Graph Databases and Hybrid RAG: From Raw Data to Natural Language Querying

Building Knowledge Graphs and Integrating with Vector-Based Retrieval

Introduction: Why Graph Databases Matter for NLP

Imagine organizing your contacts not just as a list, but as a web of relationships: who knows whom, who works where, and who shares interests. This interconnected view reveals patterns invisible in traditional tables. **Graph databases** store data as nodes (entities) and edges (relationships), making them perfect for knowledge representation in NLP applications.

Why it matters:

- **Relationships are first-class citizens:** Unlike SQL tables, graphs naturally model connections
- **Flexible schema:** Easy to add new entity types and relationship patterns
- **Semantic queries:** Natural language questions often involve traversing relationships
- **Context preservation:** Maintains rich contextual information for better retrieval

This guide covers the complete pipeline: transforming raw data into graph structures, storing them efficiently, enabling natural language querying, and integrating with vector databases for hybrid RAG systems.

1. Creating Graph Structures from Raw Data

Understanding the Transformation Process

Converting raw text or structured data into graph format requires identifying **entities** (nouns) and **relationships** (verbs/predicates). Think of it as turning sentences into a network of connected concepts.

Example transformation:

- Raw text: "Alice works at Google in Mountain View. Google was founded by Larry Page."
- Graph structure:
  - Nodes: Alice (Person), Google (Company), Mountain View (Location), Larry Page (Person)
  - Edges: Alice → WORKS_AT → Google, Google → LOCATED_IN → Mountain View, Larry Page → FOUNDED → Google

Entity and Relationship Extraction

```
import spacy
from neo4j import GraphDatabase

# Load NLP model for entity extraction
```

```
nlp = spacy.load("en_core_web_sm")

def extract_entities_relationships(text):
    doc = nlp(text)

    entities = []
    relationships = []

    for ent in doc.ents:
        entities.append({
            'text': ent.text,
            'label': ent.label_,
            'start': ent.start_char,
            'end': ent.end_char
        })

    # Extract relationships using dependency parsing
    for token in doc:
        if token.dep_ in ['nsubj', 'dobj'] and token.head.pos_ == 'VERB':
            relationships.append({
                'subject': token.text,
                'predicate': token.head.text,
                'object': [child.text for child in token.head.children
                           if child.dep_ == 'dobj']
            })

    return entities, relationships

# Example usage
text = "Apple Inc. is headquartered in Cupertino, California."
entities, relationships = extract_entities_relationships(text)
```

This code demonstrates basic entity extraction using spaCy, identifying people, organizations, and locations, then finding relationships through dependency parsing.

Advanced Graph Schema Design

For complex domains, define a clear schema:

```
# Graph schema definition
GRAPH_SCHEMA = {
    'node_types': {
        'Person': ['name', 'age', 'email'],
        'Company': ['name', 'industry', 'founded_year'],
        'Location': ['name', 'country', 'coordinates']
    },
    'relationship_types': {
        'WORKS_AT': ['start_date', 'position'],
        'LOCATED_IN': ['since'],
        'FOUNDED': ['date', 'stake_percentage']
    }
}
```

