Chapitre 4 : Ingestion, Indexation et Requête

Comme on l'a vu, un des défis majeurs de la création de bases de connaissance est la quantité d'informations à traiter pour constituer puis pour mettre à jour la base. C'est le fameux goulot d'étranglement (« bottleneck ») de l'ingestion des données. Par exemple si l'on s'intéresse au domaine juridique en France, il faudrait collecter tous les textes de lois, décisions de justice, avis juridiques, circulaires, et autres documents administratifs produits par des centaines d'organismes. Ces données, souvent hétérogènes dans leur format (PDF, XML, bases de données relationnelles, documents papier numérisés), nécessitent des étapes préalables de préparation, comme la normalisation des formats, l'extraction des métadonnées, ou encore l'élimination des doublons. Ce processus, connu sous le nom de « Data Preparation », est crucial pour garantir la qualité des données ingérées.

Au-delà du simple aspect technique, l'enjeu est aussi temporel : les nouvelles informations doivent être intégrées rapidement pour que la base reste à jour et pertinente. Dans un contexte où les flux de données sont de plus en plus massifs et continus, l'ingestion doit être capable de s'adapter en temps réel, notamment grâce à des outils comme les connecteurs, qui permettent de relier efficacement les sources de données aux systèmes de stockage.

Ainsi, l'ingestion ne se limite pas à un problème de volume : elle touche aussi à des questions de vélocité, de diversité et de qualité des données. L'objectif final est de transformer cet océan d'informations brutes en un corpus exploitable pour les besoins spécifiques des utilisateurs, qu'il s'agisse de répondre à des requêtes simples ou de permettre des analyses complexes basées sur l'intelligence artificielle. C'est ce défi d'ingestion, et les solutions qui permettent de le surmonter, que nous explorerons dans ce chapitre.

I) Le « goulot » de l'ingestion.

Avant de pouvoir tirer pleinement parti des données, encore faut-il réussir à les intégrer efficacement dans un système. Ce processus d'ingestion, souvent sous-estimé, constitue pourtant un véritable défi technique et organisationnel. La quantité massive de données, leur hétérogénéité et leur flux continu mettent à rude épreuve les infrastructures, créant un goulot d'étranglement qui peut ralentir, voire compromettre l'ensemble du pipeline de traitement. Dans cette première partie, nous explorerons les enjeux fondamentaux de l'ingestion, les obstacles qu'elle pose, ainsi que les solutions pour les surmonter.

a) Le Big Data : définition et enjeux

Le « big data » désigne le procédé qui consiste à stocker, analyser et traiter des données dans des quantités et d'une diversité telles qu'un gestionnaire de base de données (Relational DataBase Management System, ou RDBMS pour les intimes) devient incapable de gérer. La naissance du « big data » a donc plutôt été marquée par un saut quantitatif que par un changement fondamental de techologie. Un autre facteur important est que, notamment avec l'expansion de Google et Facebook dans les années 2000, la donnée de l'utilisateur a commencé à être perçue comme un élément informatif, précieux et monnayable : avant seules certaines données spécifiques été collectées (registres de ventes…), mais maintenant tout type de donnée peut être considéré comme utile, notamment dans la conception de moteur de recommandations pour les utilisateurs. Le comportement collectif des utilisateurs est utilisé pour prendre des décisions buiseness. De même le comportement individuel de chaque utilisateur est utilisé pour adapter les contenus proposés.

Si l'expression aurait été inventée en 1997, c'est dans les années 2010 que le « big data » a été perçu comme un enjeu majeur du monde des nouvelles technologies.

L'approche à suivre pour « bien traiter » ses données dépend surtout de la nature de la donnée elle même. La classification usuellement attribuée aux données est la suivante : on distingue des données :

- structurée : tables csv, RDBMS)

- semi structurée : la donnée n'est pas clairement classée mais le support de la donnée (le fichier) présente une structure claire et qui est informative sur le contenu : fichiers de logs, pages html

- non structurée (Unstructured) : tout le reste, images, données de réseaux sociaux, texte brut...

Là où les systèmes informatiques avaient surtout manipulé la donnée structurée, elle doit maintenant se saisir de la donnée non structurée ce qui est d'autant plus complexe que : l'information n'est pas dense ni équitablement répartie dans la donnée non structurée. L'essentiel de l'information « utile » peut par exemple se situer dans la première page d'un long pdf. La donnée non structurée peut être sous des formats et types de fichier très divers, alors que la donnée structurée entrait dans quelques formats spécifiques. De plus, dans une donnée structurée, on sait immédiatement ce à quoi chaque donnée correspond et comment la relier aux reste des données. Dans des données non structurées, on ne dispose pas d'indication claire pour connecter l'information à d'autres choses.

S'il s'est beaucoup développé ces dernières années, le big data et l'analyse de données non structurées n'est pas utile à tous les cas d'usage et n'est pas souhaitable si un simple RDBMS suffit.

Le Big Data s'est petit à petit formalisé autour de définitions vagues, et ses enjeux ont été résumés par une liste d'acronymes, les 5 V pour :

Volume (supérieur à des TB de données)

Variété (incluant un mix de données structurées, non structurées…)

Vélocité (vitesse du flux de données entrant et des analyses qu'on doit générer à partir d'elles)

Véracité (la qualité des données et la fiabilité des analyses est cruciale.

Valence (les données doivent se connecter ensemble.

Le poste de Data scientist, aux contours flous, a été créé comme une sorte de magicien devant résoudre ces problèmes. La big data s'oppose à la vision traditionnelle de l'informatique décisionnelle. Alors que l'informatique décisionelle fait usage de statistiques descriptives sur des données précises pour dégager les tendances essentielles, le big data fait plutôt des statistiques inférentielles sur des données peu denses mais présentes en grand volume. Il a un rôle non pas directement applicatif mais essentiellement prédictif.En 2013, le big data faisait partie des sept ambitions stratégiques de la France déterminées par la Commission innovation 2030.

Toutefois, la question de ce traitement massif de la donnée hétérogène n'est pas du tout résolu aujourd'hui. Parmi les défis à résoudre ajourdhui, il y a l'adaptation des architectures aux flux d'information continus et leur intégration dans la base de donnée. De plus, le big data est vu comme une source d'entropie : les flux d'information continus sont comparés à des flux de nourriture et expose les systèmes qui s'en abreuvent à un risque d'« infobésité » ou de pollution. L'information réellement utile se retrouve noyée dans cette marée de l'information. Il manque encore des outils pour faire émerger le sens des données hétérogènes collectées, c'est à dire trouver la structurre dans des éléments non structurés.

b) La Data Preparation

Ainsi, une des étapes clés pour un traitement réussi de la donnée consiste à « raffiner », presque au sens pétrochimique du terme, la nappe de donnée collectée. Le terme choisi par l'industrie pour désigner cette étape fondamentale est « Data preparation » ou pré-traitement. C'était jusqu'à assez récemment une étape régulièrement assez négligée mais qui reçoit plus d'attention depuis que sa nécessité a été démontrée pour l'entraînement des LLMs : la quantité de données disponibles pour l'entraînement conduit à remettre en question la « scaling law » selon laquelle augmenter la quantité de données d'entraînement permet d'augmenter la performance du modèle. Il a été montré à quantité de données égales qu'un modèle apprenait mieux si son corpus d'entraînement avait été finement nettoyé. D'où les efforts de Hugging Face notamment de proposer des Datasets transparents et des méthodes de collecte et de nettoyage des données. Le thème fera l'objet bientôt d'une publication spécifique aux éditions Manning[1].

