INALCO M2 Master ingénierie multilingue 2025
Ingénierie des connaissances : Contrôle n°1 : Copie de Open AI Deep Research

1. Limites des plongements lexicaux pour stocker les connaissances humaines

Les **plongements lexicaux** (embeddings de mots ou de phrases) représentent les termes par des vecteurs dans un espace continu, capturant surtout des similarités d'usage. Bien qu'utiles en **NLP**, ils présentent plusieurs **limites pour stocker les connaissances humaines** :

- **Absence de structure explicite** : Les embeddings compressent l'information de manière distribuée. Ils n'encodent pas explicitement les relations logiques ou hiérarchiques entre concepts (par exemple, on ne distingue pas directement *chien* est un *animal*). Toute la connaissance est implicite dans des distances/similarités de vecteurs, ce qui n'équivaut pas à une base de connaissances structurée.
- **Difficulté à représenter des faits précis** : Deux entités proches dans l'espace de plongement partagent généralement un contexte sémantique, mais cela ne garantit pas la représentation d'un fait exact. Par exemple, savoir que *Paris est la capitale de la France* n'est pas simplement une question de proximité de vecteurs. Les embeddings peinent à stocker **des relations précises** (dates, propriétés uniques, etc.) et à effectuer du raisonnement logique.
- **Polyosémie et dépendance au contexte** : Un mot qui a plusieurs sens (ex. *python* animal vs. langage) aura un vecteur unique dans un modèle static (Word2Vec, GloVe), mélangeant les sens. Les plongements contextuels (BERT, etc.) atténuent ce problème en tenant compte du contexte, mais ils restent limités par les données d'entraînement et n'associent pas un concept à un identifiant unique comme le ferait une ontologie.
- **Schéma non modélisé et manque d'interprétabilité** : Contrairement à une base de connaissances avec un schéma défini (ontologie, base de données relationnelle), les embeddings n'ont pas de schéma lisible par l'homme. Il est **difficile d'interpréter** directement les dimensions d'un vecteur ou de comprendre pourquoi deux éléments sont proches. On ne peut pas facilement extraire une explication ou effectuer une requête précise (par exemple, "donne-moi tous les pays d'Europe et leur capitale").
- **Mise à jour complexe des connaissances** : Si l'on acquiert de nouvelles connaissances, il est non trivial de mettre à jour un modèle de plongements sans le réentraîner sur un corpus incluant ces nouveautés. En comparaison, ajouter une nouvelle donnée dans une base de connaissances structurée est plus direct. Les embeddings sont donc peu flexibles pour incorporer des connaissances humaines évolutives ou spécifiques.
- **Biais et couverture incomplète** : Les embeddings apprennent à partir d'un corpus (texte) et reflètent les biais et les lacunes de ce corpus. Des connaissances absentes du corpus ne seront pas présentes dans le vecteur. De plus, ils peuvent porter des biais culturels ou sémantiques. Pour **stocker toutes les connaissances humaines**, on est limité par la qualité et l'exhaustivité des textes d'entraînement, ce qui est loin d'être garanti.

En somme, les plongements lexicaux offrent une représentation numérique utile pour comparer des termes ou alimenter des modèles, mais ils ne constituent pas en soi un dépôt structuré et complet de

la connaissance humaine. Ils manquent de structure explicite, de fiabilité pour extraire des faits, et d'un mécanisme de mise à jour ou de requête précise, autant d'aspects nécessaires pour modéliser et **stocker la connaissance** de façon exploitable.

2. Étapes du framework OLAF et leur mise en œuvre technique

Le framework **OLAF** (Ontology Learning Applied Framework) est un ensem
philipperemy.github.io
es permettant de **construire automatiquement une ontologie** à partir de textes. D'après le schéma fourni, on peut identifier plusieurs étapes de traitement successives, chacune ayant un **rôle précis** et des solutions
philipperemy.github.io
our sa réalisation :

- **1) Extraction des termes (concepts candidats)** : Première étape consistant à identifier dans le corpus les termes clés du domaine qui deviendront possiblement des concepts de l'ontologie. Ce traitement sert à obtenir le vocabulaire de base.
  Réalisation technique : on peut utiliser des méthodes de **text mining** et de TALN pour extraire les termes nominaux significatifs. Par exemple, une analyse linguistique (POS-tagging, chunks) permet d'extraire les syntagmes nominaux fréquents. Des métriques statistiques (TF-IDF, **Term Extraction Algorithms** comme RAKE, YAKE, etc.) aident à filtrer les termes importants du domaine. Le résultat est une liste de termes candidats (souvent des mots simples ou composés) représentant des concepts.
- **2) Normalisation et enrichissement des concepts** : Une fois les termes extraits, il faut les normaliser et éventuellement les regrouper ou enrichir. Cette étape vise à unifier les variantes lexicales et préparer les concepts pour l'ontologie.
  Réalisation technique : on peut appliquer de la **lemmatisation** (réduction à la forme de base) et éliminer les doublons ou variantes (singulier/pluriel, acronymes). Des techniques de **détection de synonymes** peuvent être utilisées (par exemple en utilisant des plongements lexicaux pour regrouper des termes proches sémantiquement, ou en s'appuyant sur des ressources comme WordNet). L'objectif est de consolider la liste de concepts en évitant d'avoir plusieurs nœuds pour une même notion.
- **3) Extraction de la taxonomie (relations hiérarchiques "est-un")** : C'est le cœur de la construction de l'ontologie – établir des relations de spécialisation/généralisation entre les concepts (hiérarchie de classes). Chaque concept trouvé doit être positionné, si possible, dans une arborescence (par exemple *chien est un animal*).
  Réalisation technique : Classiquement, on utilise des **patterns lexico-syntaxiques** dans le texte, appelés motifs de Hearst, pour détecter des relations hiérarchiques. Par exemple, des phrases du type « X est un Y » ou « les Y comme X » indiquent que X est une sous-classe de Y. OLAF peut intégrer un module qui scanne le corpus à la recherche de telles expressions. Alternativement, des méthodes plus statistiques ou distributionnelles peuvent être employées (regrouper un terme sous un autre si le contexte suggère une relation d'inclusion). Techniquement, cela mobilise de l'**analyse syntaxique** (pour repérer des structures sujet-copule-attribut par exemple) combinée à des règles ou modèles entraînés pour reconnaître

les hyperonymes. Le résultat attendu est un **arbre hiérarchique** initial (ou un graphe) de concepts reliés par *est_un* (subClassOf en OWL).

