

NoSQL et bases de données graphes

Ce cours aborde deux types spécifiques de bases de données NoSQL

- MongoDB, une base de données - document
- Neo4j, une graph database.
- et nous verrons aussi les vector store qui sont important pour l'IA et le NLP

Qu'est-ce qu'une base de données ?



On entend parfois le mot base de données pour désigner un fichier Excel.

Comment peut-on mettre quelque chose d'aussi simple qu'un fichier CSV ou Excel au même niveau que ces merveilles d'ingénierie que sont PostgreSQL, Weaviate, MongoDB, Neo4j, Redis, MySQL etc...

J'ai donc demandé à mon ami GPT-4 de me donner une définition d'une base de données :

En termes simples : "Une base de données est comme un carnet intelligent ou un système de classement qui vous aide à garder une trace de nombreuses informations et à trouver exactement ce dont vous avez besoin en un rien de temps."

ce qui inclut définitivement les fichiers CSV, les fichiers Excel, les fichiers JSON, les fichiers XML et tant d'autres formats simples basés sur un seul fichier.

Si nous cherchons la définition d'une base de données dans une source de connaissance plus classique et vénérable, comme l'[Encyclopédie Britannica](#), nous obtenons :

base de données, toute collection de données ou d'informations spécialement organisée pour une recherche rapide par ordinateur. Les databases sont structurées pour faciliter le storage, l'extraction, la modification et la suppression des données.

Voir aussi l'article Base de données sur [Wikipedia](#).

Très intéressant. Nous ne parlons plus seulement de trouver rapidement l'information (la partie **recherche**) mais aussi de :

- stockage
- modification
- suppression
- Administration.

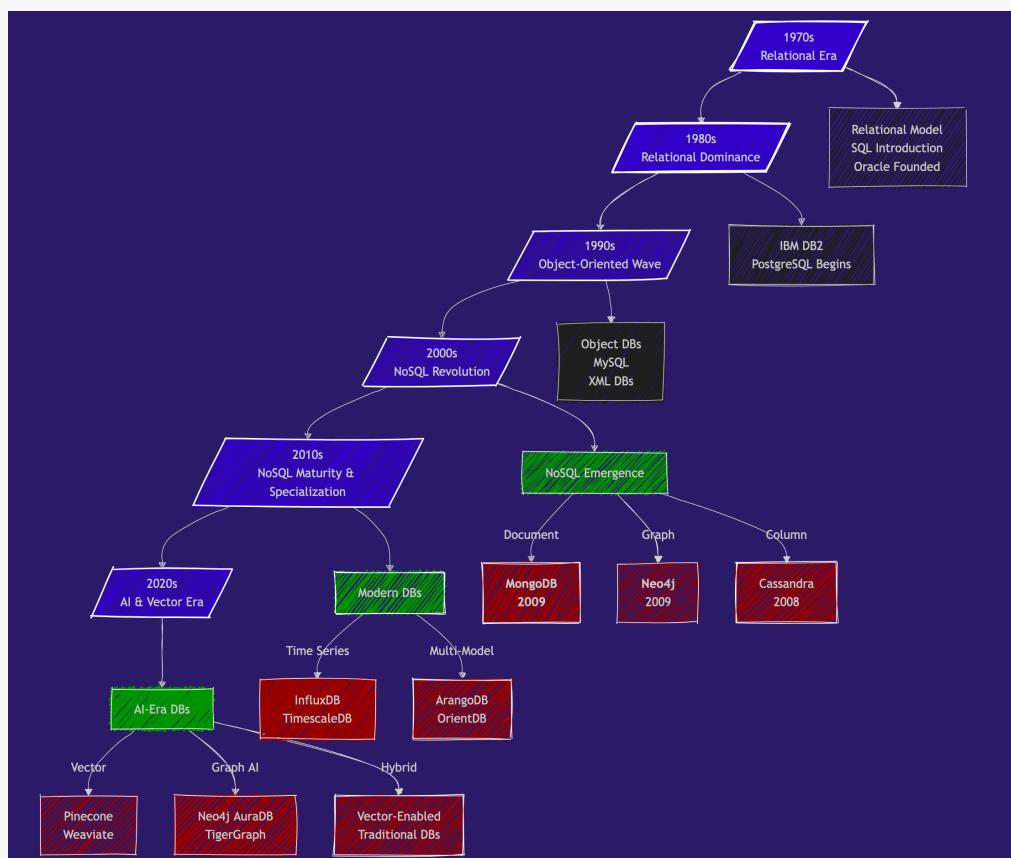
C'est là qu'un simple fichier de type tableur ne correspond plus à l'objectif.

