**End‑to‑End Data Model for GraphRAG (Neo4j + Vector + RDF)**

Generated: 2025-09-01 19:13 UTC

This document defines a reusable data model for building a GraphRAG system inspired by the two repositories you shared. It includes conceptual, logical (graph), and physical model details, plus RDBMS and RDF/OWL mappings, constraints, vector index settings, and example queries.

**1) Conceptual Model (Text Diagram)**

[Document] ──hasSection──> [Section] ──hasChunk──> [Chunk] ──NEXT──> [Chunk]  
 │ │  
 │ └─mentions──> [Entity] <──mentions── [Chunk]  
 │ └─aliasOf/label→ [Synonym/SKOS]  
 └─sourceOf/cites──> [Source]  
  
[Entity] ──typedRelations──> [Entity] (e.g., RELATED\_TO, LOCATED\_IN, PART\_OF, OCCURRED\_ON)  
  
Query paths:  
- GraphRAG: Question → LLM → Cypher → Neo4j (Entities + relations + chunk links)  
- VectorRAG: Question → Embed → Vector search over Chunk.embeddings → top‑k Chunks + metadata

**2) Logical Graph Model (Neo4j labels & properties)**

|  |  |  |
| --- | --- | --- |
| Label (Class) | Key Properties | Notes / Uniqueness |
| Document | docId (string, unique), title, sourceType (web/pdf/wiki), url, createdAt (datetime) | One node per source document. Unique on docId. |
| Section | sectionId (string, unique), name, order (int) | Optional layer between Document and Chunk (e.g., chapters). |
| Chunk | chunkId (string, unique), text, chunkSeqId (int), formItem (string), source (string), textEmbeddingOpenAI (vector) | Atomic retrieval unit. Vector index lives on textEmbeddingOpenAI. |
| Entity | uri (string, unique) or (name, type); name, type (Person/Event/Place/Topic/Org), aliases (array?) | Domain concepts you query with GraphRAG. |
| Source | sourceId (string, unique), name, kind (site/book/paper), url | Optional reference metadata node. |

Relationship types (directed):

(:Document)-[:HAS\_SECTION]->(:Section)  
(:Section)-[:HAS\_CHUNK]->(:Chunk)  
(:Chunk)-[:NEXT]->(:Chunk) // preserves order within a Section/formItem  
(:Chunk)-[:MENTIONS]->(:Entity) // link extracted mentions  
(:Document)-[:CITES]->(:Source) // optional  
(:Entity)-[:ALIAS\_OF]->(:Entity) // optional SKOS‑like aliasing  
(:Entity)-[:RELATED\_TO|:LOCATED\_IN|:PART\_OF|:OCCURRED\_ON]->(:Entity) // domain relations

**3) Physical Model (Neo4j: Constraints, Text & Vector Indexes)**

Create uniqueness constraints to ensure clean merges, plus a vector index on chunk embeddings.

// Uniqueness constraints  
CREATE CONSTRAINT doc\_id IF NOT EXISTS FOR (d:Document) REQUIRE d.docId IS UNIQUE;  
CREATE CONSTRAINT sec\_id IF NOT EXISTS FOR (s:Section) REQUIRE s.sectionId IS UNIQUE;  
CREATE CONSTRAINT chunk\_id IF NOT EXISTS FOR (c:Chunk) REQUIRE c.chunkId IS UNIQUE;  
CREATE CONSTRAINT entity\_uri IF NOT EXISTS FOR (e:Entity) REQUIRE e.uri IS UNIQUE;  
  
// Optional btree/text indexes for fast lookup by name/code  
CREATE INDEX entity\_name IF NOT EXISTS FOR (e:Entity) ON (e.name);  
CREATE INDEX doc\_title IF NOT EXISTS FOR (d:Document) ON (d.title);  
  
// Vector index (Neo4j 5.x native vector indexes)  
CREATE VECTOR INDEX chunk\_openai IF NOT EXISTS  
FOR (c:Chunk) ON (c.textEmbeddingOpenAI)  
OPTIONS { indexConfig: {  
 `vector.dimensions`: 1536,  
 `vector.similarity\_function`: 'cosine'  
}};