2. Storing Graphs in Neo4j and AWS Neptune

Neo4j Implementation

Neo4j uses **Cypher** query language and provides excellent tooling for graph visualization and management.

```
from neo4j import GraphDatabase

class GraphBuilder:
    def __init__(self, uri, user, password):
        self.driver = GraphDatabase.driver(uri, auth=(user, password))

    def close(self):
        self.driver.close()

    def create_person_node(self, name, properties=None):
        with self.driver.session() as session:
            query = """
            CREATE (p:Person {name: $name})
            SET p += $properties
            RETURN p
            """
            result = session.run(query, name=name, properties=properties or {})
            return result.single()

    def create_relationship(self, from_name, to_name, rel_type, properties=None):
        with self.driver.session() as session:
            query = f"""
            MATCH (a:Person {{name: $from_name}})
            MATCH (b:Person {{name: $to_name}})
            CREATE (a)-[r:{rel_type}]-&gt;(b)
            SET r += $properties
            RETURN r
            """
            result = session.run(query,
                                 from_name=from_name,
                                 to_name=to_name,
                                 properties=properties or {})
            return result.single()

# Usage example
graph_builder = GraphBuilder("bolt://localhost:7687", "neo4j", "password")

# Create nodes
graph_builder.create_person_node("Alice", {"age": 30, "department": "Engineering"})
graph_builder.create_person_node("Bob", {"age": 25, "department": "Marketing"})

# Create relationship
graph_builder.create_relationship("Alice", "Bob", "COLLEAGUES_WITH",
                                  {"since": "2023-01-15"})
```

AWS Neptune Implementation

AWS Neptune supports both **Gremlin** (for property graphs) and **SPARQL** (for RDF graphs).

```
from gremlin_python.driver import client
from gremlin_python.driver.driver_remote_connection import DriverRemoteConnection
from gremlin_python.process.anonymous_traversal import traversal

# Connect to Neptune
endpoint = 'wss://your-cluster.cluster-xyz.us-west-2.neptune.amazonaws.com:8182/gremlin'
connection = DriverRemoteConnection(endpoint, 'g')
g = traversal().withRemote(connection)

# Create nodes and relationships
def create_knowledge_graph():
    # Add person vertex
    alice = g.addV('Person').property('name', 'Alice').property('age', 30).next()
    google = g.addV('Company').property('name', 'Google').property('industry', 'Technology

    # Add edge (relationship)
    g.V(alice).addE('WORKS_AT').to(g.V(google)).property('position', 'Software Engineer').

    return alice, google

# Query the graph
def find_employees_at_company(company_name):
    employees = (g.V().has('Company', 'name', company_name)
                 .in_('WORKS_AT')
                 .values('name')
                 .toList())
    return employees

# Example usage
create_knowledge_graph()
employees = find_employees_at_company('Google')
print(f"Employees at Google: {employees}")
```

3. Natural Language to Query Translation

The Challenge of NL2Query

Converting natural language questions into database queries is complex because:

1. **Ambiguity:** "Who works with Alice?" could mean colleagues, subordinates, or collaborators
2. **Implicit relationships:** "Alice's company" implies a WORKS_AT relationship
3. **Complex traversals:** "Friends of Alice's colleagues" requires multi-hop navigation

Template-Based Approach

Start with predefined patterns for common question types:

```
import re
from typing import Dict, List

class NL2CypherTranslator:
    def __init__(self):
        self.patterns = [
            {
                'pattern': r'who works at (.+)',
                'template': 'MATCH (p:Person)-[:WORKS_AT]-&gt;(c:Company {name: "$1"}) RET
                'description': 'Find employees of a company'
            },
            {
                'pattern': r'where does (.+) work',
                'template': 'MATCH (p:Person {name: "$1"})-[:WORKS_AT]-&gt;(c:Company) RET
                'description': 'Find workplace of a person'
            },
            {
                'pattern': r'who are (.+)\'s colleagues',
                'template': '''MATCH (p1:Person {name: "$1"})-[:WORKS_AT]-&gt;(c:Company)
                               MATCH (p2:Person)-[:WORKS_AT]-&gt;(c)
                               WHERE p1 &lt;&gt; p2
                               RETURN p2.name''',
                'description': 'Find colleagues of a person'
            }
        ]

    def translate(self, question: str) -&gt; str:
        question = question.lower().strip('?')

        for pattern_info in self.patterns:
            match = re.search(pattern_info['pattern'], question)
            if match:
                query = pattern_info['template']
                for i, group in enumerate(match.groups(), 1):
                    query = query.replace(f'${i}', group)
                return query

        return "Sorry, I don't understand this question type yet."

# Example usage
translator = NL2CypherTranslator()
query1 = translator.translate("Who works at Google?")
query2 = translator.translate("Where does Alice work?")
query3 = translator.translate("Who are Alice's colleagues?")

print(f"Query 1: {query1}")
print(f"Query 2: {query2}")
print(f"Query 3: {query3}")
```