Ce que l'on attend d'un SGBD

Un système de gestion de base de données (SGBD) se distingue d'un simple spreadsheet par plusieurs fonctionnalités essentielles.

Feature	Description	Excel	SGBD
Data Storage and Retrieval	Stocke les données de manière organisée et les récupère selon les besoins.	✓	✓
Data Manipulation	Permet d'ajouter, modifier ou supprimer des données.	✓	✓
Data Querying	Permet de poser des questions complexes (requêtes) sur les données.	✓	✓
Data Organization	Structure les données dans des formats comme des tables, documents ou graphes pour faciliter la gestion.	✓	✓
Data Sharing	Permet à plusieurs utilisateurs ou applications d'utiliser la base de données simultanément.	✓	✓
Data Security	Protège les données contre les accès non autorisés ou la corruption.	✓	✓
Concurrency Control	Gère plusieurs utilisateurs modifiant les données en même temps sans conflits.	✓	✓
Backup and Recovery	Garantit que les données ne sont pas perdues et peuvent être restaurées en cas de défaillance.	✓	✓
Data Integrity	Garantit que les données restent précises, cohérentes et fiables.	✓	✓
Performance Optimization	Fournit des outils pour optimiser la vitesse et l'efficacité des retrievals et updates de données.	✓	✓
Support for Transactions	Garantit qu'un groupe d'opérations (transactions) est complété entièrement ou pas du tout. ACID compliance	✓	✓

Un brève histoire des bases de données



1970s - Le début de l'ère relationnelle

- 1970 : [Edgar Codd](#) publie "A Relational Model of Data for Large Shared Data Banks"
- 1974 : IBM développe System R, le premier prototype de DBMS SQL
- 1979 : Oracle lance la première implémentation SQL commerciale

1980s - La domination du relationnel

- 1989 : Début du développement de Postgres (maintenant [PostgreSQL](#)) à UC Berkeley
 - la base de données SQL de référence.
 - maintenant peut gérer le no-sql & vector,
 - nombreuses extensions (http, postgis, ...).
 - performances exceptionnelles.
 - et OPEN SOURCE (gratuit, efficace et sécurisé).

1990s - La vague orientée objet

- 1991 : Les bases de données Object-Oriented gagnent en attention.
La plupart des OODBs des années 90 ne sont plus utilisées. Mais elles ont influencé l'évolution des databases SQL et NoSQL.

- **1995** : MySQL est publié en open source

2000s - Le début de la révolution NoSQL

- 2 papiers importants qui posent les bases des systèmes NoSQL : [BigTable paper](#) (Google, 2004) et [Dynamo paper](#) (Amazon, 2007)

Et en **2009**, 2 nouvelles databases sont lancées :

-  MongoDB
-  Neo4j

Tadaaaah !

Pourquoi à ce moment-là ?

L'essor du world wide web (myspace - 2003 😍, youtube - 2005) et l'augmentation massive de l'échelle des applications de plusieurs ordres de grandeur.

Soudainement, nous avons des millions de personnes qui tentent simultanément d'accéder et de modifier des Terabytes de données en quelques millisecondes.

Les databases relationnelles ne peuvent pas suivre l'échelle des applications, la nature chaotique des données non structurées et les exigences de vitesse.

La promesse du NoSQL est le **volume et la vitesse**.

2010s - Le NoSQL arrive à maturité & Spécialisation - Big Data et Databases Spécialisées

- **Big Data Databases** : Des systèmes comme **Apache Hadoop** (2006) et **Apache Spark** (2009) ont permis le traitement de données à très grande échelle.
- Les **Graph Databases** gagnent en popularité avec des cas d'usage comme la détection de fraude, les knowledge graphs, et la gestion de la chaîne d'approvisionnement. Neo4j et Amazon Neptune deviennent des acteurs clés.
- **Time-Series Databases (ex : InfluxDB, TimescaleDB)** : conçues pour les systèmes de monitoring : IoT, logs, ...
- **Cloud Databases** : services managés comme Amazon **RDS**, Google **BigQuery**, ou **Snowflake**

et entre-temps, en 2013, les containers **Docker** révolutionnent le déploiement des databases

2020s : IA, Vector Databases, et Besoins Temps Réel

- **Vector Databases**  (ex : Pinecone, Weaviate, Qdrant, Milvus, Faiss, ...) :
 - Gèrent les vector embeddings de haute dimension utilisés dans les applications AI/ML
- et aussi :
 -  **Graph + AI** :  knowledge graphs et LLMs.
 - Multi-Model Databases qui supportent plusieurs modèles de données (document, graph,

key-value) dans un seul système.