Embedding in-DB (typical pattern using a procedure or your app layer):

// Pseudocode / Cypher pattern:  
MATCH (c:Chunk) WHERE c.textEmbeddingOpenAI IS NULL  
CALL genai.vector.encode(  
 'OpenAI',  
 { token: $OPENAI\_API\_KEY, input: c.text, model: 'text-embedding-3-small' }  
) YIELD embedding  
CALL db.create.setNodeVectorProperty(c, 'textEmbeddingOpenAI', embedding) YIELD node  
RETURN count(node);

**4) RDBMS Analogue (tables & bridge tables)**

TABLE document(doc\_id PK, title, source\_type, url, created\_at)  
TABLE section(section\_id PK, doc\_id FK, name, seq\_no)  
TABLE chunk(chunk\_id PK, section\_id FK, text, chunk\_seq\_id, form\_item, source)  
TABLE entity(entity\_id PK, uri UNIQUE, name, type)  
TABLE chunk\_entity(chunk\_id FK, entity\_id FK) -- MENTIONS bridge  
TABLE entity\_relation(src\_entity\_id FK, rel\_type, dst\_entity\_id FK, props JSONB)  
TABLE source(source\_id PK, name, kind, url)  
TABLE document\_source(doc\_id FK, source\_id FK) -- CITES bridge

This mirrors the graph model: bridge tables capture edges; `chunk` is your retrieval unit; you’d keep embeddings in a vector store (e.g., pgvector) or externally.

**5) RDF/OWL Mapping (classes, properties, SKOS)**

@prefix ex: <http://example.com/kg#> .  
@prefix dct: <http://purl.org/dc/terms/> .  
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .  
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .  
  
ex:Document a rdfs:Class .  
ex:Section a rdfs:Class .  
ex:Chunk a rdfs:Class .  
ex:Entity a rdfs:Class .  
ex:Source a rdfs:Class .  
  
ex:hasSection a rdf:Property ; rdfs:domain ex:Document ; rdfs:range ex:Section .  
ex:hasChunk a rdf:Property ; rdfs:domain ex:Section ; rdfs:range ex:Chunk .  
ex:next a rdf:Property ; rdfs:domain ex:Chunk ; rdfs:range ex:Chunk .  
ex:mentions a rdf:Property ; rdfs:domain ex:Chunk ; rdfs:range ex:Entity .  
ex:cites a rdf:Property ; rdfs:domain ex:Document ; rdfs:range ex:Source .  
ex:aliasOf a rdf:Property ; rdfs:domain ex:Entity ; rdfs:range ex:Entity .  
  
# Labels / synonyms via SKOS  
ex:Entity skos:prefLabel "Canonical name"@en ;  
 skos:altLabel "Alias"@en .  
  
# Example instance  
ex:doc1 a ex:Document ; dct:title "Napoleon overview" .  
ex:sec1 a ex:Section ; ex:name "Military Career" .  
ex:doc1 ex:hasSection ex:sec1 .  
ex:ch1 a ex:Chunk ; ex:text "In 1804 Napoleon crowned himself..." .  
ex:sec1 ex:hasChunk ex:ch1 .

**6) Load Patterns (Neo4j MERGE + Vector)**

// 1) Load a document, its sections, chunks  
UNWIND $docs AS d  
MERGE (doc:Document {docId:d.docId})  
SET doc.title = d.title, doc.sourceType = d.sourceType, doc.url = d.url  
WITH doc, d.sections AS sections  
UNWIND sections AS s  
MERGE (sec:Section {sectionId:s.sectionId})  
SET sec.name = s.name, sec.order = s.order  
MERGE (doc)-[:HAS\_SECTION]->(sec)  
WITH sec, s.chunks AS chunks  
UNWIND chunks AS c  
MERGE (ch:Chunk {chunkId:c.chunkId})  
SET ch.text = c.text, ch.chunkSeqId=c.chunkSeqId, ch.formItem=c.formItem, ch.source=c.source  
MERGE (sec)-[:HAS\_CHUNK]->(ch)  
WITH ch, c  
ORDER BY c.chunkSeqId  
WITH collect(ch) AS cs  
UNWIND range(0, size(cs)-2) AS i  
WITH cs[i] AS a, cs[i+1] AS b  
MERGE (a)-[:NEXT]->(b);  
  