La préparation des données, ou *Data Preparation*, ne se résume pas à une simple étape, mais constitue en réalité un pipeline complexe de traitements successifs, ajustés en fonction de l'usage envisagé. Ce processus inclut plusieurs étapes essentielles, chacune ayant un rôle spécifique pour garantir que les données sont prêtes à être exploitées de manière optimale. Les principales étapes de ce pipeline sont l'acquisition filtrée, l'exploration, le nettoyage et la transformation.

L'acquisition filtrée est la première phase, où il s'agit de sélectionner des sources de données pertinentes. Ce filtrage est crucial, car il permet de se concentrer uniquement sur les données utiles, tout en évitant de gaspiller des ressources sur des sources qui n'apporteraient aucune valeur ajoutée.

Vient ensuite **l'exploration**, qui consiste à analyser les données collectées pour en comprendre la nature et l'organisation. Cette étape permet de découvrir les types de données, leur structure et leurs tendances sous-jacentes, afin d'orienter les traitements futurs. Elle répond notamment à des questions comme : "Les données sont-elles bien structurées ?", "Quels sont leurs formats ?", ou encore "Peut-on déjà identifier des patterns ou des anomalies ?".

1 https://www.manning.com/books/data-preparation-handbook

Le nettoyage est une phase cruciale pour garantir la qualité des données. Ici, on identifie et traite les valeurs manquantes, les *outliers* (valeurs aberrantes) et les doublons. Cette étape inclut également la vérification de la cohérence des données : il s'agit de repérer les incompatibilités entre les données nouvellement acquises et celles déjà présentes dans le système. Ce processus est indispensable pour éviter les biais ou erreurs qui pourraient compromettre l'analyse ou les résultats finaux.

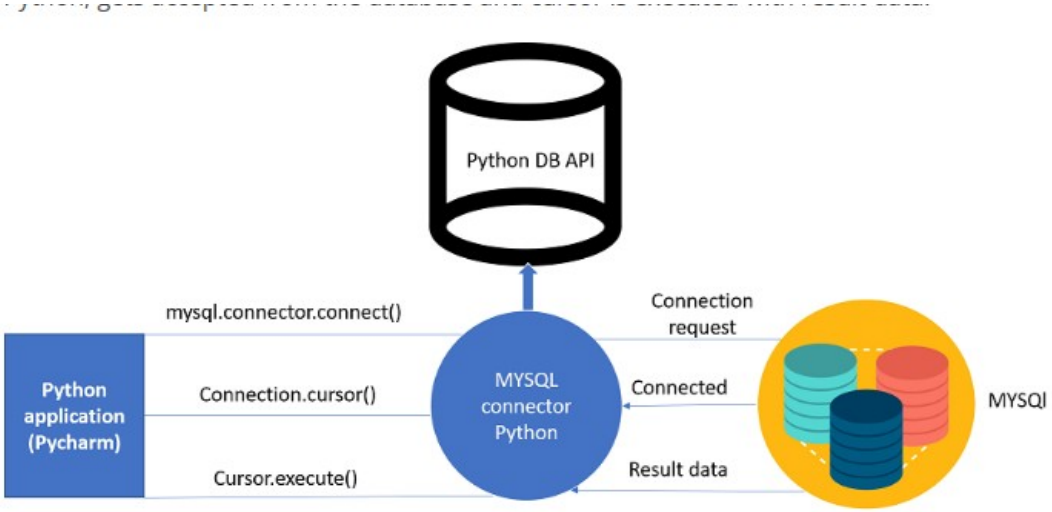
Enfin, **la transformation** consiste à convertir les données brutes dans un format plus adapté à l'analyse ou à l'ingestion dans un système cible. Cela peut inclure des processus comme l'extraction et la conversion de formats (HTML, XML, TXT, voire CSV), le *wrangling* – un terme désignant la manipulation et la restructuration des données pour les rendre analytiquement exploitables – et la définition de stratégies de combinaison entre des données issues de différentes sources. En outre, cette étape peut inclure un enrichissement des données, par exemple via l'application de solutions externes comme l'extraction d'entités nommées ou la catégorisation automatique. Ces ajouts permettent d'accroître la valeur informative des données et de les rendre directement exploitables pour des analyses avancées.

En résumé, le pipeline de *Data Preparation* est un processus dynamique et itératif qui transforme les données initiales, souvent désordonnées et hétérogènes, en un ensemble structuré et exploitable. La qualité de cette préparation conditionne directement le succès des étapes ultérieures, qu'il s'agisse de l'ingestion, de l'indexation ou de l'analyse des données.

Ce processus de nettoyage et d'enrichissement des données est aussi nécessaire à la préparation de corpus d'entraînement pour des modèles de machine learning que pour l'exploitation de ces données dans des outils de visualisation (dashboards), ou encore leur intégration à des bases de connaissances ce qui nous intéresse surtout ici.

c) Les connecteurs

Avant de pouvoir nettoyer les données hétérogènes et les transformer en base de connaissances, encore faut-il les collecter ce qui est une tâche indépendante et qui relève plus de l'ingénierie logicielle. Il s'agit même d'un design pattern de programmation (micro services notamment). Un connecteur est un composant qui communique avec une source tierce, le plus souvent des API et qui permet l'intégration des flux venant de cette source tierce dans une application. La ressource tierce peut être sur la même machine / infrastructure (base de donnée, RDBMS) ou appartenir à une institution différente.



Par exemple on peut écrire un connecteur en python à une base de donée SQL qui permettra de vérifier que la base est bien active et de requêter des données. On peut ensuite élaborer le connecteur de façon à ce que le code python soit notifié à chaque fois que la base de données SQL est modifiée.[2]

Le rôle d'un connecteur est notamment de gérer correctement la notion d'événement, c'est-à-dire de détecter et de réagir aux changements ou mises à jour survenus dans la source tierce. Par exemple, dans le cas d'une base de données SQL, un connecteur bien conçu pourrait non seulement récupérer des données sur demande, mais aussi être configuré pour surveiller les événements tels que l'ajout, la modification ou la suppression d'enregistrements. Ce type de comportement, souvent appelé *event-driven architecture*, permet de s'assurer que l'application qui consomme les données reste toujours synchronisée avec la source.

Un autre aspect crucial des connecteurs est leur capacité à gérer la diversité des sources. Les flux de données provenant de sources hétérogènes, comme des APIs REST, des services cloud tiers, des bases de données relationnelles, ou encore des fichiers stockés localement, nécessitent des connecteurs spécifiques pour traduire ces données dans un format unifié et utilisable. Par exemple, un connecteur pour une API REST doit gérer les authentifications, les limitations de requêtes (*rate-limiting*), et les formats de réponse JSON ou XML, tandis qu'un connecteur pour des données brutes en local doit pouvoir lire et interpréter divers formats de fichiers comme CSV ou Excel.

De plus, un connecteur doit être conçu pour assurer une certaine robustesse face aux erreurs. Les interruptions de connexion, les réponses invalides, ou encore les délais de réponse trop longs sont autant de problèmes que le connecteur doit anticiper et gérer sans compromettre l'ensemble du pipeline d'ingestion. C'est ici que des mécanismes comme les tentatives automatiques (*retries*), les files d'attente des événements (*queues*), et les journaux des erreurs (*error logs*) entrent en jeu, garantissant une fiabilité accrue du système.