- Real-Time Analytics : optimisés pour le streaming de données en temps réel et l'analytique.
- Serverless Databases

Tendances Actuelles (2025)

La vector search est en plein essor. Les capacités de vector search sont intégrées dans la plupart des DBMS existants, y compris PostgreSQL, MongoDB et Neo4j.



En Bref

- 1989 : Lancement de PostgreSQL
- 2009 : Lancement de MongoDB et Neo4j
- 2024 : les vector databases sont en vogue tandis que les databases plus anciennes intègrent la vector search

Catégories principales de databases aujourd'hui

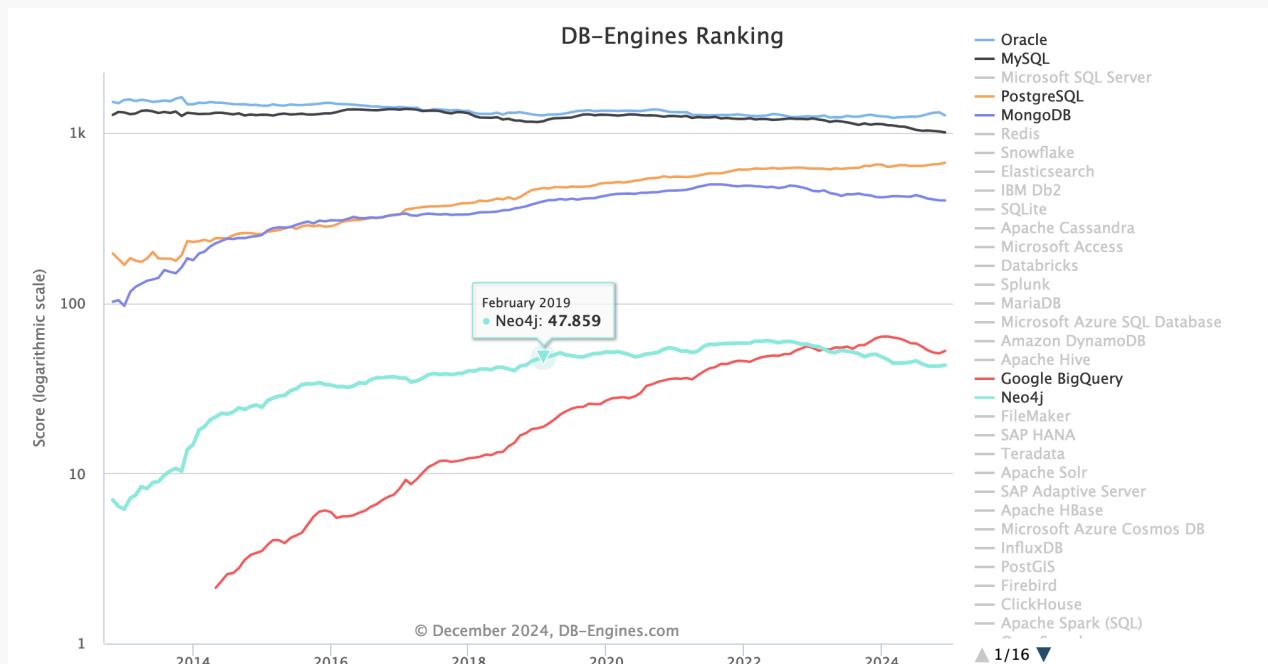
Nous avons de nombreuses bases de données parmi lesquelles choisir. Tout dépend de l'échelle, de la nature de l'application, du budget, etc.

