memgraph Advanced Graph Extensions), a library of graph algorithms and query modules. The custom Python traversal logic written for the file-based graph can often be ported directly into Memgraph as a **Query Module** (written in Python). This allows the exact same BFS/DFS logic (with custom filtering rules) to run *inside* the Memgraph database engine, providing a seamless transition from "Client-Side Python Logic" to "Server-Side Python Logic".[28]

---

7. Conclusions

The constraints of the Baby Brains Montessori project—specifically the inability to run a resident Graph DBMS—necessitate a **Serverless Graph Architecture**. By rejecting the assumption that a graph must live in RAM, we utilize the file system's capabilities to build a performant, disk-resident system.

Key Recommendations:

1. **Storage:** Adopt **JSONL** with a binary **Sidecar Offset Index**. This combination provides the $O(1)$ random access required for graph traversal while maintaining the flexibility of a schema-less document store.
2. **Algorithmics:** Implement graph traversals using **NoGraphs** or custom generators that leverage **Lazy Loading**. This keeps the memory footprint bounded to the current traversal frontier, regardless of the total graph size.
3. **Search:** Deploy **LanceDB** for embedded, disk-based vector search. Use a "Semantic Search $\rightarrow$ Structural Filter" pattern to enable hybrid queries.
4. **Schema:** Align strictly with **IEEE 1752.1** for the health data layer. Use fitdecode to map Garmin binary streams into this standardized JSON format.
5. **Future-Proofing:** Structure the JSONL data to mirror the Property Graph model (Nodes, Relationships, Properties). This ensures that migrating to Memgraph or Neo4j in the future is a data ingestion task, not a re-architecting task.

This report demonstrates that "Big Data" techniques (mmap, offset indexing, lazy evaluation) can be successfully miniaturized to solve complex relationship management problems in low-resource environments.

Appendix A: Detailed Schema Mapping Tables

Table 1: Garmin FIT to Open mHealth / IEEE 1752.1 Mapping

Garmin FIT Field (Message: record)	**Open mHealth / IEEE 1752.1 Schema**	**Graph Node Representation (JSONL)**	**Unit**
heart_rate	omh:heart-rate:1.0	{"label": "Observation", "props": {"type": "heart_rate", "val": 120}}	beats/min
cadence	omh:cadence:1.0	{"label": "Observation", "props": {"type": "cadence", "val": 85}}	steps/min
steps	omh:step-count:2.0	{"label": "Observation", "props": {"type": "step_count", "val": 500}}	steps
speed	omh:speed:1.0	{"label": "Observation", "props": {"type": "speed", "val": 2.5}}	m/s
Message: session			
total_calories	omh:calories-burned:1.0	{"label": "SessionSummary", "props": {"type":	kcal

		"calories", "val": 300}}	
start_time	omh:time-frame	{"label": "SessionSummary", "props": {"timestamp": "2023-10-01..."}}	ISO 8601

Table 2: Montessori YAML to Graph Topology

YAML Field	Graph Element	Logic
activity_name	Node	Create Node with label :Activity.
prerequisites: [list]	Edge	Create Directed Edge `` from Activity to each Prereq.
age_range	Property	Add property age_range to the Activity Node.
materials: [list]	Edge	Create Edge `` from Activity to Material Node.
control_of_error	Property	Add text property describing how the child self-corrects.

Table 3: Technology Stack Selection

Component	Recommended Tool	Rationale for Low-RAM
Graph Storage	**JSONL + Offset Index**	Zero-copy mmap access; $O(1)$ seek; append-only friendly.

Vector Store	**LanceDB**	Disk-based IVF-PQ index; embedded; Arrow integration.
Graph Algo	**NoGraphs**	Pure Python; lazy-evaluation first; low overhead.
FIT Parser	**fitdecode**	Handles chained files; access to raw headers; memory efficient iterators.
Schema	**IEEE 1752.1**	Standardized interoperability; JSON-native.