Il existe des frameworks d'ingestion de données comme Kafka[3] qui permettent de gérer la collection depuis des sources diverses. Apache Kafka est une plateforme open-source de streaming d'événements distribuée, développée par la Fondation Apache, conçue pour ingérer, stocker et traiter des flux de données en temps réel. Elle est largement utilisée pour construire des pipelines de données en temps réel et des applications qui s'adaptent aux flux de données. Kafka fonctionne sur un cluster composé de serveurs appelés brokers, qui stockent et gèrent les données. Les données

2 https://www.edureka.co/blog/python-database-connection/ source de l'illustration
3 https://www.confluent.io/fr-fr/what-is-apache-kafka/

dans Kafka sont organisées en topics, qui sont subdivisés en partitions. Chaque partition est une séquence ordonnée de messages, permettant une parallélisation et une distribution efficaces des données. Kafka est conçu pour être hautement performant, capable de gérer des millions de messages par seconde, tout en assurant une faible latence et une tolérance aux pannes. Il est utilisé dans divers cas d'utilisation, tels que la collecte de journaux, le suivi des activités des utilisateurs, la surveillance en temps réel et l'intégration de données entre systèmes.

Les connecteurs jouent un rôle fondamental dans le processus d'ingestion, car ils forment le lien entre les sources de données et l'application cible. En transformant des flux disparates en un flux unifié, tout en assurant la synchronisation, la gestion des événements et la résilience, ils sont une pièce essentielle de l'écosystème de gestion des données.

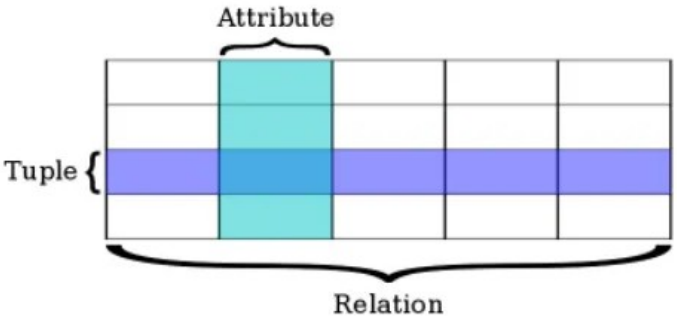
II) Les différents types de bases de données et leur fonctionnement

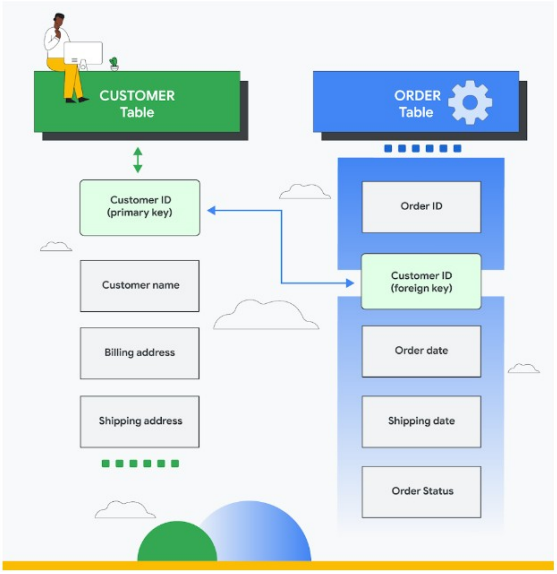
Une fois la donnée collectée et nettoyée, il convient de la stocker pour en faire usage plus tard. C'est à dire que la donnée doit être stockée de manière à pouvoir être récupérée facilement en cas de nécessité. Une base de données est un système organisé qui permet de stocker, gérer et accéder facilement à des informations. Ces données sont structurées pour faciliter leur utilisation, que ce soit pour les consulter, les modifier ou les analyser. Les bases de données peuvent être numériques (comme des fichiers ou des systèmes informatiques) et sont souvent gérées par des logiciels spécialisés appelés systèmes de gestion de bases de données (SGBD). Elles sont utilisées dans presque tous les domaines, des sites web aux applications bancaires, en passant par les réseaux sociaux. Le stockage de la donnée doit également répondre à certains principes de confidentialité, de sécurité, de fiabilité (la donnée ne doit pas être perdue en cas de défaillance technique) de latence (la recherche doit être rapide) et de scalabilité (la base doit bien fonctionner avec une très grande quantité de donnée, si possible l'utilisation des fonctionnalités de recherche et d'aggrégation doit être au pire linéaire en fonction du nombre de documents indexés. Il convient donc de bien choisir le type de base dans laquelle on stocke ses données.

a) Les Bases de données relationnelles : un paradis structuré

L'idée derrière les bases relationnelles germe dans les années 1970, avec le travail révolutionnaire d'Edgar F. Codd, un chercheur chez IBM. Codd publie un article intitulé *"A Relational Model of Data for Large Shared Data Banks"* dans lequel il propose un modèle de données fondé sur les mathématiques des relations, un concept issu de la théorie des ensembles. Ce modèle offre une approche rigoureuse et logique pour structurer et interroger des données, contrastant avec les modèles hiérarchiques, dominants à l'époque.En 1974, IBM lance le projet System R pour mettre en œuvre les idées de Codd. Ce projet aboutit à la création de SQL (Structured Query Language), un langage standardisé pour interroger et manipuler les bases relationnelles. SQL devient rapidement le langage de référence, encore largement utilisé aujourd'hui.La première base de données relationnelle commerciale, Oracle, voit le jour en 1979.

Le concept central des bases de données relationnelles repose sur des tables[4] (appelées relations) composées de lignes (tuples) et de colonnes (attributs). Cette abstraction rend le stockage des données plus flexible, tout en garantissant leur cohérence grâce à des concepts tels que les clés primaires, les clés étrangères et l'intégrité référentielle. Chaque ligne représente une observation. Les colonnes (autres que la clé primaire) sont des attributs de la ligne et on peut considérer que chaque ligne de la table est comme un tuple. L'ordre des éléments importe (type des attributs) et il ne fait sens que dans son ensemble. On appelle « schéma » de la base de donnée la liste des attributs de la table.

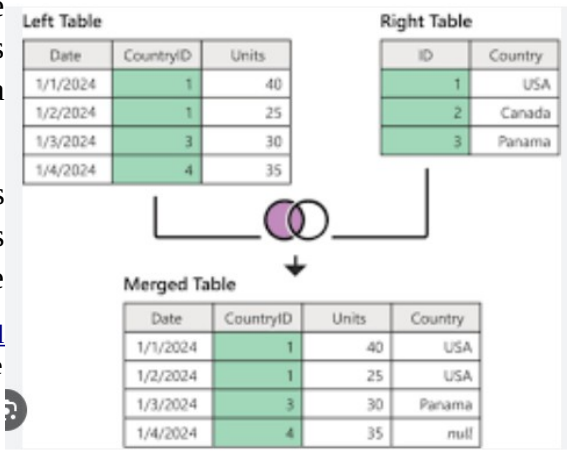




Une table dans une base de donnée relationnelle est l'équivalent d'une table dans un tableur / fichier csv[5]

Chaque table dispose d'une « clé primaire » qui est unique pour chaque ligne de la table. Cette clé permet d'identifier la ligne / l'objet dans la base.

Les autres colonnes sont des attributs et apportent des informations supplémentaires sur l'observation. La présence de ces attributs permet de faire des recherches filtrées en ne renvoyant que les lignes de la table présentant certaines valeurs dans les attributs. En complément de la clé primaire, qui identifie de manière unique chaque ligne d'une table, les bases de données relationnelles utilisent également le concept de **clé étrangère** (*foreign key*). Une clé étrangère est une colonne (ou un ensemble de colonnes) dans une table qui établit un lien avec la clé primaire d'une autre table. Ce lien permet de relier des informations stockées dans des tables différentes, garantissant la cohérence et la relation logique entre les données. Par exemple, dans une base de données contenant une table Customer et une table Order, la table Order pourrait inclure une colonne Customer_ID qui sert de clé étrangère, pointant vers la clé primaire ID_Customer de la table Customer. Cela indique qu'une commande donnée est associée à un client spécifique. Ce mécanisme permet de structurer les données de manière modulaire et d'éviter les duplications inutiles de tous les attributs du client dans la table des commandes.