LLM-Based Translation

For more sophisticated translation, use large language models:

```
import openai
from typing import Optional

class LLMNLQueryTranslator:
    def __init__(self, openai_api_key: str):
        openai.api_key = openai_api_key

        self.system_prompt = """
        You are an expert at translating natural language questions into Cypher queries.

        Graph Schema:
        - Nodes: Person(name, age), Company(name, industry), Location(name, country)
        - Relationships: WORKS_AT, LOCATED_IN, FOUNDED, COLLEAGUES_WITH

        Examples:
        Question: "Who works at Google?"
        Cypher: MATCH (p:Person)-[:WORKS_AT]-&gt;(c:Company {name: 'Google'}) RETURN p.nam

        Question: "What companies are in California?"
        Cypher: MATCH (c:Company)-[:LOCATED_IN]-&gt;(l:Location {name: 'California'}) RETU

        Always return valid Cypher syntax only.
        """

    def translate(self, question: str) -&gt; str:
        try:
            response = openai.ChatCompletion.create(
                model="gpt-4",
                messages=[
                    {"role": "system", "content": self.system_prompt},
                    {"role": "user", "content": f"Question: {question}"}
                ],
                temperature=0.1,
                max_tokens=200
            )
            return response.choices[0].message.content.strip()
        except Exception as e:
            return f"Error translating query: {str(e)}"

# Usage
translator = LLMNLQueryTranslator("your-openai-api-key")
cypher_query = translator.translate("Who are the founders of companies in Silicon Valley?"
```

4. Hybrid RAG Implementation: Vector + Graph

Architecture Overview

Hybrid RAG combines the strengths of both approaches:

- **Vector databases:** Semantic similarity for unstructured text
- **Graph databases:** Precise relationship traversal for structured knowledge

The system routes queries to the appropriate database based on query type and combines results for comprehensive answers.