- **4) Extraction des relations non taxonomiques** : En plus des liens hiérarchiques, on veut extraire d'autres relations sémantiques pertinentes du domaine (par exemple *X traite Y*, *X provoque Y*, *X fait partie de Y*, etc.). Ces relations enrichissent l'ontologie en ajoutant des propriétés entre concepts.
  Réalisation technique : On peut faire appel à des techniques d'**Open Information Extraction** (extraction libre d'information – voir question 4) pour obtenir des triplets (sujet, prédicat, objet) depuis les phrases du corpus. Une autre approche est d'utiliser l'**analyse de dépendances** : par exemple, repérer des verbes ou des prépositions qui lient deux termes extraits. On peut coupler l'analyse syntaxique à des règles ou à des classifieurs entraînés pour reconnaître certains types de relations cibles. Techniquement, cela implique du **parsing** des phrases, puis une recherche de motifs (p. ex. « [X] verbe [Y] » spécifique) ou l'utilisation d'outils existants d'extraction de relations. Chaque relation trouvée doit être interprétée et typée si possible (soit en la mappant sur une propriété existante
  philipperemy.github.io
  éant une nouvelle propriété ad hoc dans l'ontologie).

- **5) Construction de l'ontologie formelle (axiomatisation)** : Toutes les informations précédemment extraites (concepts, hiérarchie, relations) sont alors assemblées pour former l'ontologie au sens formel (par ex. en OWL/RDF). Le rôle de cette étape est de **traduire les connaissances extraites en axiomes ontologiques** exploitables par des machines (raisonneurs, SPARQL, etc.).
  Réalisation technique : On utilise des outils ou bibliothèques d'**ingénierie ontologique**. Par exemple, un module va parcourir la liste des classes (concepts) et créer dans un modèle OWL chaque classe. Ensuite, les relations *est_un* deviennent des assertions de sous-classe (OWL `rdfs:subClassOf`), et les relations non taxonomiques deviennent des *objets propriétés* liant deux classes (OWL `ObjectProperty`, avec éventuellement des domaines/portées renseignés). OLAF, par sa conception modulaire, peut intégrer ici une **transformation automatisée en OWL** : on peut coder des règles de mappage qui convertissent un triplet texte en triplet RDF. Par exemple, si on a détecté "X part_of Y", on crée une propriété `partOf` reliant les classes X et Y. Techniquement, des frameworks comme OWL API (en Java) ou RDFLib (en Python) peuvent être utilisés pour générer le fichier OWL. Une fois l'ontologie construite, il est possible de la passer dans un **raisonneur** automatique pour vérifier sa cohérence logique et éliminer d'éventuelles contradictions.

- **6) Validation et itération** (optionnel dans OLAF) : Cette étape consiste à évaluer la qualité de l'ontologie apprise et éventuellement itérer pour l'améliorer. Son rôle est de **s'assurer de la pertinence** des connaissances ajoutées.
  Réalisation technique : On peut comparer l'ontologie générée à une ontologie de référence s'il en existe une, ou bien solliciter un expert humain pour valider les concepts et relations. D'un point de vue automatique, comme mentionné, un **raisonneur** pe
  aclanthology.org
  s incohérences (par ex. une classe qui serait sous-classe de deux classes disjointes). OLAF peut implémenter une boucle où, en cas d'incohérence, certaines axiomes faibles sont supprimés jusqu'à obtenir une base cohérente【7†L47-L52】. Par ailleurs, des métriques de

qualité (richesse, spécificité, couverture du corpus) peuvent être calculées. Sur le plan technique, l'évaluation peut aussi impliquer des tests manuels sur des requêtes SPARQL pour voir si l'ontologie répond correctement à des questions cibles.

En résumé, OLAF est un framework modulaire couvrant tout le pipeline d'**ontology learning** depuis le texte brut jusqu'à l'ontologie finalisée. Chaque étape (extraction des termes, structuration hiérarchique, extraction de relations, formalisation OWL, etc.) remplit une fonction spécifique dans la construction des connaissances, et s'appuie sur des techniques NLP ou des algorithmes connus. En combinant ces modules, OLAF vise à automatiser autant que possible la création d'ontologies dans n'importe quel domaine à partir de corpus textuels.

3. Les LLMs pour automatiser la création d'ontologies

À première vue, les **LLMs** (Large Language Models) pourraient aider à générer automatiquement des ontologies, car ils comprennent le langage naturel et disposent de connaissances vastes. Par exemple, on peut les solliciter pour extraire des triplets (concept-relation-concept) à partir de textes, proposer des définitions de classes, ou compléter une taxonomie. Des approches récentes ont expérimenté la génération d'axiomes OWL via des prompts en langage naturel ou l'extraction de relations par LLM, avec un certain succès initial.

Cependant, les LLMs ne sont pas une solution magique ni complète pour construire des ontologies de qualité :

- **Véracité et cohérence** : Un LLM peut « halluciner » des relations qui semblent plausibles mais qui sont fausses. Il n'y a pas de garantie que les connaissances générées soient exactes ou logiquement cohérentes dans l'ensemble. Par exemple, le modèle pourrait inventer qu'un concept est sous-classe d'un autre à tort. Les ontologies exigent de la **précision** et le respect de contraintes logiques (pas de cycles illogiques, etc.), choses que les LLMs n'assurent pas d'eux-mêmes.
- **Contrôle du schéma** : Automatiser la création d'une **ontologie formelle** requiert le respect de syntaxe et de règles (OWL, RDF). Un LLM brut peut produire un texte ou du code OWL, mais il peut facilement violer des restrictions de l'ontologie (par ex., faire d'une instance une classe). Sans cadre ou post-traitement, la sortie d'un LLM risque de ne pas être un OWL valide ou consistant.
- **Dépendance aux données d'entraînement** : Les LLMs reflètent les données sur lesquelles ils ont été entraînés. S'il s'agit d'un domaine très spécifique, le LLM peut manquer d'information ou utiliser des connaissances génériques inadaptées. **L'adaptation au domaine** est nécessaire (via fine-tuning éventuellement), ce qui redevient un effort significatif. De plus, un LLM peut introduire des biais présents dans les données d'entraînement, biais qui se retrouveraient dans l'ontologie produite.
- **Difficulté à évaluer et affiner** : Une ontologie générée automatiquement devrait idéalement être examinée par un expert humain. Si on automatise entièrement avec un LLM, on risque de devoir corriger manuellement a posteriori de nombreuses erreurs subtiles, ce qui réduit l'intérêt d'une automatisation complète. En pratique, les méthodes d'**ontology learning** ont

toujours une intervention humaine pour valider/régler les ambiguïtés – les LLMs ne font pas exception.