Pour exploiter les relations établies par les clés étrangères, les bases de données relationnelles utilisent les **jointures**. Une jointure est une opération qui permet de

4 Source de l'illustration : https://medium.com/@authfy/introduction-rel

5 Plus précisément les tableurs et leur syntaxe ont été inspirés des bases de

combiner des données provenant de plusieurs tables en fonction d'un critère commun, souvent basé sur une clé étrangère. La clé étrangère et les jointures sont des éléments fondamentaux des bases de données relationnelles, car elles permettent d'éviter les redondances. Par exemple, les informations des clients sont stockées dans une seule table et reliées aux commandes par une clé étrangère. Grâce à des contraintes d'intégrité référentielle, une base relationnelle peut garantir que chaque clé étrangère correspond toujours à une clé primaire valide.Les jointures permettent de regrouper et d'exploiter efficacement les données réparties sur plusieurs tables. Ces mécanismes illustrent la puissance et la flexibilité des bases de données relationnelles pour gérer des données complexes de manière organisée et fiable.

Avantages	Inconvénients
1. Structure claire et organisée : Les données sont stockées dans des tables bien définies, ce qui facilite leur compréhension et leur manipulation.	**1. Moins adaptées aux données non structurées** : Les bases relationnelles ne sont pas optimales pour gérer des données comme des images, des vidéos ou des documents.
2. Intégrité et cohérence des données : Grâce aux clés primaires, étrangères et aux contraintes d'intégrité, les bases relationnelles garantissent des données fiables.	**2. Complexité des schémas** : La normalisation peut conduire à des schémas très fragmentés, nécessitant des jointures coûteuses.
3. Standardisation : SQL est un langage universellement utilisé pour interagir avec les bases relationnelles, ce qui facilite leur adoption et leur portabilité.	**3. Performances limitées pour les grands volumes** : Les bases relationnelles peuvent devenir lentes lorsque les données atteignent des volumes massifs ou nécessitent des traitements intensifs.
4. Sécurité et contrôle des accès : Les bases relationnelles offrent des mécanismes robustes pour gérer les autorisations et sécuriser les données sensibles.	**4. Scalabilité verticale** : Les bases relationnelles sont plus adaptées à une scalabilité verticale (ajout de ressources à un seul serveur), moins efficace qu'une scalabilité horizontale.
5. Support des transactions : Les bases relationnelles gèrent les transactions avec des propriétés ACID (Atomicité, Cohérence, Isolation, Durabilité), essentielles pour les applications critiques.	**5. Coûts élevés** : Les systèmes de gestion de bases relationnelles (SGBD) commerciaux peuvent être coûteux en termes de licence et d'infrastructure.
6. Large écosystème et communauté : Disponibilité d'outils, de documentation et de support technique grâce à leur adoption massive.	**6. Moins flexibles pour les évolutions rapides** : Modifier un schéma relationnel peut être complexe et nécessiter une planification minutieuse.
7. Facilité de requêtage complexe : Les jointures et les fonctions SQL permettent d'exécuter des analyses sophistiquées directement dans la base.	**7. Surconsommation de ressources** : Les jointures et requêtes complexes peuvent consommer beaucoup de mémoire et de temps d'exécution.

De par leur clarté et leur requêtage intuitif, les bases de données relationnelles ont fait leur nid dans la plupart des applications technologiques que nous utilisons quotidiennement. Néanmoins, elles souffrent de plusieurs limitations. Si c'est leur structure stricte qui permet leur usage facile, elles ne conviennent pas du tout à des données qui ne sont pas structurées (où cela impliquerait un travail de pré-traitement lourd pour faire rentrer « au forceps » les données hétérogènes dans les schémas définis.

De plus, ces bases de données sont peu souple, et certaines opérations de jointure peuvent être très coûteuses en temps de calcul sur des tables immenses (vous vous souvenez quand pandas mettait du temps à traiter toutes vos données?)

Devenues omniprésentes depuis les années 90 grâce à leur capacité à gérer efficacement de grandes quantités de données et à leur compatibilité avec des systèmes informatiques variés, les bases de données relationnelles dominent encore aujourd'hui de nombreux secteurs, bien que des alternatives comme les bases NoSQL aient émergé pour répondre à des besoins spécifiques, comme le traitement de données non structurées ou massivement distribuées.

b) Les bases de données No SQL ou la recherche tout en souplesse

Outre un coût computationnel important, un défaut structurel des bases SQL est qu'elles ne permettent pas de mise à jour facile du schéma des tables. Or une application qui grandit a parfois besoin de stocker plus de données. Par exemple un site peut modifier et agrandir son formulaire d'informations personnelles. Mais les gestionnaires de bases de données relationnelles ne

permettent pas facilement d'ajouter des attributs et ce simple ajout nécessiterait de recréer des nouvelles tables et de copier depuis les anciennes… Que de complications. C'est notamment dû au fait que dans SQL, chaque ligne de la table est censée avoir le même nombre d'attribut. Une des premières démarcations du NoSQL a donc été de faire tomber cet impératif.

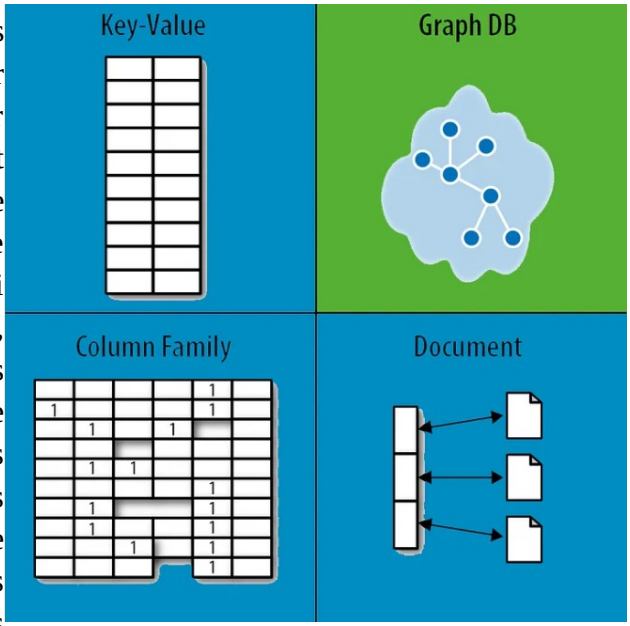
NoSQL ne signifie pas simplement « No SQL », mais Not only SQL et se veut comme une extension de ce que l'on peut faire avec des bases de données relationnelles. Le NoSQL regroupe donc une grande diversité d'implémentation de bases de données qui présentent toutefois quelques points communs :

* L'absence de schéma contraignant

* La consistance éventuelle : l'état du système doit respecter les protocoles non pour chaque opération mais une fois que toutes les opérations reçues en même temps ont fini d'être traitées (condition relâchée par rapport au RDB)

* Les données sont dupliquées sur plusieurs stockages (shards) pour que si l'un est détruit, la donnée soit quand même sauvegardée.

* pensé pour la scalabilité et pour accueillir des données hétérogènes.



Il existe plusieurs types de bases de données NoSQL[6]. La plus basique consiste en un stockage clé valeur (principe de la table de hachage), ce qui permet d'associer un ID à une représentation d'un objet. Ces bases sont extrêmement performantes mais ont une finesse de recherche limitée. Une variante récente de la base de donnée clé valeur est la base de donnée vectorielle qui consiste à construire un embedding d'un objet (texte, image) et de faire de cet embedding la clé dans la base.elles ne se contentent pas de stocker des embeddings en tant que clés. Elles utilisent ces embeddings pour permettre des recherches par similarité à l'aide de métriques spécifiques (cosinus, distance euclidienne, etc.). Ces bases, comme Pinecone, sont particulièrement adaptées aux applications d'intelligence artificielle et de machine learning, comme les moteurs de recherche sémantiques.