Type de base de données	Objectif	Exemples	Application
Relational - SQL	Schema fixe	PostgreSQL, MySQL, Oracle	Transactions, normalisation
Document Stores	Schema flexible, documents type JSON	MongoDB, CouchDB	Applications web, gestion de contenu
Graph Databases	Données centrées sur les relations	Neo4j, ArangoDB	Réseaux sociaux, moteurs de recommandation
Key-Value Stores	Recherches simples et rapides	Redis, DynamoDB	Caching, gestion des sessions
Vector Databases	Recherche par similarité, AI embeddings	Pinecone, Weaviate	Applications IA, recherche sémantique
Column-Family Stores	Données colonnes larges, haute scalabilité	Cassandra, HBase	Time-series, applications big data
Time-Series Databases	Données ordonnées dans le temps	InfluxDB, TimescaleDB	IoT, systèmes de monitoring

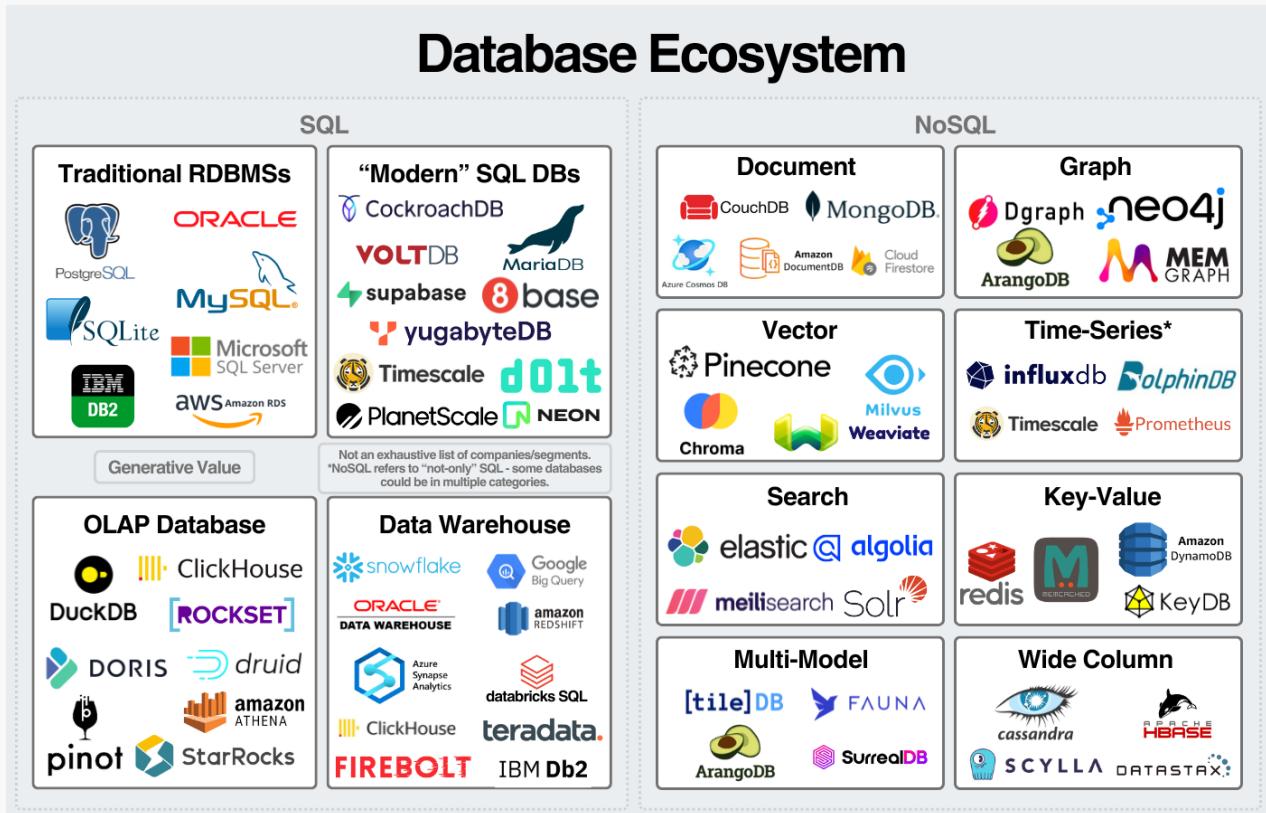
Écosystème

Consultez le classement de toutes les bases de données sur <https://db-engines.com/en/ranking>

Les tendances : https://db-engines.com/en/ranking_trend



De très nombreux acteurs :



source : <https://www.generativevalue.com/p/a-primer-on-databases>

Voir aussi cette carte interactive qui recense tous les acteurs en 2023.

<https://mad.firstmark.com/>

Prenons du recul et comparons une base de données relationnelle SQL et une base de données NoSQL.

SQL

Une base de données relationnelle SQL (et variations de SQL):

- utilise un **schema** prédéfini : la structure des données (colonnes, types de données, etc.) est fixe.
- les bdd relationnelles sont efficaces pour les requêtes complexes, pour les transactions et pour assurer la cohérence des données (conformité ACID).