En conclusion, **les LLMs peuvent être utiles comme assistants** dans le processus (pour suggérer des relations, accélérer l'extraction de connaissances à partir de texte, etc.), mais ils ne constituent pas encore une solution suffisante pour **automatiser de bout en bout la création d'ontologies**. Il est probable qu'une approche hybride soit la plus efficace : utiliser les LLMs pour proposer des candidats (classes
milvus.io
ns) puis valider et organiser ces propositions via des algorithmes symboliques et l'expertise humaine. Cela permet de tirer parti de la puissance des LLMs en réduisant le risque d'erreurs ontologiques graves.

4. Open IE en TALN : définition, techniques, évaluation, exemples, protocole

Open IE (*Open Information Extraction*) est une technique de Traitement du Langage Naturel qui vise à **extraire des faits structurés (triplets)** sujet–relation–objet à partir de texte lib
milvus.io
voir défini à l'avance une liste de relations cibles. Autrement dit, le système doit découvrir les relations exprimées dans le texte de manière non supervisée et sans schéma prédéfini【16†L14-L22】. Par exemple, à partir de la phrase *« Barack Obama was born in Hawaii »*, un système Open IE extrait typiquement le triplet *(Barack Obama; was born in; Hawaii)*, correspondant à la relation ouverte "was born in" (né à)【16†L14-L22】.

Comment réaliser techniquement l'Open IE ? Historiquement, les premières approches Open IE utilisaient des **patrons lexicaux et syntaxiques**. Elles procédaient en plusieurs étapes :

1. Analyser la phrase (segmentation en clauses, **parsing** éventuel).
2. Identifier une expression candidate pour être la relation (souvent un verbe principal ou une ph
   milvus.io
   le). Par exemple, dans *« X est né à Y »*, l'expression "est né à" joue le rôle de prédicat.
3. Identifier les arguments de cette relation (le sujet X et l'objet Y dans l'exemple).
4. Restituer le triplet (X; est né à; Y).

Des systèmes marquants comme **TextRunner**, puis **ReVerb** et **OLLIE**, ont implémenté ces idées en anglais : ReVerb utilisait par exemple des motifs regex sur des tags POS pour reconnaître les relations (suite de verbes, prépositions, etc.) de façon fiable. OLLIE allait plus loin en utilisant le résultat d'un analyseur de dépendances pour extraire des relations plus complexes (y compris implicites).

Plus récemment, avec l'essor du deep learning, des approches apprises (**supervisées**) ont émergé pour l'Open IE. On a formulé la tâche comme un problème de séquence-à-séquence ou d'étiquetage de séquences : le modèle neuronal apprend à produire directement des triplets à partir de la phrase, ou bien à étiqueter chaque mot comme faisant partie de l'argument 1, de la relation ou de l'argument 2. Ces méthodes utilise

spiceworks.com
le des **transformers** entraînés sur des corpus annotés pour Open IE, ou exploitent les LLMs en les incitant (prompt) à reformuler une phrase en un ensemble de triplets. L'avantage est d'être plus générique, mais elles nécessitent des données d'entraînement et peuvent souffrir de généralisation hasardeuse hors domaine.

Difficulté de l'évaluation : L'Open IE est notoire pour être **difficile à évaluer** de manière automatique. Contrairement à une tâche supervisée classique, il n'y a pas un ensemble fini de relations prédéterminées, donc pas de référence claire pour chaque phrase. On peut comparer aux triples d'un corpus annoté manuellement, mais la création de ce **jeu de vérité terrain** est fastidieuse et ambiguë (il peut y avoir plusieurs manières correctes d'exprimer le même fait). De plus, les systèmes Open IE produisent souvent des variantes (par ex. fragmenter une relation en deux sub-triplets). Ainsi, les métriques de précision et rappel doivent être interprétées avec prudence. Souvent, l'évaluation implique une part de jugement humain : on prend un échantillon de triplets extraits et on vérifie manuellement s'ils sont corrects, afin d'estimer une précision. Pour le rappel, on peut vérifier si les principaux faits d'un texte ont été extraits. Globalement, **évaluer Open IE** est difficile car il faut décider quels triplets *auraient dû* être extraits, ce qui dépend du contexte et de l'utilité visée.

Exemples de résultats Open IE :
– *« Marie Curie découvre le radium en 1898. »* ⟶ (Marie Curie; découvre; le radium), (découverte; en; 1898).
– *« La tour Eiffel se situe à Paris. »* ⟶ (La tour Eiffel; se situe à; Paris).
Ces exemples illustrent que le même texte peut donner plusieurs triplets (notamment si la phrase comporte des compléments circonstanciels comme les dates). Un bon système Open IE doit extraire les informations nucléaires (ici la relation de découverte, la relation de localisation).

Protocole possible pour l'Open IE : Pour implémenter et évaluer une solution Open IE de manière rigoureuse, on peut proposer le protocole suivant :

1. **Choisir un corpus de test** dans le domaine d'intérêt. Idéalement, disposer de quelques phrases annotées manuellement avec les triplets attendus (jeu de référence) pour l'évaluation.
2. **Appliquer un outil Open IE** (existant ou développé en interne) sur ce corpus. Par exemple, utiliser un wrapper Python du Stanford OpenIE【16†L23-L27】 ou un modèle de transformation fine-tuné pour l'extraction ouverte.
3. **Post-traiter les triplets** extraits : éventuellement filtrer ceux qui sont incomplets ou trop génériques. On peut aussi normaliser les entités (résolution de coréférence, nettoyer les libellés identiques).
4. **Évaluer la qualité** : comparer les triplets extraits aux triplets de référence si disponibles. Calculer une précision (pourcentage de triplets extraits qui sont corrects et pertinents) et un rappel (pourcentage des faits attendus qui ont été couverts par l'Open IE). Si pas de référence établie, faire évaluer par des experts humains un échantillon de sorties.
5. **Analyse d'erreurs** : catégoriser les erreurs (faux positifs : relations mal identifiées, arguments mal extraits, etc., et faux négatifs : informations non extraites). Cette analyse

permettra d'améliorer l'outil (par ex., ajouter un patron pour un type de phrase mal géré, ou entraîner un modèle sur plus d'exemples).

6. **Itérations d'amélioration** : affiner le système Open IE d'après les observations (ajustement de paramètres, enrichissement de la base de règles ou entraînement complémentaire du modèle). Réévaluer jusqu'à obtention d'un niveau satisfaisant.

En résumé, l'Open IE est une tâche puissante pour peupler des connaissances à partir de textes non structurés, **techniquement réalisable** via des méthodes à base de règles ou d'apprentissage profond. Elle n'est pas triviale à évaluer, et un protocole soigné impliquant une validation humaine est souvent nécessaire pour garantir la qualité des faits extraits.