Les bases de données par colonne quant à elles sont des extensions plus souples des bases de données relationnelles : les tables n'ont pas de schéma fixe mais des colonnes, comme par exemple Kassandra, qui peuvent être laissées vides. Ces bases permettent aussi d'ajouter de nouvelles colonnes à la volée. D'autres bases de données, comme MongoDB ont à leur coeur la notion de document et ont été pensées pour le traitement de la donnée hétérogène. Ce sont souvent des documents semi structurés (json , html) qui doivent être stockés pour permettre les requêtes filtrées sur les champs / balises.

6 https://medium.com/free-code-camp/nosql-databases-5f6639ed9574

Les bases de données à base de graphe, sur lesquelles on reviendra, sont aussi considérées comme s’inscrivant dans la lignée du NoSQL.

Il convient également de faire une distinction entre une base de donnée et un moteur de recherche (Lucene, Elastic). La distinction entre base de données (à requêter) et moteur de recherche réside principalement dans la manière dont les données sont stockées, indexées et récupérées. Une base de données (notamment relationnelle ou NoSQL) est conçue pour stocker et gérer des données structurées de manière persistante. Elle permet des opérations CRUD (Création, Lecture, Mise à jour, Suppression) sur des données structurées ou semi-structurées. L'objectif principal est de gérer des transactions, garantir la cohérence des données (en particulier pour les bases relationnelles) et permettre des requêtes complexes sur des ensembles de données. Les bases de données sont donc idéales pour des systèmes où l'intégrité des données et la possibilité de les manipuler de façon flexible sont primordiales.Un moteur de recherche est un système conçu pour indexer et rechercher rapidement des informations dans de grandes quantités de données (souvent non structurées ou semi-structurées). Il est optimisé pour les recherches textuelles et pour la récupération de documents pertinents en fonction de requêtes de recherche. L’objectif principal est de fournir des résultats pertinents basés sur une recherche par mots-clés, des expressions ou des critères de similarité. Techniquement, un moteur de recherche se distingue par la création d’un index inversé permettant des recherches rapides à partir de requêtes en langage naturel. Ces deux technologies sont complémentaires dans un système où il est nécessaire de stocker des données (base de données) et d'effectuer des recherches efficaces sur ces données (moteur de recherche).

c) Les bases de données à base de graphe ou la recherche du lien.

Une des reproches majeurs faite aux bases de données SQL et en colonne est qu’elles ne fournissent pas de lien entre les observations qu’elles stockent. Au mieux elles proposent des renvois qui sont des sortes de lien symboliques visant à optimiser le stockage et l’inférence. Bien que cette approche puisse être fonctionnelle, elle reste limitée dans sa capacité à modéliser des connexions profondes et dynamiques entre les entités. Cela devient particulièrement problématique dans des contextes où les données sont intrinsèquement connectées et où la recherche de liens complexes, tels que les relations entre personnes, objets, événements, ou autres entités, est essentielle. Par exemple, un client peut avoir passé 3 commandes, envoyé une réclamation au service client au sujet d’une de ces commandes car une des commandes a été envoyée en double ; il a commandé deux fois la même paire de chaussure ! Les interactions entre les objets commandés, les commandes et le ticket de réclamation n’est pas visible sur les tables aplaties où seules l’ID du client permet de relier les lignes.

C'est ici qu'interviennent les bases de données à base de graphe, qui sont spécialement conçues pour représenter et naviguer dans des réseaux de données fortement interconnectées. Contrairement aux bases de données relationnelles, qui organisent les données en tables de lignes et de colonnes, les bases de données à base de graphe utilisent des nœuds, des arêtes et des propriétés pour modéliser les données.

- Les nœuds représentent les entités, telles que des personnes, des lieux, des événements, ou tout autre objet que l'on souhaite stocker.
- Les arêtes sont les liens ou relations entre ces nœuds, comme par exemple "ami de", "situé à", "auteur de".

- Les propriétés sont des attributs des nœuds ou des arêtes, comme "nom", "âge", ou "date de création".

Les nœuds représentent donc finalement une ligne d'une table relationnelle, avec ses attributs, et les relations précisent les liens qu'ont plusieurs objets entre eux permettant de repérer des patterns.

L'un des grands avantages de ce modèle est la capacité à naviguer facilement dans des relations complexes. Les requêtes qui impliquent des relations multiples (par exemple, "trouver tous les produits que commandent ceux qui ont commandé ces bottines" pour un système de recommandation) sont particulièrement efficaces dans une base de données à base de graphe, car les arêtes permettent de parcourir directement ces liens. Le modèle de graphe est particulièrement adapté pour des applications telles que les réseaux sociaux, les recommandations la détection de fraude, ou les systèmes de gestion de la connaissance.

D'un point de vue technique, ces bases sont optimisées pour chercher les entités connectées à une entité d'intérêt et explorer, sans avoir besoin de faire de jointure entre les tables entières comme en SQL. On a donc des performances bien supérieures aux bases de donnée relationnelles pour répondre à des questions sur « les amis des amis des amis... »

A finir avec limitations + expliquer noms : noeds , relations : verbes.

Illustrer

+ discuter ambivallence entre attributs, nœuds indépendant, importance de bien concevoir son schéma…

III) retrouver sa donnée dans l'océan : le principe de requêtes

Une base de données n'est pas simplement un espace de stockage : elle doit offrir des outils efficaces pour retrouver rapidement les informations pertinentes parmi l'ensemble des données. L'objectif est de pouvoir localiser précisément la ou les données recherchées, même dans des bases massives, et ce, en un temps réduit. Cela est rendu possible grâce à des mécanismes de requêtes, qui permettent d'interroger la base de données de manière structurée.

Le principe de requête repose sur une syntaxe ou un langage conçu pour exprimer les besoins de l'utilisateur. Parmi les langages de requêtes les plus courants, on trouve **SQL (Structured Query Language)**, utilisé dans les bases relationnelles, ou des langages spécifiques comme **SPARQL**, adapté aux bases de données orientées graphes et aux données RDF dans le cadre du web sémantique. Ces requêtes sont ensuite traduites par le gestionnaire de BDD en une série d'instructions à exécuter en vue de renvoyer à l'utilisateur des résultats pertinents. Derrière cette interface languagière se cachent des algorithmes d'optimisation puissants dont l'objectif est de diminuer le temps de latence.

On considère que l'opération de requête sur une base de données figée est déterministe, c'est à dire que la même requête exécutée plusieurs fois donne le même résultat (ce qui n'est pas toujours le cas

pour les moteurs de recherche, notamment ceux augmentés par IA). On étudiera la structure des langages de requête avant d'essayer de comprendre la réalité algorithmique qu'ils cachent pour retourner rapidement des résultats.

a) Le langage de la recherche : Comparaison des langages de Query et de leur expressivité

* L'origine des langages de requête

Les langages de requête trouvent leurs origines dans les premiers systèmes de gestion de bases de données développés dans les années 1960 et 1970, lorsque l'explosion des données a rendu nécessaire l'élaboration de méthodes pour interroger efficacement des ensembles d'informations stockées. Initialement, ces systèmes reposaient sur des modèles hiérarchiques et en réseau, avant que le modèle relationnel proposé par Edgar F. Codd dans les années 1970 ne s'impose comme une révolution. Le modèle relationnel introduisait une abstraction élégante pour organiser les données sous forme de tables interconnectées, et avec lui, est né **SQL (Structured Query Language)**, le premier langage largement adopté pour interagir avec des bases relationnelles. Le premier langage de requête n'est donc au final que la transposition langagière / en un langage de programmation de l'algèbre relationnel élaboré par Codd.

