Knowledge Management & Graphs Concepts

Fabien FURFARO

24 juillet 2025

Sommaire

Historique

La gestion des connaissances (Knowledge Management) a évolué en plusieurs phases :

- Avant 1980 : pratiques informelles, catalogage manuel, échanges oraux.
- 1980-1995 : premiers outils informatiques et bases de connaissances structurées.
- 1995-2010 : plateformes collaboratives, moteurs de recherche spécialisés.
- 2010 et après : NLP/NLU, graph databases, transformers, accent sur gouvernance et modélisation.

Défis actuels et objectifs

- **Fragmentation** de l'information et silos de données.
- **Hétérogénéité** des systèmes.
- **Perte du savoir** par turnover ou absence de capitalisation.

Objectif : créer des plateformes « Google-like », capables de retrouver et structurer la connaissance avec un **KMS** (Knowledge Management System) s'appuyant sur curation humaine, ontologie métier, indexation croisée et technologies avancées (ex : knowledge graph, IA).

Composants d'un KMS moderne

- Modèles formels, ontologies.
- Moteurs de règle, systèmes experts.
- Knowledge graph, sémantique.
- IA (NLP/NLU, entity recognition, retrieval).
- Interfaces LLM/RAG, intégration outils métier.

Cas d'usage : Curiosity

- KMS basé sur un **Linked Property Graph (LPG)** développé en C#.
- Détection automatique et pondération des entités (NLP).
- Recherche par similarité (subgraph embedding, vectorisation texte).
- Rapidité sur grands volumes (plusieurs centaines de milliers de docs).

Bénéfices et limites

- Gain de temps, pertinence, efficacité de support client
- Sécurité du patrimoine informationnel
- Défis persistants de désilotage, capitalisation experte, automatisation des process

Exemple concret : aéronautique

Déploiement progressif chez un avionneur : de 15 000 utilisateurs internes à plus de 100 000 clients externes, grâce à la puissance du knowledge graph pour résoudre des problématiques techniques complexes.

Approche Capgemini « Trust before technology »

- Diagnostic métier et technique approfondi, co-construction humaine + technologique, prototypage rapide permettant d'incarner la valeur ajoutée.

Graphes et connaissance

- **Graphe orienté ou non orienté** : $G = (V, E)$ avec V = nœuds (entités), E = arêtes (relations).
- **Linked Property Graph (LPG)** : chaque nœud/arête peut avoir des propriétés clé-valeur.
- **Requête sur graphe** : trouver des sous-graphes isomorphes, calcul de chemins, recherche de motifs.

Sémantique, ontologies

- **Ontologie** : formalise les concepts et relations d'un domaine (ex : OWL, RDF Schema) ; utile pour donner du sens et permettre des inférences au-delà du simple graphe.

Vectorisation et recherche sémantique

- **Embeddings** : vectorisation des nœuds/document associés pour calculer la similarité (cosinus, euclidienne) :

$$\cos(\theta) = \frac{v_1 \cdot v_2}{\|v_1\|\|v_2\|}$$

- Techniques courantes : Word2Vec, GloVe, transformers (BERT/LLM).

IA et NLP

- Extraction d'entités : reconnaissance automatique des entités dans un texte.
- Similarité sémantique : utiliser des modèles de langage pour relier de nouveaux cas à l'historique.

Algorithmes fondamentaux

- Algorithme de parcours de graphe (DFS, BFS).
- Pagerank pour déterminer l'importance d'un nœud.
- Recherche de plus courts chemins (Dijkstra, A*).
- Recherche de sous-graphes similaires (isomorphisme de sous-graphe, graph embedding).

Exemple d'algorithme : BFS en C#

```
1  Queue<Node> queue = new Queue<Node>();
2  HashSet<Node> visited = new HashSet<Node>();
3  queue.Enqueue(startNode);
4
5  while (queue.Count > 0)
6  {
7      Node current = queue.Dequeue();
8      if (!visited.Contains(current))
9      {
10         visited.Add(current);
11         foreach (Node neighbor in current.Neighbors)
12         {
13             queue.Enqueue(neighbor);
14         }
15     }
16 }
```

Exemple de similarité cosinus entre deux embeddings

```
1  public double CosineSimilarity(double[] v1, double[]
       v2)
2  {
3      double dot = 0.0, mag1 = 0.0, mag2 = 0.0;
4      for (int i = 0; i < v1.Length; i++)
5      {
6          dot += v1[i] * v2[i];
7          mag1 += v1[i] * v1[i];
8          mag2 += v2[i] * v2[i];
9      }
10     return dot / (Math.Sqrt(mag1) * Math.Sqrt(mag2));
11 }
```

PageRank

Formule de base :

$$PR(u) = \frac{1-d}{N} + d \sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

avec d = facteur d'amortissement (classiquement 0.85), N = nombre total de nœuds.

Définition d'un nœud et d'une relation en C#

```
1  public class Node
2  {
3      public string Id { get; set; }
4      public Dictionary<string, object> Properties { get
           ; set; }
5  }
6
7  public class Relationship
8  {
9      public Node From { get; set; }
10     public Node To { get; set; }
11     public string Type { get; set; }
12     public Dictionary<string, object> Properties { get
           ; set; }
13 }
```

Recherche des voisins d'un nœud (extraction de sous-graphe)

```
1 public List<Node> GetNeighbors(Node node, List<
      Relationship> relationships)
2 {
3     return relationships
4     .Where(r => r.From == node)
5     .Select(r => r.To)
6     .ToList();
7 }
```

Pourquoi les graphes ?