5. Joint training : principe, utilité et mise en œuvre

Le **joint training** (apprentissage conjoint) est une approche d'entraînement de modèles où l'on entraîne simultanément plusieurs tâches ou objectifs sur un même modèle, au lieu de les entraîner séparément. L'idée est de **partager certaines composantes du modèle** entre plusieurs tâches de façon à exploiter les points communs et synergies entre ces tâches【21†L145-L152】. En pratique, cela correspond souvent à la **multi-tâche** (*multi-task learning*) où un même réseau neuronal a des couches partagées qui apprennent des représentations utiles à tous les objectifs, et des couches finales spécifiques à chaque tâche.

Utilité : Le joint training offre plusieurs avantages potentiels. D'abord, il peut améliorer la **généralisation** : apprendre plusieurs tâches reliées en parallèle fait office de régularisation, le modèle évite de sur-spécialiser sur une seule tâche et capture des caractéristiques plus abstraites utiles. Par exemple, entraîner conjointement une tâche de reconnaissance d'entités nommées et une tâche de classification de relations peut aider chaque tâche, car identifier correctement les entités peut aider à trouver les relations et vice-versa. De même, en vision par ordinateur, on peut entraîner ensemble la détection d'objets et la segmentation : les deux compétences se renforcent mutuellement. Le joint training est aussi **économe** : au lieu d'entraîner deux modèles séparés, on n'en entraîne qu'un seul, ce qui peut nécessiter moins de données globalement (les données de tâches différentes se complètent) et moins de ressources de calcul à l'inférence (un seul modèle multi-tâche déployé). Enfin, cela peut être utile lorsque les tâches ont un **lien hiérarchique** : par exemple, entraîner un modèle de traduction dans plusieurs langues en même temps permet de partager une partie du lexique et des constructions communes.

Mise en œuvre pratique : Pour réaliser un joint training, on doit concevoir une architecture de modèle qui comporte des composantes partagées et possiblement des sorties multiples. Concrètement, on peut avoir un **réseau à couches communes** (par ex. des couches profondes partagées) puis des couches de sortie dédiées à chaque tâche. Durant l'entraînement, on définit une fonction de coût qui combine les erreurs de chaque tâche (par exemple somme pondérée des losses de chaque tâche). À chaque itération, on peut soit alterner les mini-lots de données de chaque tâche, soit mélanger les données de toutes les tâches dans un même lot si c'est applicable. Le modèle met ainsi à jour ses poids pour **minimiser l'erreur globale multi-tâches**, ce qui force les couches partagées à servir au mieux toutes les tâches.

Un exemple concret en NLP : on peut entraîner conjointement l'étiquetage grammatical (Part-of-Speech tagging) et la reconnaissance d'entités nommées. Le réseau partage ses embeddings de mots et ses couches bi-LSTM, puis a deux têtes de sortie, l'une prédisant le POS de chaque mot, l'autre

prédisant s'il fait partie d'une entité nommée. On calcule la loss totale = loss_POS + loss_NER, et on entraîne. Il a été montré que ce **apprentissage conjoint** améliore souvent la performance sur les entités nommées, surtout quand les données annotées en entités sont limitées, car le POS fournit

aire utile.

En résumé, le joint training consiste à **entraîner simultanément plusieurs tâches en partageant des paramètres**, ce qui est utile pour exploiter des similarités, réduire le sur-apprentissage et économiser des ressources. La mise en œuvre se fait via une architecture partagée et une fonction de perte multi-objectifs, et nécessite de bien équilibrer les contributions de chaque tâche (choix des poids de loss, etc.) pour que aucune tâche ne domine ou ne soit négligée pendant l'apprentissage.

6. Limitations des bases SQL et introduction aux bases NoSQL

Limites des bases de données SQL (relationnelles) : Les SGBDR (Systèmes de gestion de bases de données relationnelles) comme MySQL, PostgreSQL, Oracle, etc., sont très efficaces pour les données tabulaires structurées, mais ils présentent certaines limites bien connues lorsqu'il s'agit de gérer les connaissances à grande échelle ou très variées :

- **Passage à l'échelle horizontal difficile** : Les bases SQL sont conçues historiquement pour être déployées sur un seul serveur (scale-up). Pour gérer plus de données ou plus de trafic, il faut un serveur plus puissant. Le **sharding** (répartition sur plusieurs nœuds) n'est pas natif dans la plupart des SQL traditionnels, ce qui complique la distribution. Cela les rend moins flexibles pour le Big Data massif ou les applications web à très haute charge par rapport à certains NoSQL pensés pour le **scale-out** (ajout de serveurs).【36†L1-L4】
- **Rigidité du schéma** : Un schéma SQL doit être défini à l'avance (tables, colonnes avec types). Toute modification (ajouter une colonne, changer un type) peut être coûteuse en production. Cette rigidité pose problème quand les données évoluent rapidement ou sont semi-structurées. Par exemple, stocker des données aux formats variés (JSON, multimédia avec métadonnées variables) est mal aisé en SQL (même si des champs JSON existent désormais, on perd en performance de requête).【36†L1-L4】
- **Données hiérarchiques ou fortement reliées** : Représenter des hiérarchies ou des graphes dans un modèle tabulaire nécessite des jointures multiples et complexes. Les **requêtes avec de nombreuses jointures** peuvent devenir lentes à mesure que les tables grossissent. Par exemple, pour des données de type réseau social (graphes d'amis, de followers), une base SQL montre vite ses limites en complexité de requêtage.
- **Données non structurées** : Les SGBD relationnels ne gèrent pas directement les documents texte, les données brutes (images, etc.) ou les données sans format fixe. On doit les stocker sous forme de BLOB, sans possibilité de les requêter facilement. Ce qui signifie qu'on ne profite pas de la base pour autre chose que du stockage, sans la richesse de requêtes. 【36†L1-L4】

En somme, les bases SQL excellent pour la **cohérence ACID** des transactions et les données bien structurées, mais souffrent de **limitations de scalabilité, de flexibilité de schéma et d'adaptabilité à des données complexes ou non structurées**.

Bases NoSQL : Le terme *NoSQL* signifie "Not Only SQL". Il désigne une famille de bases de données non relationnelles, conçues pour pallier certaines limites ci-dessus. Elles stockent les données dans des formats plus flexibles que les tableaux de relations. En général, les bases NoSQL se caractérisent par :

- un **schéma flexible ou inexistant** (chaque enregistrement peut avoir des champs différents),
- une capacité de **scalabilité horizontale** plus facile (distribution sur plusieurs nœuds, tolérance à la partition),
- souvent un sacrifice de la stricte cohérence ACID pour obtenir une plus grande disponibilité ou performance (modèle CAP : beaucoup de NoSQL préfèrent une cohérence éventuelle).