SQL a été conçu comme un langage déclaratif : l'utilisateur spécifie ce qu'il veut obtenir, et non comment l'obtenir, laissant au système de gestion de bases de données (SGBD) le soin d'optimiser et d'exécuter les opérations nécessaires. Ce paradigme contrastait avec les approches procédurales des premiers langages informatiques, en rendant l'interrogation des données à la fois plus intuitive et plus puissante. Avec l'émergence du web sémantique, des langages comme **SPARQL** ont ensuite été conçus pour gérer des données plus complexes, exprimées sous forme de graphes, élargissant encore l'horizon de l'expressivité.

*Un langage structuré : La syntaxe et les mots-clés

Les langages de requête, à l'instar des langages de programmation possèdent une grammaire, des règles syntaxiques, et des mots-clés qui structurent les requêtes. Prenons l'exemple de SQL, qui repose sur un vocabulaire réduit mais très expressif. Une requête SQL typique pourrait s'écrire :

```
SELECT titre, auteur FROM livres WHERE année > 2000 ORDER BY année DESC;
```

Dans cet exemple, chaque mot-clé a une fonction précise :

- SELECT : Indique les colonnes ou attributs que l'utilisateur souhaite obtenir.
- FROM : Désigne la table ou les tables où chercher l'information.
- WHERE : Spécifie des conditions pour filtrer les résultats.
- ORDER BY : Définit l'ordre de tri des données et le sens (DESC)

Cette structure est lisible pour un humain tout en étant traduisible en instructions compréhensibles pour une machine, grâce à des algorithmes d'optimisation intégrés dans le SGBD. Ainsi, le langage est agnostique à la base de donnée. Chaque RDBMS implémente son interpréteur du langage (le

rendant parfois plus expressif que le SQL de base) et le traduit en instructions au moment de l'exécution de la requête.

De manière similaire, SPARQL adopte une syntaxe déclarative mais adaptée aux graphes. Une requête pour retrouver les livres publiés après 2000 pourrait s'écrire ainsi :

```
SELECT ?titre ?auteur
WHERE {
  ?livre rdf:type :Livre .
  ?livre :titre ?titre .
  ?livre :auteur ?auteur .
  ?livre :année ?année .
  FILTER (?année > 2000) }
ORDER BY DESC(?année)
```

SPARQL exploite une syntaxe qui reflète la structure des graphes RDF, en associant chaque élément (comme ?livre) à des relations (rdf:type, :titre) pour naviguer dans des relations complexes entre objets. Le langage Cypher, créé par Neo4J se veut comme une extension des fonctionnalités de Sparql.

*** L'expressivité des langages de requête**

L'expressivité d'un langage de requête se mesure par sa capacité à formuler des demandes complexes de manière concise et intuitive. Cette expressivité repose sur plusieurs aspects :

1. **Manipulation des données** : Les langages comme SQL permettent des opérations riches (agrégats, jointures entre tables, sous-requêtes), rendant possible l'analyse approfondie des données.
2. **Navigation dans des relations** : SPARQL, grâce à son orientation graphes, est idéal pour explorer des réseaux denses, comme les bases de connaissances (e.g., Wikidata, Knowledge Graphs).
3. **Puissance des conditions** : Les filtres (WHERE, FILTER) permettent d'exprimer des critères précis et d'exclure les résultats indésirables.
4. **Opérations avancées** : Certains langages offrent des fonctionnalités avancées comme les expressions régulières (REGEX dans SPARQL) ou la gestion des données temporelles.

L'expressivité d'un langage de requête dépend donc des mots clés qu'il contient et qui correspondent à des opérations plus ou moins complexes implémentées dans le DBMS.

Voici une liste non exhaustive qui compare les mots clés des langages SQL, Sparql et Cypher. On remarque que ces langages partagent la plupart de leurs mots clés. Même si parfois ce ne sont pas les mêmes mots, ils ont une fonction équivalente. Ensuite SQL se sépare de Sparql et Cypher car il dispose de mots clés particuliers pour gérer les tables (jointures) là où les deux autres ont des mots particuliers pour gérer des relations. Certains mots-clés n'ont pas d'équivalents directs mais peuvent être simulés avec d'autres constructions (par exemple, HAVING en SQL peut être partiellement remplacé par WITH ... WHERE en Cypher). L'expressivité du langage de requête n'est donc au final que la réflexion des possibilités de la base.

On peut classer les mots clés en plusieurs catégories ;

* Les opérateurs de sélection (SELECT, RETURN) qui permettent de préciser les champs que l'on doit retirer de la base

* Les opérateurs d'extraction (FROM) qui indiquent la base / table où chercher

Fonction	SQL	SPARQL	Cypher
Sélection de données	`SELECT`	`SELECT`	`RETURN`
Source des données	`FROM`	`FROM` (SPARQL 1.1 pour graphes)	`MATCH`
Filtrage des données	`WHERE`	`FILTER`	`WHERE`
Tri des résultats	`ORDER BY`	`ORDER BY`	`ORDER BY`
Agrégation	`GROUP BY`	`GROUP BY`	`WITH`
Filtrage des groupes	`HAVING`		`WITH ... WHERE`
Jointures	`JOIN`		`MATCH` avec relations
Alias	`AS`	`AS`	`AS`
Conditions alternatives	`OR`		`OR`
Conditions multiples	`AND`		`AND`
Négation	`NOT`	`!` (ou `NOT`)	`NOT`
Limitation des résultats	`LIMIT`	`LIMIT`	`LIMIT`
Décalage des résultats	`OFFSET`	`OFFSET`	`SKIP`
Valeurs distinctes	`DISTINCT`	`DISTINCT`	`DISTINCT`
Comptage	`COUNT`	`COUNT`	`COUNT`
Somme	`SUM`		`SUM`
Moyenne	`AVG`		`AVG`
Valeur minimale	`MIN`		`MIN`
Valeur maximale	`MAX`		`MAX`
Conditions sur des ensembles	`IN`	`VALUES`	`IN`
Union de résultats	`UNION`	`UNION`	`UNION`
Sous-requêtes	Sub-`SELECT`	Sub-`SELECT`	Sub-`MATCH`
Expressions régulières		`REGEX`	`=~`
Création de données	`INSERT INTO`	`INSERT DATA`	`CREATE`
Mise à jour de données	`UPDATE`	`DELETE` / `INSERT`	`SET`
Suppression de données	`DELETE`	`DELETE DATA`	`DELETE`
Conditions de vide	`IS NULL`	`BOUND`	`IS NULL`
Existence d'une relation		`EXISTS`	`EXISTS`
Noms des relations			`RELATIONSHIP`
Création de schéma	`CREATE TABLE`		`CREATE CONSTRAINT`
Suppression de schéma	`DROP TABLE`		`DROP CONSTRAINT`

* Les opérateurs de Filtrage (WHERE, IF…) qui restreignent les résultats

* Les opérateurs d'agrégation qui permettent de faire des différentes valeurs d'un attribut le pivot de la recheche.

* Des opérateurs effectuant une transformations sur un champs.

* Des opérateurs gérant d'autres aspects du CRUD (sans le R, du coup) : création, update et Déletion d'un élément ;

* Des opérateurs logiques.