Implementation Framework

```
import numpy as np
from sentence_transformers import SentenceTransformer
import faiss
from neo4j import GraphDatabase
from typing import List, Dict, Tuple

class HybridRAGSystem:
    def __init__(self, neo4j_uri: str, neo4j_user: str, neo4j_password: str):
        # Initialize graph database
        self.graph_driver = GraphDatabase.driver(neo4j_uri, auth=(neo4j_user, neo4j_passwo

        # Initialize vector database
        self.embedding_model = SentenceTransformer('all-MiniLM-L6-v2')
        self.vector_index = None
        self.documents = []

        # Query routing patterns
        self.graph_patterns = [
            r'who (works|worked) at',
            r'what companies',
            r'(colleagues|friends|partners) of',
            r'founded by',
            r'located in'
        ]

    def add_documents(self, documents: List[str]):
        """Add documents to vector index"""
        self.documents.extend(documents)
        embeddings = self.embedding_model.encode(documents)

        if self.vector_index is None:
            dimension = embeddings.shape[1]
            self.vector_index = faiss.IndexFlatIP(dimension)

        self.vector_index.add(embeddings.astype('float32'))

    def should_use_graph(self, query: str) -&gt; bool:
        """Determine if query should use graph database"""
        query_lower = query.lower()
        return any(re.search(pattern, query_lower) for pattern in self.graph_patterns)

    def graph_query(self, cypher_query: str) -&gt; List[Dict]:
        """Execute Cypher query against graph database"""
```

```
        with self.graph_driver.session() as session:
            result = session.run(cypher_query)
            return [record.data() for record in result]

    def vector_search(self, query: str, k: int = 5) -&gt; List[Tuple[str, float]]:
        """Search vector database for similar documents"""
        if self.vector_index is None:
            return []

        query_embedding = self.embedding_model.encode([query])
        scores, indices = self.vector_index.search(query_embedding.astype('float32'), k)

        results = []
        for score, idx in zip(scores[0], indices[0]):
            if idx &lt; len(self.documents):
                results.append((self.documents[idx], float(score)))

        return results

    def retrieve_context(self, query: str) -&gt; Dict[str, any]:
        """Retrieve context from both graph and vector databases"""
        context = {'graph_results': [], 'vector_results': [], 'query_type': 'hybrid'}

        if self.should_use_graph(query):
            # Translate natural language to Cypher (using previous translator)
            nl_translator = NL2CypherTranslator()
            cypher_query = nl_translator.translate(query)

            if not cypher_query.startswith('Sorry'):
                context['graph_results'] = self.graph_query(cypher_query)
                context['query_type'] = 'graph'

        # Always get vector results for additional context
        context['vector_results'] = self.vector_search(query, k=3)

        return context

    def generate_response(self, query: str, context: Dict) -&gt; str:
        """Generate final response using retrieved context"""
        prompt_parts = [f"Question: {query}\n"]

        if context['graph_results']:
            prompt_parts.append("Structured Knowledge:")
            for result in context['graph_results']:
                prompt_parts.append(f"- {result}")
            prompt_parts.append("")

        if context['vector_results']:
            prompt_parts.append("Related Documents:")
            for doc, score in context['vector_results']:
                prompt_parts.append(f"- {doc} (similarity: {score:.3f})")
            prompt_parts.append("")

        prompt_parts.append("Based on the above information, provide a comprehensive answe

        # Here you would call your LLM (GPT, Claude, etc.)
```

```
        # For this example, we'll return a structured summary
        return {
            'answer': "Generated answer would go here",
            'sources': {
                'graph_facts': len(context['graph_results']),
                'documents': len(context['vector_results']),
                'query_type': context['query_type']
            },
            'prompt': '\n'.join(prompt_parts)
        }

# Example usage
hybrid_rag = HybridRAGSystem("bolt://localhost:7687", "neo4j", "password")

# Add some documents
documents = [
    "Google was founded by Larry Page and Sergey Brin in 1998.",
    "Apple Inc. is headquartered in Cupertino, California.",
    "Microsoft was established in 1975 by Bill Gates and Paul Allen."
]
hybrid_rag.add_documents(documents)

# Query the system
query = "Who founded Google?"
context = hybrid_rag.retrieve_context(query)
response = hybrid_rag.generate_response(query, context)

print(f"Query: {query}")
print(f"Response: {response}")
```

Query Routing Logic

The system uses intelligent routing to determine the optimal retrieval strategy:

```
def advanced_query_routing(query: str) -&gt; str:
    """Determine optimal retrieval strategy based on query analysis"""

    # Relationship queries -&gt; Graph
    if re.search(r'\b(who|what|which).*(work|employ|found|own)', query, re.I):
        return 'graph_primary'

    # Factual questions about entities -&gt; Graph + Vector
    if re.search(r'\b(about|describe|explain)', query, re.I):
        return 'hybrid'

    # Semantic/conceptual questions -&gt; Vector primary
    if re.search(r'\b(similar|like|compare|difference)', query, re.I):
        return 'vector_primary'

    # Default to hybrid
    return 'hybrid'
```

5. Performance Optimization and Best Practices

Graph Database Optimization

Indexing Strategy:

```
-- Create indexes for frequent lookup properties
CREATE INDEX person_name IF NOT EXISTS FOR (p:Person) ON (p.name);
CREATE INDEX company_name IF NOT EXISTS FOR (c:Company) ON (c.name);

-- Create composite indexes for complex queries
CREATE INDEX person_company IF NOT EXISTS FOR (p:Person) ON (p.name, p.company);
```