Il existe **plusieurs types de bases NoSQL** selon le modèle de données qu'elles adoptent【24†L73-L81】:

1. **Bases orientées documents** : Elles stockent des documents semi-structurés (généralement au format JSON ou BSON). Chaque document peut avoir sa propre structure, et on peut indexer les champs JSON. C'est adapté pour des données hiérarchiques ou des agrégats complets. *Exemples* : MongoDB, CouchDB. Dans MongoDB, on peut stocker un document complet (par ex. la fiche d'un produit avec champs variables) sans le normaliser en plusieurs tables.
2. **Bases key-value (clé-valeur)** : Les données sont stockées sous forme de paires clé → valeur, un peu comme une grande table de hachage distribuée. La valeur est opaque pour la base (blob ou sérialisation), la base ne gère que le recouvrement par la clé et éventuellement des opérations simples. Très performant pour récupération par clé et stockage de sessions, caches, etc. *Exemples* : Redis (en mémoire), Riak, DynamoDB (d'Amazon). Cela convient pour stocker des sessions utilisateur, des préférences, où la clé est un ID et la valeur un objet arbitraire.
3. **Bases en colonnes (wide column stores)** : Inspirées de Google BigTable, elles stockent les données par blocs de colonnes. Chaque entrée est identifiée par une clé de ligne, mais les colonnes sont groupées en familles et peuvent varier par ligne. C'est efficace pour les **très grandes tables sparses** et les analyses distribuées. *Exemples* : Apache Cassandra, Apache HBase. On les utilise souvent pour des données de séries temporelles ou de logs distribués.
4. **Bases orientées graphes** : Conçues pour stocker des entités et des relations de manière native, elles représentent les données sous forme de nœuds et arêtes. On peut ainsi naviguer très rapidement d'un nœud à l'autre sans effectuer de multiples jointures. Elles supportent des requêtes de graphes (par ex. chemin le plus court, voisinage, motifs) via des langages spécialisés (Gremlin, Cypher). *Exemples* : Neo4j, OrientDB, JanusGraph. Un cas d'usage est le **knowledge graph** ou réseau social, où ces bases excelllent à requêter des liens complexes (amis d'amis, recommandations…).

Chacun de ces types NoSQL est adapté à des cas d'utilisation spécifiques. Par exemple, pour de la **grande volumétrie** de données peu structurées, une base documents ou colonne est appropriée; pour des **relations très interconnectées**, une base graphes est toute désignée. À noter que "NoSQL" n'implique pas absence de langage de requête : beaucoup de NoSQL proposent un langage propre ou des API REST, et certaines (p. ex. Cassandra) ont un SQL-like. L'important est

qu’elles s’éloignent du modèle tabulaire rigide pour gagner en flexibilité et scalabilité. En pratique, **SQL et NoSQL coexistent** : SQL reste excellent pour les transactions structurées (ex : données financières), tandis que NoSQL brille pour le big data hétérogène ou distribué. Le choix dépend donc des besoins de l’application (cohérence forte vs disponibilité, structure vs flexibilité, etc.).

7. Étapes de la Data Preparation et définition d’un connecteur

La **Data Preparation** consiste à transformer des données brutes en données prêtes à l’emploi (pour l’analyse, l’apprentissage machine ou l’alimentation de systèmes). Les étapes principales de préparation des données et leur utilité sont :

1. **Collecte et intégration des données** : Rassembler les données depuis leurs différentes sources (fichiers CSV, base de données, API, etc.) et les combiner. *Utilité* : Obtenir en un seul endroit toutes les données nécessaires. Cela peut impliquer de fusionner plusieurs jeux de données, faire correspondre des enregistrements (par ex. même entité apparaissant dans deux sources) et convertir les formats de fichiers. À cette étape, un **connecteur** (défini ci-dessous) intervient souvent pour accéder à chaque source.
2. **Nettoyage des données** : Identifier et corriger les données de mauvaise qualité. *Utilité* : Améliorer la fiabilité de l’analyse en supprimant les erreurs. Concrètement, on va gérer les valeurs manquantes (les remplir via une méthode, ou éliminer les enregistrements incomplets), supprimer les duplicatas, corriger les incohérences (par ex., normaliser l’orthographe des catégories, rectifier les outliers aberrants s’il y en a, vérifier les contraintes d’intégrité). Le nettoyage assure que la base est *propre* et cohérente.
3. **Transformation et formatage** : Convertir les données dans une forme appropriée à l’usage futur. *Utilité* : Uniformiser et enrichir les données pour qu’elles soient exploitables par les outils en aval (algorithmes de ML, entrepôt de données, etc.). Cela inclut la **normalisation** (ex: mettre toutes les dates au même format, transformer des unités, standardiser la casse du texte), la **mise en type** (s’assurer que les nombres sont bien numériques, etc.), et éventuellement la **création de nouvelles variables** plus utiles (feature engineering simple, comme extraire l’année d’une date, ou catégoriser une valeur continue en tranche). Dans un contexte multilingue, cela peut impliquer de traduire certains champs ou de les translittérer.
4. **Réduction et sélection des données** : (Selon le cas d’usage) Diminuer la masse de données tout en gardant l’essentiel. *Utilité* : Accélérer les traitements et éviter le bruit. Il peut s’agir de **sélectionner des features** pertinentes (éliminer des colonnes non informatives pour le modèle), d’**échantillonner** une partie des données si le volume est trop grand, ou de **compresser** certaines informations (par exemple, regrouper par catégories). En apprentissage automatique, on sépare aussi les données en ensembles d’entraînement, validation, test à cette étape, ce qui fait partie de la préparation.

Après ces étapes, les données sont prêtes, dans un format cohérent et de qualité, pour être exploitées avec confiance.