Les langages de requête ne sont pas figés mais évoluent. Chaque nouvelle mise à jour du KDBMS apporte son lot de nouvelles fonctionnalités et les nouvelles syntaxe pour les accueillir. Quand une nouvelle base de donnée est créée, elle peut partir d'un langage de requête existant et l'enrichir selon ses nouvelles spécificités. C'est ce qui a souvent été fait au début des bases NoSQL pour les bases dites à colonne. L'exemple le plus marquant est le langage Cypher, qui au départ était une création de l'entreprise NEo4J qui avait la position de leader sur les bases de données de type graphe, mais ce langage a été Forké et une version open source se développe en parallèle de celle fixée par l'entreprise[7]. Cette version parallèle de Cypher a été adoptée par de

7 https://s3.amazonaws.com/artifacts.opencypher.org/openCypher9.pdf

nombreuses bases de données de type graphe, avec l'espoir que des utilisateurs fervents de Neo4J puissent facilement migrer vers leur solution. Le langage de requête est donc un élément fondamental d'un système de donnée car il conditionne les potentielles requêtes que l'on peut faire sur la base. Néanmoins toutes les bases de données n'implémentent pas de langage de requête. Les bases noSQL de type clé valeur et les bases de données vectorielles n'en ont par exemple pas. C'est bien la preuve que ces langages ne sont que des surcouches logicielles et cachent des réalités algorithmiques bien plus complexes.

b) Les algorithmes de la recherche

* Pour les bases SQL

Les langages de requête comme SQL offrent à l'utilisateur une abstraction élégante, mais derrière cette simplicité apparente se cachent des algorithmes complexes qui rendent possible l'interrogation rapide et efficace des bases de données. Lorsqu'une requête est soumise, le système de gestion de bases de données (SGBD) passe par plusieurs étapes pour la traiter. Tout commence par une analyse syntaxique et sémantique. La requête est vérifiée pour s'assurer qu'elle respecte les règles du langage SQL et que les tables, colonnes ou relations mentionnées existent réellement dans la base. Une fois cette vérification faite, le SGBD procède à une optimisation : il génère plusieurs plans d'exécution possibles et évalue leur coût en termes de temps et de ressources. L'objectif est de choisir le plan le plus efficace avant de passer à l'exécution.

Chaque clause d'une requête SQL est ensuite traduite en une série d'opérations algorithmiques spécifiques. Par exemple, lorsqu'une requête inclut une condition sur une colonne indexée, le SGBD utilise des structures d'index comme les arbres B ou B+ pour localiser directement les lignes concernées, évitant ainsi de parcourir toute la table. Si aucun index n'est disponible, le système effectue un parcours séquentiel, examinant ligne par ligne pour trouver les correspondances, ce qui est beaucoup moins performant. Lorsqu'une requête implique des jointures entre tables, comme dans le cas d'un INNER JOIN, le SGBD peut choisir parmi plusieurs algorithmes en fonction de la taille des tables et des données. Une jointure par boucle imbriquée, par exemple, compare chaque ligne d'une table à chaque ligne de l'autre, une méthode simple mais lente. Un autre algorithme, appelé jointure par hachage, crée une table de hachage pour l'une des tables, ce qui accélère les comparaisons. Enfin, la jointure par tri-fusion trie les deux tables sur les colonnes pertinentes avant de les parcourir simultanément pour identifier les correspondances, ce qui est particulièrement efficace pour les très grands ensembles de données.

D'autres parties de la requête, comme les agrégations ou les regroupements, reposent également sur des algorithmes sophistiqués. Les fonctions comme GROUP BY ou SUM nécessitent souvent un tri préalable ou l'utilisation de tables de hachage pour regrouper les données avant de calculer les résultats. De même, les clauses comme ORDER BY, qui imposent un tri des résultats, font appel à des algorithmes tels que QuickSort ou MergeSort, adaptés pour minimiser l'utilisation de mémoire et optimiser les performances.

* Pour les bases de données vectorielles

Comme on l'a vu, la requête des bases SQL se fait surtout par des algorithmes qui parcourent les tables, avec des astuces à bases d'arbre et de tables de hachage. Sur une recherche simple ou une agrégation, la complexité est O(n) avec n le nombre de lignes dans la table. C'est

seulement quand on fait des jointures que les choses deviennent horribles car on multiplie les complexités. En revanche, pour les bases de données vectorielles, la situation est a priori la même pour chaque requête puisque on requête par similarité à partir du vecteur représentant la requête.

Il faudrait donc a priori calculer la mesure de similarité (cosinus, norme L2…) entre ce vecteur et tous les vecteurs de la base, car la haute dimensionnalité des vecteurs fait que le résultat de l'opération de similarité n'est pas visible. Or, vous le savez, le calcule de vecteur peut être long et coûteux, surtout sans GPU. Des algorithmes ont donc été développés pour optimiser la recherche vectorielle.

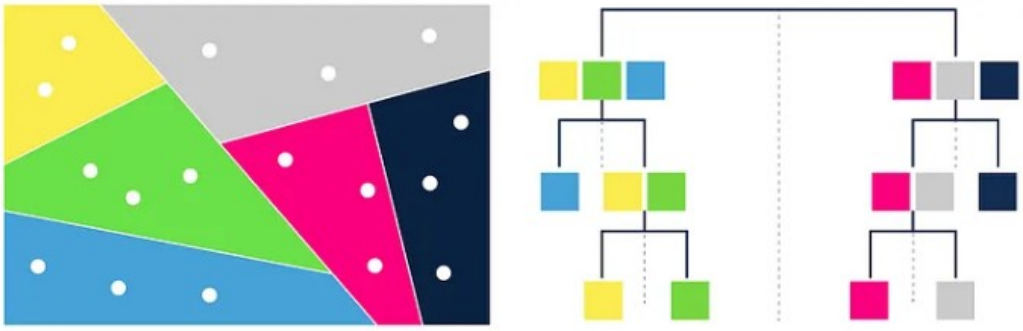
Puisque la recherche vectorielle exacte est très coûteuse, une alternative fiable dite **Approximate nearest neighbour** a été développée.

L'Approximate Nearest Neighbour **(ANN)** est une méthode utilisée pour trouver les points les plus proches d'une requête dans un espace de données de grande dimension, tout en sacrifiant légèrement la précision pour gagner en rapidité. Contrairement aux algorithmes classiques de recherche des voisins les plus proches, qui garantissent un résultat exact mais sont souvent coûteux en calcul pour de très grands ensembles de données, ANN privilégie l'efficacité.

L'idée est de simplifier la recherche en divisant l'espace en structures comme des arbres, des graphes ou des clusters, permettant de réduire considérablement le nombre de comparaisons nécessaires. Les algorithmes ANN, comme **HNSW (Hierarchical Navigable Small World)** ou **LSH (Locality-Sensitive Hashing)**, exploitent ces structures pour repérer rapidement des candidats potentiels, avant de les affiner pour produire un résultat proche du voisinage exact.

ANN désigne une famille d'algorithmes qui ont parfois des implémentations très différentes.

* les algorithmes à base d'arbres[8]

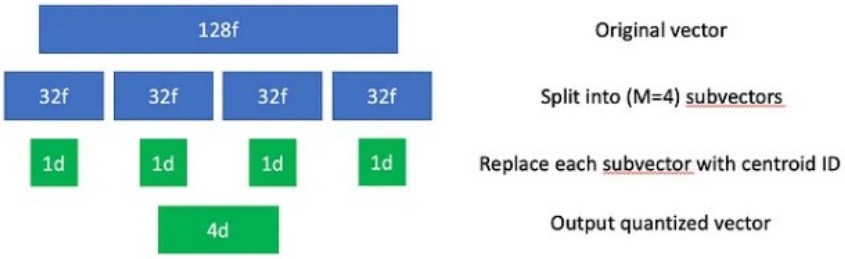


On espère réduire le nombre de calcul de n à log(n) en groupant ensemble les vecteurs similaires.