Query Optimization:

```
-- Instead of this (inefficient)
MATCH (p:Person)
WHERE p.name = 'Alice'
MATCH (p)-[:WORKS_AT]-&gt;(c:Company)
RETURN c.name;

-- Use this (efficient)
MATCH (p:Person {name: 'Alice'})-[:WORKS_AT]-&gt;(c:Company)
RETURN c.name;
```

Vector Database Optimization

```
# Use appropriate index types based on use case
# For exact search
index = faiss.IndexFlatL2(dimension)

# For approximate search (faster, large datasets)
index = faiss.IndexIVFFlat(quantizer, dimension, nlist)

# For memory efficiency
index = faiss.IndexIVFPQ(quantizer, dimension, nlist, m, nbits)
```

Caching Strategy

```
from functools import lru_cache
import hashlib

class CachedHybridRAG(HybridRAGSystem):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.query_cache = {}

    def _cache_key(self, query: str) -&gt; str:
        return hashlib.md5(query.encode()).hexdigest()
```

```
    @lru_cache(maxsize=1000)
    def cached_graph_query(self, cypher_query: str) -&gt; str:
        return str(self.graph_query(cypher_query))

    def retrieve_context(self, query: str) -&gt; Dict[str, any]:
        cache_key = self._cache_key(query)

        if cache_key in self.query_cache:
            return self.query_cache[cache_key]

        context = super().retrieve_context(query)
        self.query_cache[cache_key] = context
        return context
```

6. Reflection and Real-World Applications

Key Takeaways

Graph databases excel at:

- Relationship-heavy queries ("Who knows whom?")
- Multi-hop traversals ("Friends of friends")
- Exact factual retrieval
- Explainable results with provenance

Vector databases excel at:

- Semantic similarity matching
- Handling synonyms and paraphrasing
- Working with unstructured text
- Fuzzy matching

Hybrid systems provide:

- Comprehensive coverage of query types
- Enhanced accuracy through multiple evidence sources
- Flexibility to handle diverse information needs

Common Pitfalls

1. **Over-relying on one approach:** Always consider whether hybrid retrieval would improve results
2. **Poor schema design:** Inconsistent entity names and relationship types hurt precision
3. **Inadequate query routing:** Sending graph queries to vector DB and vice versa
4. **Ignoring performance:** Not indexing frequently queried properties
5. **Context overload:** Providing too much retrieved context can confuse language models

Real-World Applications

Enterprise Knowledge Management:

- Employee expertise networks
- Project dependency mapping
- Organizational structure queries

Financial Services:

- Transaction relationship analysis
- Risk assessment through entity connections
- Regulatory compliance tracking

Healthcare:

- Patient-doctor-treatment relationship modeling
- Drug interaction networks
- Medical literature integration

E-commerce:

- Product recommendation through relationship graphs
- Customer behavior analysis
- Supply chain optimization

Future Directions

The field is moving toward:

- **Multimodal graphs:** Incorporating images, videos, and audio
- **Dynamic graphs:** Real-time updates and temporal relationships
- **Federated systems:** Querying across multiple distributed graph and vector databases
- **Automated schema evolution:** Self-improving graph structures

Conclusion

Building hybrid RAG systems with graph and vector databases represents the cutting edge of information retrieval. By combining the precision of structured knowledge graphs with the flexibility of semantic vector search, we create systems that can handle the full spectrum of user queries.

The key to success lies in:

1. **Careful data modeling:** Design graphs that reflect real-world relationships
2. **Smart query routing:** Direct queries to the most appropriate retrieval mechanism
3. **Performance optimization:** Index strategically and cache frequently accessed data
4. **Continuous iteration:** Monitor query patterns and refine the system accordingly

As natural language interfaces become more prevalent, these hybrid systems will become essential for building intelligent, context-aware applications that can truly understand and respond to human information needs.

This comprehensive guide provides everything needed to build production-ready graph database systems with natural language querying and hybrid RAG capabilities. The techniques covered here form the foundation for next-generation AI applications that seamlessly blend structured knowledge with unstructured content.