Connecteur : Un connecteur de données est un **module d’interface** qui permet à un système d’accéder à une source de données particulière et de la *faire entrer* dans le pipeline de préparation ou d’analyse. En d’autres termes, c’est un composant logiciel qui se charge de la connexion et de l’extraction des données depuis un format ou un emplacement donné, en masquant les détails de

cette source. Par exemple, un connecteur JDBC pour bases SQL permet de se connecter à une base de données relationnelle et d'en lire les tables via des requêtes. Un connecteur pour fichiers CSV saura lire un fichier délimité et fournir les enregistrements au format interne (tableau, DataFrame...). On trouve des connecteurs pour à peu près tout : API web (connecteur REST/JSON), systèmes Big Data (connecteur Hadoop/Hive), flux en temps réel, etc. **L'utilité d'un connecteur** est de simplifier l'ingestion des données : au lieu d'écrire du code personnalisé pour chaque source, on utilise un connecteur standard qui gère l'authentification, la lecture, la conversion de format, et parfois des optimisations (lecture par paquets, reprise sur erreur, etc.). Dans un workflow ETL (Extract, Transform, Load), les connecteurs réalisent le "Extract". En résumé, un connecteur assure la **portabilité et l'automatisation** de l'accès aux données, ce qui est crucial dans la phase de préparation pour enchaîner ensuite sur le nettoyage, la transformation, etc., sans se soucier des spécificités de chaque source.

8. Quatre manières de représenter un graphe en informatique

【37†embed_image】 *Graphique : Exemple de graphe à six nœuds (A–F) utilisé pour illustrer les différentes représentations.* Ce graphe comporte des sommets nommés A, B, C, D, E, F et des arêtes non orientées entre certains d'entre eux (par exemple A–B, A–C, B–D, B–E, C–F, E–F). On va le représenter de **quatre façons différentes** :

1. **Liste d'adjacence** – Pour chaque nœud, on stocke la liste de ses voisins (les sommets adjacents connectés par une arête) :

   ```
   plaintext
   CopierModifier
   A : B, C
   B : A, D, E
   C : A, F
   D : B
   E : B, F
   F : C, E
   ```

   Ici, chaque ligne donne un sommet suivi de la liste des sommets directement liés. Par exemple, `B : A, D, E` indique que les arêtes connectant B vont vers A, D et E. Cette représentation est **économique en mémoire** pour les graphes clairsemés, et permet de parcourir efficacement les voisins d'un nœud.

2. **Matrice d'adjacence** – Il s'agit d'une matrice (tableau 2D) de dimension $N \times N$ si le graphe a N sommets, où l'entrée (i,j) indique la présence d'une arête entre le sommet i et le sommet j. On numérotera les sommets A=0, B=1, C=2, D=3, E=4, F=5 pour construire la matrice :

	A	**B**	**C**	**D**	**E**	**F**
A	0	1	1	0	0	0
B	1	0	0	1	1	0
C	1	0	0	0	0	1
D	0	1	0	0	0	0
E	0	1	0	0	0	1
F	0	0	1	0	1	0

   Un **1** indique qu'une arête existe entre la ligne et la colonne correspondante, **0** indique l'absence d'arête. Par exemple, la ligne A comporte des 1 en colonne B et C, reflétant les

arêtes A–B et A–C. La matrice est symétrique ici car le graphe est non orienté (si le graphe était orienté, on mettrait 1 uniquement à [i,j] pour une arête i → j). Cette représentation facilite certaines opérations algébriques et la détection rapide de la connexion entre deux nœuds (accès direct à la case [i,j]), mais elle est coûteuse en mémoire pour les grands graphes creux (beaucoup de zéros si le graphe a peu d'arêtes par rapport au nombre de sommets).

3. **Liste d'arêtes** – Il s'agit tout simplement de la liste de toutes les arêtes du graphe, chacune étant un couple (ou tuple) de sommets. Pour notre graphe, on peut lister les arêtes non orientées comme suit :

   ```
   plaintext
   CopierModifier
   A–B, A–C, B–D, B–E, C–F, E–F
   ```

   Chaque entrée représente une connexion entre deux nœuds. On peut aussi ajouter un troisième élément si les arêtes sont pondérées (par ex. (A, B, poids)). Cette représentation est très simple et utile pour, par exemple, implémenter des algorithmes de parcours d'arêtes (comme Kruskal pour le calcul de l'arbre couvrant minimal). Elle est peu redondante (chaque arête n'est listée qu'une fois), mais ne donne pas directement le voisinage d'un nœud sans parcourir toute la liste.

4. **Matrice d'incidence** – Cette matrice représente la relation entre les sommets et les arêtes. Si l'on a N sommets et M arêtes, c'est une matrice N×M où l'entrée (i, j) indique si le sommet i est incident à (c'est-à-dire connecté par) l'arête j. On doit d'abord nommer les arêtes : prenons par exemple e1=A–B, e2=A–C, e3=B–D, e4=B–E, e5=C–F, e6=E–F. La matrice d'incidence serait :

Sommet	e1 (A–B)	e2 (A–C)	e3 (B–D)	e4 (B–E)	e5 (C–F)	e6 (E–F)
A	1	1	0	0	0	0
B	1	0	1	1	0	0
C	0	1	0	0	1	0
D	0	0	1	0	0	0
E	0	0	0	1	0	1
F	0	0	0	0	1	1

   Ici, un **1** signifie que le sommet de la ligne est l'un des deux extrémités de l'arête de la colonne. Par exemple, la colonne e4 (B–E) a des 1 sur la ligne B et la ligne E, indiquant que l'arête e4 relie B et E. Cette représentation est moins utilisée en pratique, mais utile pour certaines formulations mathématiques (par exemple en théorie des graphes pour écrire des équations d'équilibrage de flux, etc.). Elle est assez redondante (chaque arête non orientée correspond à deux "1"). Son intérêt est plus théorique, ou pour des graphes très denses où M est proche de N^2, elle revient presque à la même complexité qu'une matrice d'adjacence.

En résumé, les **listes d'adjacence** et **listes d'arêtes** sont privilégiées en algorithmique pour leur simplicité et leur légèreté, tandis que la **matrice d'adjacence** est utile pour les petites structures ou les calculs matriciels sur les graphes, et la **matrice d'incidence** sert dans des contextes spécifiques. Chaque représentation a ses cas d'usage selon qu'on optimise la mémoire, la rapidité d'accès aux voisins, ou la facilité d'implémentation.

