

RAG AVANCÉ : GRAPHES ET LLAMAINDEX

Aller plus loin que la recherche vectorielle

Philippe Helluy

- 1.** Limites du RAG “naïf”
- 2.** Introduction aux Graphes de Connaissances
- 3.** Graph RAG : le meilleur des deux mondes ?
- 4.** Algorithmes de Reranking
- 5.** Introduction à LlamaIndex
- 6.** Démonstration

La recherche vectorielle est puissante mais...

- Elle repose sur la similarité sémantique “floue”.
- Elle perd souvent les **relations explicites** entre les entités.
- Elle a du mal avec les questions nécessitant un raisonnement multi-sauts (*multi-hop*).
- “Qui est le père du frère de Luke ?” → Difficile pour un simple vecteur.

Texte : “Apple est une entreprise tech. Une pomme est un fruit.”

Question : “Quelle est la relation entre Apple et le fruit ?”

Problème :

- Les vecteurs de “Apple” (entreprise) et “pomme” (fruit) peuvent être éloignés ou proches selon le contexte, mais la **relation** explicite n'est pas codée.
- Le RAG vectoriel ramène les chunks, mais le LLM doit deviner le lien s'il n'est pas explicite dans un seul chunk.

Structurer l'information

- **Noeuds** : Entités (Personnes, Lieux, Concepts)
- **Arêtes** : Relations (EST_UN, TRAVAILLE_POUR, SITUÉ_À)

Exemple : (Elon Musk) -- [PDG de] --> (Tesla) (Tesla) -- [SITUÉ_À] --> (Austin)

→ Permet de traverser les relations pour trouver des réponses précises.

Combiner Vecteurs et Graphes

1. Indexation :

- Extraire les entités et relations du texte (via LLM).
- Stocker dans une base de graphe (ex: Neo4j).
- Garder aussi l'index vectoriel pour le texte brut.

2. Recherche :

- **Keyword Search** sur le graphe.
- **Vector Search** sur les chunks.
- **Graph Traversal** : Explorer les voisins des entités trouvées.

Avantages :

- **Précision** : Capture les faits exacts.
- **Complétude** : Trouve des infos connectées mais distantes dans le texte.
- **Explicabilité** : On peut visualiser le chemin de raisonnement.

Inconvénients :

- **Coût** : L'extraction de graphe (Triple Extraction) consomme beaucoup de tokens.
- **Complexité** : Infrastructure plus lourde (Neo4j, etc.).

Améliorer la pertinence

Le problème : La recherche vectorielle ramène parfois des documents “proches” mais pas “pertinents” pour la réponse.

Solution :

1. Récupérer beaucoup de documents (ex: top-50).
2. Utiliser un modèle de **Reranking** (Cross-Encoder) plus précis mais plus lent.
3. Garder les top-5 pour le LLM.

Algorithmes :

- Cohere Rerank
- BGE Reranker
- Reciprocal Rank Fusion (RRF) pour hybrider keyword + vector.

Le framework de données pour les LLM

Contrairement à LangChain (focalisé sur les chaînes d'exécution), LlamaIndex se focalise sur **l'ingestion et la structuration des données**.

Concepts clés :

- **Data Connectors** : Lire n'importe quoi (PDF, Notion, SQL...).
- **Nodes** : Unité de base (chunk + métadonnées).
- **Indices** : VectorStore, List, Tree, Keyword, Graph.
- **Query Engines** : Interface pour poser des questions.

```
from llama_index.core import VectorStoreIndex, SimpleDirectoryReader

# 1. Chargement
documents = SimpleDirectoryReader("data").load_data()

# 2. Indexation (Vectorielle par défaut)
index = VectorStoreIndex.from_documents(documents)

# 3. Persistance
index.storage_context.persist()
```

```
from llama_index.core import KnowledgeGraphIndex

# Construction automatique du graphe via LLM
graph_index = KnowledgeGraphIndex.from_documents(
    documents,
    max_triplets_per_chunk=2,
    include_embeddings=True
)
```

Note : Nécessite un LLM puissant pour extraire les triplets (Sujet, Prédicat, Objet).

Critère	Vector RAG	Graph RAG
Construction	Rapide	Lente (Extraction)
Coût	Faible	Élevé
Questions “Quoi”	Excellent	Bon
Questions “Comment/ Lien”	Moyen	Excellent
Hallucinations	Faibles	Très faibles

Ce que nous allons voir :

1. Utilisation de `llama_index`.
2. Construction d'un index vectoriel.
3. Construction d'un index de graphe (Knowledge Graph).
4. Comparaison des réponses :
 - Recherche vectorielle seule.
 - Recherche hybride / graphe.

Code : `rag_graph.py`