* Des méthodes de hashage regroupant des vecteurs similaires en paquets et se dirigeant à chaque étape vers le paquet dont le représentant est le plus proche de la requête. Mais risque de mauvais résultat pour les cas limites si on est orienté vers le paquet voisin.

* les méthodes à base de quantization[9] ou de compression : on diminue la taille prise pour encoder un vecteur, voire la dimensionnalité du vecteur, au risque de perdre de l'information.

8 **https://medium.com/@david.gutsch0/vector-databases-understanding-the-algorithm-part-3-bc7a8926f27c**
9 Source de l'illustration: comme précédent



* Des techniques à base de graphe, élégantes mais chères computationellement.

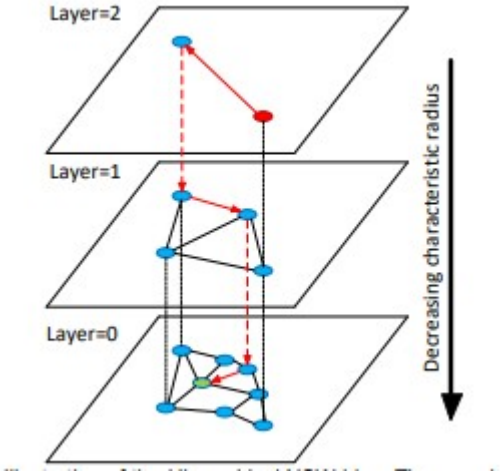
HNSW[10] est un des algorithmes les plus importants.

Le **HNSW (Hierarchical Navigable Small World)** est un algorithme avancé conçu pour effectuer des recherches Approximate Nearest Neighbour (ANN) de manière rapide et efficace dans des ensembles de données de grande dimension. Il s'appuie sur une structure de graphe hiérarchique où les points de données sont organisés en plusieurs niveaux, chaque niveau fournissant une représentation simplifiée et moins dense de l'espace global.

Le fonctionnement de HNSW repose sur trois principes clés. Tout d'abord, l'algorithme construit une hiérarchie de graphes, chaque niveau étant un sous-ensemble des données. Les couches supérieures contiennent très peu de points et offrent une vue globale et grossière de l'espace, tandis que les couches inférieures sont de plus en plus détaillées et contiennent davantage de points. Cette organisation permet une navigation progressive vers les données pertinentes, en commençant par les couches les plus abstraites et en affinant la recherche au fur et à mesure de la descente.

Ensuite, chaque graphe au sein des couches est structuré selon le principe du "Small World". Cela signifie que chaque point est connecté à un ensemble limité de voisins, choisis de manière à minimiser les distances entre eux. Ces connexions, tout en restant localement pertinentes, permettent également de naviguer rapidement dans des régions éloignées de l'espace des données. Ce mécanisme assure une transition fluide entre différentes zones de l'espace et garantit une exploration efficace.

La recherche commence au sommet de la hiérarchie, à la couche la plus abstraite, où les connexions globales permettent de localiser rapidement une zone approximative contenant les voisins recherchés. L'algorithme descend ensuite couche par couche, affinant progressivement la recherche. Chaque niveau utilise les connexions locales pour réduire la distance entre la requête et les points cibles. Une fois que l'algorithme atteint la couche la plus basse, qui contient tous les points de données, il finalise la recherche en explorant les voisins proches pour garantir un résultat précis.



10 https://arxiv.org/pdf/1603.09320

L’efficacité de HNSW repose sur sa capacité à limiter le nombre de calculs nécessaires pour trouver les voisins approximatifs. Sa structure hiérarchique permet de réduire la taille de l’espace de recherche à chaque étape, tandis que les connexions "Small World" assurent une navigation rapide et ciblée. L’idée de passer par un graphe étend la structure classique du Binary Search Tree et en décuple les bénéfices. Bien que l’approche soit approximative, elle offre souvent des résultats très proches de ceux obtenus par des méthodes exactes, mais avec une vitesse de traitement bien supérieure.

D’autres algorithmes continuent régulièrement d’être publiés[11] (preuve que les questions d’optimisation ont encore un bel avenir devant elles) : chaque milliseconde aide !

c) coûts de requête et scalabilité

A l’heure du Big data, le stockage et la récupération de données devient un enjeu majeur à son exploitation : la requête doit être rapide, quelle que soit la taille de la base de données, au risque que l’information enfouie aux tréfonds des serveurs ne soit plus exploitable. Les bases de données dépassent maintenant les milliards de documents, et là où avant, des millièmes de secondes de performances sur une requête pouvaient paraître négligeable, puisque la rapidité de requête dépend du nombre d’objets indexés, cela prend une importance cruciale.

Les BDD sont désormais réparties sur plusieurs serveurs voire data centers entiers. On parle de Péta octets. Les BDD et leurs développeurs ont donc redoublé d’ingéniosité pour accélérer les choses. Parmi les techniques employées, on peut citer :

*L’ optimisation dynamique des requêtes. Par exemple, les étapes de traitement sont souvent combinées : le système peut appliquer un filtre défini dans une clause WHERE avant de procéder à une jointure, ce qui réduit immédiatement la quantité de données à manipuler. De même, les optimisations incluent le réordonnancement des opérations, afin que les étapes les plus coûteuses soient exécutées en dernier, et l’utilisation de caches et de buffers pour accélérer l’accès aux données fréquemment consultées.

* Pour les bases de données modernes qui gèrent des volumes massifs, comme Google BigQuery ou Amazon Redshift, les recherchers sont souvent exécutées de manière distribuée. Cela signifie que les données sont divisées en partitions, chacune traitée indépendamment par un nœud du système, avant que les résultats intermédiaires ne soient agrégés pour produire une réponse finale. Cette architecture parallèle, combinée aux algorithmes d’optimisation et de traitement des requêtes, permet aux bases relationnelles de répondre rapidement à des requêtes, même sur des ensembles de données gigantesques. On parle aussi de sharding.

Souvent quand on parle de scalabilité, on désigne un phénomène réparti en deux dimensions. La **scalabilité verticale** consiste à améliorer les performances en augmentant les ressources d’une machine individuelle, comme l’ajout de mémoire, de puissance de calcul ou de stockage. HNSW, avec sa structure hiérarchique et ses graphes imbriqués, peut tirer parti d’une telle scalabilité en profitant de processeurs plus rapides et d’une mémoire vive étendue pour maintenir plus de données

11 https://github.com/google-research/google-research/tree/master/scann

en mémoire et traiter des requêtes plus complexes. Cependant, cette approche a ses limites : elle finit par rencontrer des plafonds matériels et des coûts d'équipement élevés.

En revanche, la **scalabilité horizontale** repose sur l'ajout de machines supplémentaires pour répartir la charge de travail. HNSW peut être adapté à cette approche grâce à la partition des données : chaque machine peut gérer un sous-ensemble des couches ou des graphes, permettant ainsi une recherche distribuée. Lorsqu'une requête est exécutée, elle peut être propagée entre les nœuds pour affiner les résultats de manière collaborative. Cette approche est particulièrement avantageuse pour des bases de données vectorielles massives dans des environnements distribués, tels que des systèmes de recommandation ou de recherche d'images à l'échelle industrielle.

À cela j'ajouterais la **« scalabilité interne »** qui repose à choisir les outils et stratégies les plus efficaces et adaptées à son cas d'usage. L'optimisation des BDD est un des combats invisibles les plus féroces de notre époque[12].

TODO : donner des chiffres

TODO conclusion : à chaque cas d'usage sa base de données.

12 https://people.csail.mit.edu/bradley/BenderKuszmaul-tutorial-xldb12.pdf J'ai trouvé ce diaporama aux illustrations tout simplement incroyables ; en cas de doute toujours faire confiance au MIT.