9. Implémentation de l’algorithme DFS (Depth-First Search) en Python

La **recherche en profondeur** (DFS) est un algorithme de parcours de graphe qui explore autant que possible le long d’une branche avant de revenir en arrière pour explorer les alternatives. Pour un graphe donné sous forme de liste d’adjacence, on peut implémenter la DFS de manière récursive ou itérative. Dans le contrôle, le graphe était fourni sous forme de dictionnaire Python, par exemple :

```
python
CopierModifier
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}
```

Voici une **implémentation en Python (récursive)** de la DFS pour un graphe non orienté représenté par un tel dictionnaire de listes d’adjacence :

```
python
CopierModifier
def dfs(graph, start, visited=None):
    if visited is None:
        visited = []            # liste pour conserver l'ordre de visite
    visited.append(start)       # marquer le nœud courant comme visité
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
    return visited

# Exemple d'utilisation :
visited_nodes = dfs(graph, 'A')
print(visited_nodes)  # Exemples de sortie possible: ['A', 'B', 'D', 'E', 'F',
'C']
```

Explication : la fonction `dfs` prend le graphe et un nœud de départ `start`. On utilise une liste `visited` pour suivre les sommets visités (on l’initialise à la première appel). On ajoute le sommet courant à la liste, puis pour chacun de ses voisins non encore visité, on appelle récursivement `dfs`. Ainsi, la recherche s’enfonce en profondeur tant qu’il y a un nouveau voisin à visiter. Lorsqu’elle ne trouve plus de nouveau voisin, la récursion se désempile et reprend à la dernière bifurcation laissée. Au final, `visited` contient l’ordre dans lequel les sommets ont été parcourus en profondeur.

Dans l’exemple, en partant de 'A', on visitera 'B', puis de 'B' on ira vers 'D' (puis remonte), puis 'E' puis 'F', puis de 'F' vers 'C', etc., produisant un ordre de visite comme `A, B, D, E, F, C`. Cet ordre peut varier légèrement selon l’ordre des voisins dans les listes (ici on a exploré 'B' avant 'C' depuis 'A'). L’important est que chaque sommet atteignable est visité une et une seule fois. La complexité de DFS est linéaire O(N + M) (N sommets, M arêtes), et l’implémentation ci-dessus exploite implicitement la pile d’appels Python pour la pile de parcours. Une version itérative pourrait utilis

spiceworks.com
ile (`stack`) mais suit le même principe.

10. Problème des routes magiques (Lothlórien) et algorithme de Prim

a) Objet mathématique à construire : Le problème décrit (relier des cités cachées par un réseau de routes magiques de sorte que toutes soient connectées tout en minimisant la longueur totale) correspond exactement à la recherche d'un **arbre couvrant minimal** (Minimum Spanning Tree, MST) dans un graphe pondéré. En langage mathématique, il faut modéliser la Terre du Milieu avec un **graphe** dont les nœuds sont les cités elfiques et les arêtes représentent les routes envisageables entre cités, pondérées par leur longueur. La solution du problème est alors un sous-ensemble d'arêtes formant un **arbre couvrant** (qui connecte tous les nœuds sans cycle) de poids total minimal. Construire l'arbre couvrant minimal répond à la contrainte "toutes les cités restent connectées" (propriété d'arbre couvrant) et "minimiser la longueur totale" (propriété de minimalité du poids).

En résumé, la formulation mathématique est : *Soit $G = (V, E)$ le graphe des cités (V) et routes potentielles (E) avec une fonction de poids $w(e)$ = longueur de la route. Trouver $T \subseteq E$ tel que (V, T) forme un arbre couvrant de poids $\Sigma_{e \in T} w(e)$ minimal.* Cet arbre couvrant minimal est l'objet à construire.

b) Implémentation de l'algorithme de Prim en Python : L'algorithme de **Prim** est une méthode gloutonne pour trouver un arbre couvrant minimal. Il démarre d'un sommet arbitraire et ajoute petit à petit les arêtes les moins coûteuses menant à un nouveau sommet non encore inclus, jusqu'à ce que tous les sommets soient connectés.

On suppose en entrée une **liste d'arêtes** pondérées et un sommet de départ (pivot). Chaque arête peut être représentée comme un triplet `(u, v, w)` avec `u` et `v` les deux villes reliées et `w` la longueur (poids). En sortie, on renverra la liste des arêtes choisies formant le MST. Voici une implémentation possible :

```
python
CopierModifier
def prim(edges, start):
    # edges: liste de triplets (u, v, w)
    # start: sommet de départ (ex: 'A')
    # Extraction de la liste des sommets à partir des arêtes
    vertices = set()
    for u, v, w in edges:
        vertices.add(u); vertices.add(v)
    if start not in vertices:
        return []  # sommet de départ invalide

    visited = {start}
    mst = []
    # Tant qu'il reste des sommets non visités
    while visited != vertices:
        min_edge = None
        for (u, v, w) in edges:
            # arête avec une extrémité visitée et l'autre non visitée
            if (u in visited and v not in visited) or (v in visited and u not in
visited):
                if min_edge is None or w < min_edge[2]:
```

```
                    min_edge = (u, v, w)
        if min_edge is None:
            break  # le graphe n'est pas connexe (pas d'arête joignant les
composants)
        # Ajouter l'arête minimale trouvée
        mst.append(min_edge)
        # Marquer le nouveau sommet comme visité
        visited.add(min_edge[0]); visited.add(min_edge[1])
    return mst

# Exemple d'utilisation :
edges_list = [
    ('A', 'B', 7), ('A', 'D', 5),
    ('B', 'C', 8), ('B', 'D', 9), ('B', 'E', 7),
    ('C', 'E', 5),
    ('D', 'E', 15), ('D', 'F', 6),
    ('E', 'F', 8), ('E', 'G', 9),
    ('F', 'G', 11)
]
result = prim(edges_list, 'A')
print(result)
# Sortie possible (arbre couvrant minimal) :
# [('A', 'D', 5), ('D', 'F', 6), ('F', 'E', 8), ('E', 'C', 5), ('A', 'B', 7),
('B', 'E', 7)]
```

Dans cet exemple, `edges_list` pourrait représenter un graphe pondéré (cités A, B, C, D, E, F, G et distances). Le résultat imprimé est une liste des arêtes du MST trouvé et connecte bien tous les sommets avec le poids total minimal. L'algorithme fonctionne ainsi :

- On maintient l'ensemble `visited` des sommets déjà connectés à l'arbre partiel.
- À chaque itération, on parcourt toutes les arêtes et on sélectionne la **plus petite arête** `(u, v)` telle que l'un de `u` ou `v` est dans `visited` et l'autre n'y est pas encore. C'est l'arête de poids minimal pour rejoindre un nouveau sommet.
- On ajoute cette arête au résultat (`mst`) et on marque comme visité le nouveau sommet atteint.
- On répète jusqu'à ce que tous les sommets soient visités (ou qu'aucune arête ne puisse plus étendre l'arbre, ce qui indiquerait que le graphe n'était pas connexe).