Works cited

1. Out-of-core Graph Algorithms - Apache GraphAr, accessed on January 4, 2026, https://graphar.apache.org/docs/libraries/cpp/examples/out-of-core/
2. Converting JSON into knowledge graphs : r/Neo4j - Reddit, accessed on January 4, 2026, https://www.reddit.com/r/Neo4j/comments/1knje83/converting_json_into_knowledge_graphs/
3. Python mmap: Improved File I/O With Memory Mapping, accessed on January 4, 2026, https://realpython.com/python-mmap/
4. Why mmap() is faster than sequential IO? [duplicate] - Stack Overflow, accessed on January 4, 2026, https://stackoverflow.com/questions/9817233/why-mmap-is-faster-than-sequential-io
5. Bad Linux Memory Mapped File Performance with Random Access C++ & Python, accessed on January 4, 2026, https://stackoverflow.com/questions/26349045/bad-linux-memory-mapped-file-performance-with-random-access-c-python
6. Flat files vs SQLite - SQLite User Forum, accessed on January 4, 2026, https://sqlite.org/forum/forumpost/3d7be1ad3d?t=h
7. fast-jsonl - PyPI, accessed on January 4, 2026, https://pypi.org/project/fast-jsonl/
8. rrivera1849/fastjsonlreader: Reads JSONL files quickly without loading them into memory, co-written by GPT-5 - GitHub, accessed on January 4, 2026, https://github.com/rrivera1849/fastjsonlreader
9. Justin1904/sjsonl - GitHub, accessed on January 4, 2026, https://github.com/Justin1904/sjsonl

10. Random indexing of large Json file compressed as Gzip - Stack Overflow, accessed on January 4, 2026, https://stackoverflow.com/questions/74460186/random-indexing-of-large-json-file-compressed-as-gzip
11. Adjacency List in Python - GeeksforGeeks, accessed on January 4, 2026, https://www.geeksforgeeks.org/dsa/adjacency-list-in-python/
12. Adjacency List (With Code in C, C++, Java and Python) - Programiz, accessed on January 4, 2026, https://www.programiz.com/dsa/graph-adjacency-list
13. The Lazy Loading Pattern: How to Make Python Programs Feel Instant - YouTube, accessed on January 4, 2026, https://www.youtube.com/watch?v=ENnDxEOAKKc
14. nographs · PyPI, accessed on January 4, 2026, https://pypi.org/project/nographs/
15. Graph traversal python patterns that help you think less and code faster - Medium, accessed on January 4, 2026, https://medium.com/@pv.safronov/graph-traversal-python-patterns-that-help-you-think-less-and-code-faster-66f76e1ab820
16. Chroma vs LanceDB | Vector Database Comparison - Zilliz, accessed on January 4, 2026, https://zilliz.com/comparison/chroma-vs-lancedb
17. Indexing Data - LanceDB, accessed on January 4, 2026, https://docs.lancedb.com/indexing
18. What is a Graph-Based Vector Database? (And When to Use It Over Milvus) | Sealos Blog, accessed on January 4, 2026, https://sealos.io/blog/what-is-a-graph-based-vector-database-and-when-to-use-it-over-milvus
19. Vectors and Graphs: Better Together - Graph Database & Analytics - Neo4j, accessed on January 4, 2026, https://neo4j.com/blog/developer/vectors-graphs-better-together/
20. 1752.1 Metadata Schemas - of IEEE Standards Working Groups, accessed on January 4, 2026, https://sagroups.ieee.org/1752/wp-content/uploads/sites/277/2022/04/2022-04-28-1752-CardioRespiratory-slides.pdf
21. schemas · 6dd9386d1ff33bb912ea450df8ca3aca25ca2752 · Open Mobile Health / 1752 · GitLab, accessed on January 4, 2026, https://opensource.ieee.org/omh/1752/-/tree/6dd9386d1ff33bb912ea450df8ca3aca25ca2752/schemas
22. fitdecode — fitdecode, accessed on January 4, 2026, https://fitdecode.readthedocs.io/
23. polyvertex/fitdecode: A FIT file parsing and decoding library written in Python3 - GitHub, accessed on January 4, 2026, https://github.com/polyvertex/fitdecode
24. Under the Covers With LightRAG: Extraction - Graph Database & Analytics - Neo4j, accessed on January 4, 2026, https://neo4j.com/blog/developer/under-the-covers-with-lightrag-extraction/
25. Top 10 Graph Database Use Cases (With Real-World Case Studies) - Neo4j, accessed on January 4, 2026, https://neo4j.com/blog/graph-database/graph-database-use-cases/
26. LOAD CSV - Cypher Manual - Neo4j, accessed on January 4, 2026,

https://neo4j.com/docs/cypher-manual/current/clauses/load-csv/
27. Import best practices - Memgraph, accessed on January 4, 2026, https://memgraph.com/docs/data-migration/best-practices
28. How to Implement Custom JSON Utility Procedures With Memgraph MAGE and Python., accessed on January 4, 2026, https://memgraph.com/blog/how-to-implement-custom-json-utility-procedures-with-memgraph-mage-and-python