// 2) Mentions (NER or rule-based)  
UNWIND $mentions AS m  
MATCH (ch:Chunk {chunkId:m.chunkId})  
MERGE (e:Entity {uri:m.uri})  
SET e.name = m.name, e.type = m.type  
MERGE (ch)-[:MENTIONS]->(e);  
  
// 3) Embeddings (done after load)  
MATCH (c:Chunk) WHERE c.textEmbeddingOpenAI IS NULL  
CALL genai.vector.encode('OpenAI', {token:$OPENAI\_API\_KEY, input:c.text, model:'text-embedding-3-small'}) YIELD embedding  
CALL db.create.setNodeVectorProperty(c,'textEmbeddingOpenAI',embedding) YIELD node  
RETURN count(node);

**7) Query Patterns (GraphRAG & VectorRAG)**

GraphRAG example (relationships first, then compose an answer):

// Entities connected by mentions within the same document  
MATCH (d:Document)-[:HAS\_SECTION]->(:Section)-[:HAS\_CHUNK]->(c:Chunk)-[:MENTIONS]->(e1:Entity)  
MATCH (c)-[:MENTIONS]->(e2:Entity)  
WHERE e1.name='Napoleon' AND e2.type='Place'  
RETURN d.title AS document, e2.name AS place, count(\*) AS coMentions  
ORDER BY coMentions DESC LIMIT 10;

VectorRAG example (top‑k chunks + follow NEXT edges to expand context):

// Top‑k vector search  
CALL db.index.vector.queryNodes('chunk\_openai', $k, $queryEmbedding) YIELD node, score  
WITH node, score  
OPTIONAL MATCH path = node-[:NEXT\*0..2]->(:Chunk)  
WITH node, score, nodes(path) AS ctx  
RETURN node.text AS hit, score, [x IN ctx | x.text][0..3] AS extra\_context;

**8) Governance & Quality (Constraints & Shapes)**

Neo4j constraints protect keys. For rich shape constraints (e.g., each Chunk must belong to exactly one Section) use SHACL in RDF or validate with Cypher checks.

// Example Cypher validation: each Chunk should have exactly one incoming HAS\_CHUNK  
MATCH (c:Chunk)  
WITH c, size( ()-[:HAS\_CHUNK]->(c) ) AS parents  
WHERE parents <> 1  
RETURN c.chunkId AS chunk, parents AS parentCount;

**9) Extending to Domain Entities (Car Example)**

Add domain labels and link them to chunks via MENTIONS; keep vector search on chunks.

(:VehicleModel {code:'FALCON\_X'})<-[:BELONGS\_TO]-(:Trim {code:'T101', modelYear:2023})-[:HAS\_FEATURE]->(:Feature {code:'ACC'})  
(:SaleRecord {id:'S1', saleDate:date('2023-03-15'), quantity:120})-[:FOR\_TRIM]->(:Trim {code:'T101'})-[:BELONGS\_TO]->(:VehicleModel {code:'FALCON\_X'})  
(:SaleRecord {id:'S1'})-[:SOLD\_IN]->(:Region {code:'CA'})

Link text to entities: `(:Chunk)-[:MENTIONS]->(:Trim {code:'T101'})`, etc. Your GraphRAG prompts should name these labels and relations so the LLM generates valid Cypher.

**10) Lineage & Serving (Text Diagram)**

[Raw Docs] → [Preprocess/Section JSON] → [Chunk] → [Neo4j MERGE]  
 └─► [Entity extraction] → (Chunk)-[:MENTIONS]->(Entity)  
 └─► [Vector encode] → Chunk.textEmbeddingOpenAI (vector index)  
  
Serve:  
- GraphRAG: Question → LLM → Cypher → Paths/Rows → Compose Answer (with node/edge citations)  
- VectorRAG: Question → Embed → Vector search → Top‑k Chunks (+ NEXT context) → Compose Answer (with chunk citations)