- Représentation directe des entités et relations complexes (réseaux sociaux, routes, web, biologie).
- Résolution efficace des problématiques où les connexions comptent.
- Exemple : problème des sept ponts de Königsberg, modélisé par Euler (naissance de la théorie des graphes).

Concepts de base

- **Noeuds (sommets / vertices)**, **arêtes (edges)**, **propriétés**, **labels**
- Types : orienté / non orienté, pondéré / non pondéré, graphe cyclique / acyclique
- Modèles : adjacency list, adjacency matrix

Relationnel vs Graphe

- SQL : données en tables, relations via clés étrangères.
- Graphe : connexions directes entre entités, requêtes par traversée rapide.
- Avantages : absence d'index pour navigation, expressivité accrue pour les requêtes relationnelles complexes.

Index-free Adjacency (Neo4j)

Les relations sont stockées sous forme de pointeurs, optimisant l'accès et la navigation dans le graphe.

Le Property Graph Model

- Noeuds et relations portent des propriétés clé-valeur.
- Typage par labels (noeuds) et types (relations).

Exemple

```
(:Person {name: "Alex"})-[:KNOWS]->(:Person {name:
"Jordan"})
```

Cypher : Concepts clés

- Syntaxe déclarative et visuelle inspirée de l'ASCII art.
- `MATCH`, `WHERE`, `RETURN`, `CREATE`, `MERGE`, `DELETE`
- Exemples :
  - Requête basique :

```
MATCH (n:Person)-[:KNOWS]->(friend)
WHERE n.name = "Alex"
RETURN friend.name
```

  - Création :

```
CREATE (a:Person {name:"Alice"})
```

Comparaison Cypher, Gremlin, GQL

- **Cypher** : Déclaratif, standard Neo4j, pattern-matching, facile à apprendre.
- **Gremlin** : Orienté traversée (impératif), TinkerPop, multiplateforme, syntaxe en étapes (pipes).
- **GQL** (Graph Query Language) : Vers une normalisation ISO, synthèse de Cypher, PGQL et SQL/PGQ.
- **SPARQL** : Pour RDF, orienté graphe sémantique.

Transactions et Index

- ACID, gestion des transactions.
- Indexation des nœuds et propriétés.
- Contrôle d'accès : RBAC (Role-Based Access Control).
- Sauvegarde, restauration, haute disponibilité.

Jointures vs. Traversées de Graphe

- Jointure relationnelle devient : traversée directe de relations.
- Avantage énorme sur les requêtes multi-hop ou à profondeur variable.

Algorithmes et applications

- **Recherche de chemin :**
  - BFS, DFS, Dijkstra, A* (trouver le plus court chemin)
- **Centralité et importance :**
  - Degree, PageRank, Betweenness, Closeness
- **Détection de communautés :**
  - Louvain, Label Propagation
- **Similarité :**
  - Cosine, Jaccard, node embedding (Word2Vec pour graphes)

Chemin le plus court Cypher

```
1 MATCH (start:Person {name: "Alice"}), (end:Person {
      name: "Bob"}),
2 path = shortestPath((start)-[*]-(end))
3 RETURN path
```

Chemin le plus court Gremlin

```
g.V().has('name','Alice')
   .repeat(both().simplePath())
   .until(has('name','Bob'))
   .path()
   .limit(1)
```

Community Detection / Embeddings

- Community detection : séparation naturelle des sous-réseaux d'un graphe.
- Embeddings : vectorisation des nœuds pour le Machine Learning (proximité dans l'espace vectoriel encode la similarité structurelle).
- Applications : recommandation, analyse sociale, détection de fraudes.

Travail avec les données

- Import : CSV, JSON, connecteurs (Kafka, Spark, etc.).
- Export : mêmes formats, intégration BI et ML.
- APIs : Driver Python, Java, REST, Bolt.
- Focus sur la conversion de graphes relationnels (`LOAD CSV`, mapping tables $\rightarrow$ nœuds et relations).

Modélisation et bonnes pratiques

- Limiter le nombre de labels par noeud (recommandé : ≤ 4)
- Favoriser les pattern simples et typés
- Utiliser index et contraintes pour l'unicité/sécurité
- Tester la performance des requêtes (`EXPLAIN`/`PROFILE`, analyse des plans d'exécution)

Certification Neo4j / Ressources

- Certification officielle Neo4j Professional gratuite, accès en ligne.
- Sujets : Cypher, Modélisation, Sécurité, Algorithmes, Transactions.
- Plateformes de formation : Neo4j GraphAcademy, blog posts, documentation Neo4j.
- Pour aller plus loin : exploration de Gremlin (multi-db), comparaisons Cypher/Gremlin/SQL, utilisation de la GDS (Graph Data Science) Library pour l'IA.

Conclusion

Les graphes et bases de données orientées graphe sont au cœur des nouveaux challenges d'analyse et d'Intelligence Artificielle (exploiter les connexions, pas seulement les enregistrements). Cypher et Gremlin sont vos alliés pour explorer, analyser, et modéliser le réel numérique interconnecté.

Ressources complémentaires sur :
`https://graphacademy.neo4j.com/certifications/neo4j-certification/`
`https://data-xtractor.com/blog/category/graphs/`
`https://graphacademy.neo4j.com/courses/graph-data-science-fundamentals/`