Cette implémentation a une complexité O(V * E) dans le pire cas (car on scanne les arêtes à chaque ajout). On pourrait l'optimiser en utilisant une file de priorité (min-heap) pour sélectionner plus rapidement l'arête minimale à chaque étape, réduisant la complexité à O(E log V). Mais pour un nombre modéré de villes, la version simple est plus lisible. L'essentiel est qu'au terme de l'exécution, la liste `mst` contient les arêtes du réseau de routes magiques optimal à construire.

BONUS : Réponse critique à un influenceur LinkedIn

> **Postulat de l'influenceur (résumé)** : *« Les ontologies et les bases de connaissances structurées, c'est dépassé. Aujourd'hui, avec les embeddings et les modèles de langage (LLM) qui savent tout, on n'a plus besoin de ces vieilles méthodes. D'ailleurs, les bases SQL traditionnelles vont disparaître au profit du NoSQL et du big data non structuré. Inutile de perdre du temps avec la préparation des données ou la construction manuelle de graphes, l'IA fait tout toute seule maintenant ! »*

Ma réponse (critique et argumentée) :
Je comprends l'enthousiasme autour des nouvelles technologies d'IA, mais il faut tempérer ces affirmations. Les **ontologies et bases de connaissances** conservent une importance cruciale dans de nombreux domaines, et les LLMs ou embeddings ne les remplacent pas magiquement. Certes, les grands modèles de langage savent générer du texte plausible et ont mémorisé beaucoup d'informations, mais cela ne signifie pas qu'ils peuvent fournir une base de connaissances fiable, structurée et explicable. Par exemple, une ontologie bien construite permet de **vérifier la cohérence** des connaissances (grâce à des raisonnements logiques), d'assurer qu'aucune règle métier n'est violée – un LLM n'a pas cette capacité, il peut affirmer une chose et son contraire selon le prompt. Les embeddings, de leur côté, réduisent des concepts à des vecteurs : c'est formidable pour calculer des similarités ou alimenter un algorithme, mais cela ne capture pas la richesse des relations sémantiques précises. **Stocker la connaissance humaine ne se résume pas à des proximités de vecteurs** – on a besoin de savoir, par exemple, qu'« Paris est la capitale de la France » de manière formelle et non ambiguë, ce qu'une base de faits (knowledge graph) fait très bien, alors qu'un embedding de "Paris" ou un LLM pourraient l'oublier ou le mélanger avec "Paris est la capitale du Texas" (exemple d'hallucination).

De plus, les **LLMs** actuels ont des limites : ils peuvent produire des *hallucinations*, ils n'ont pas de garantis sur la vérité factuelle, et leur connaissance est figée à leur date d'entraînement. Pour une entreprise ou une application critique, baser tout le savoir sur un modèle opaque est risqué. C'est pourquoi, loin de remplacer les ontologies, on voit émerger des approches hybrides (*neuro-symboliques*) qui combinent l'usage des LLMs avec des **connaissances structurées** pour profiter des deux mondes. Par exemple, un assistant intelligent pourra consulter une base de connaissances structurée pour vérifier une information avant de répondre en langage naturel. Plutôt que d'opposer ces technologies, on les utilise de manière complémentaire.

Concernant les **bases SQL vs NoSQL**, là aussi il y a un équilibre à trouver. Les bases NoSQL ont apporté de la flexibilité et de la scalabilité, c'est indéniable et très utile pour les big data non structurées. Cependant, dire que SQL va disparaître est excessif. Les systèmes relationnels sont toujours le socle de la plupart des applications transactionnelles (banque, assurance, gestion, etc.) car ils offrent une structure, des garanties ACID et une puissance de requête inégalée pour les données bien structurées. NoSQL et SQL répondent à des besoins différents : on ne va pas stocker toutes les données d'une entreprise en JSON sans schéma si celles-ci ont une structure stable – ce serait réinventer de façon moins efficace ce que fait un SGBDR. En pratique, les architectures modernes combinent souvent des bases SQL *et* NoSQL selon les cas d'usage (par exemple SQL pour les données maîtres, NoSQL pour les logs ou profils hétérogènes, et même des graph databases pour les relations complexes). Donc, non, SQL n'est pas mort et le NoSQL n'est pas une panacée universelle : ce sont des **outils complémentaires** dans la boîte à outils du gestionnaire de données.

Enfin, l'idée qu'il serait inutile de faire de la **préparation des données** parce que "l'IA fait tout" est un mythe dangereux. En pratique, le vieil adage "garbage in, garbage out" reste on ne peut plus vrai. Aucune IA, fut-elle la plus avancée, n'arrivera à tirer des conclusions fiables à partir de données chaotiques, bruitées ou biaisées sans nettoyage. La préparation des données (nettoyage, structuration, etc.) représente souvent 80% du travail dans un projet data, et ce n'est pas par plaisir qu'on y consacre autant de temps, c'est parce que c'est indispensable. Un modèle de machine learning entraîné sur des données non préparées donnera des résultats médiocres ou trompeurs. Même les LLMs, lorsqu'on veut les spécialiser, nécessitent de bien préparer les *prompts* ou les

données d’entraînement supplémentaires (instruction tuning, etc.). Par ailleurs, la **construction manuelle de graphes ou de schémas** reste nécessaire dans des cas où l’on a besoin de contrôle et d’interprétabilité. Par exemple, pour la gestion des connaissances dans une organisation (Knowledge Management), on élabore une ontologie métier à la main, en concertation avec des experts, afin que les concepts soient précisément définis. Ce travail ne peut pas être entièrement délégué à une IA qui n’a pas le contexte ni la responsabilité du domaine.

En conclusion, je dirais à cet influenceur que s’il est enthousiasmé par les avancées en embeddings, LLMs et NoSQL (enthousiasme que je partage jusqu’à un certain point), il ne faut pas pour autant jeter aux oubliettes des décennies de bonnes pratiques en ingénierie des connaissances et en gestion de données. **Structurer la connaissance** via des ontologies ou des bases de données relationnelles apporte de la rigueur, de la fiabilité et de la transparence. Les nouveaux outils d’IA sont fantastiques pour exploiter la connaissance, trouver des liens cachés, ou automatiser certaines tâches fastidieuses, mais ils doivent être intégrés intelligemment dans des systèmes hybrides. La clé, ce n’est pas de tout opposer (symbolique vs connexionniste, SQL vs NoSQL, manuel vs automatique), c’est de combiner leurs forces. En d’autres termes, l’avenir appartient à ceux qui sauront **faire collaborer** ontologies et LLMs, bases SQL et NoSQL, préparation méticuleuse des données et algorithmes adaptatifs. C’est ainsi qu’on obtiendra le meilleur des deux mondes pour des systèmes vraiment efficaces et fiables. 【36†L1-L4】【24†L73-L81】