Une **base de données relationnelle** peut être comparée à une collection de spreadsheets bien organisés (tables) où chaque colonne est définie, et les tables sont interconnectées.

Les tables ont des colonnes et des lignes de données. Chaque table a une clé unique appelée **primary key**.

La conception du schema repose sur le concept de normalisation / dénormalisation. Dans une bdd

normaliséem une information n'existe que dans une seule table. Une bdd normalisée valide une série de règles appelées NF1, NF2, ...

Concept important de **Normalisation** : une information n'existe qu'à un seul endroit et un seul.

Hiérarchie d'une base de données SQL :

- Une clé primaire unique pour chaque élément d'une table donnée
- Une **foreign key** lie une table à une autre table
- Un **row** contient la valeur d'une *entity*
- Un **column** est un **attribute** ou une propriété de l'*entity*
- Un **table** contient toutes les entités regroupées dans une structure fixe de colonnes

Les databases SQL = rigides, contrôlées, données cohérentes, stables. Peuvent être complexes.

NoSQL

Une **base de données non-relationnelle** est un système de dossiers flexible où vous pouvez stocker des éléments - de différentes formes - sans règles strictes.

Les bases de données NoSQL utilisent des modèles de données non relationnels, tels que :

- Clé-valeur (ex : Redis, DynamoDB)
- Document (ex : MongoDB, Couchbase)
- Colonnes larges (ex : Cassandra, HBase)
- Graphes (ex : Neo4j, Arangodb)

Les Bdd NoSQLI recoupent des systèmes d'organisations des données très différents. Une representation des données sur la base d'un graphe de données est très différente d'une representation en mode document de type JSON ou la variabilité ddu contenu est sans limite.

Dans tous les cas on parle de **schema flexible** avec des données non structurées ou semi-structurées. La flexibilité du schéma permet d'ajouter de nouveaux types de données ou de modifier la structure des données sans nécessiter de migrations complexes.

Les bdd NoSQL sont idéales pour - haute scalabilité: capacité d'un système à gérer une augmentation significative de la charge de travail (données, utilisateurs, transactions) tout en maintenant des performances optimales. - Scalabilité verticale : augmenter les ressources d'un seul serveur (CPU, RAM, disque dur) pour gérer plus de charge. - Scalabilité horizontale : ajouter des serveurs supplémentaires au cluster pour répartir la charge. - RéPLICATION et partitionnement: les données sont réparties sur plusieurs serveurs ou nœuds (sharding pour MongoDB) - RéPLICATION : Les données sont copiées sur plusieurs serveurs pour assurer la disponibilité et la résilience en cas de défaillance d'un serveur. - Partitionnement (sharding) : Les données sont divisées en fragments (shards) et réparties sur plusieurs serveurs.

Comparaison réPLICATION et partitionnement

Bien que les bases de données SQL et NoSQL prennent en charge la réPLICATION et le sharding, il existe des différences clés dans leur mise en œuvre et leur utilisation :

Aspect	Bases de données SQL	Bases de données NoSQL
RéPLICATION	Souvent synchrone ou asynchrone, avec une forte cohérence possible.	Souvent asynchrone, avec <i>consistance éventuelle</i> (eventual consistency).
Sharding	Plus complexe à mettre en œuvre, nécessite des outils externes ou des configurations manuelles.	Intégré nativement dans de nombreuses bases de données NoSQL (ex : MongoDB, Cassandra).
Flexibilité	Moins flexible en raison du schéma rigide et des contraintes ACID.	Très flexible, adapté aux données non structurées et aux modèles de données variés.
Scalabilité horizontale	Possible, mais souvent plus difficile à gérer en raison des jointures et des transactions distribuées.	Conçu dès le départ pour la scalabilité horizontale, avec des architectures distribuées natives.

Terminologie

Dans une bdd NoSQL de type document comme MongoDB

- Une **collection** est un ensemble de **documents**.
- Un document est une combinaison de keys : values .
- Chaque collection a une clé unique (le champ `_id`)
- Dans un document, vous pouvez avoir des sous-documents imbriqués. Par exemple: Une personne peut avoir plusieurs téléphones, adresses email, emplois, ...

Tous les documents d'une collection sont **similaires en structure** mais n'ont pas besoin d'être exactement identiques.

Il n'y a pas de concept de normalisation.

Hiérarchie de MongoDB - base de données NoSQL - MongoDB

- Une **base de donnée** contient des **collections**
- Une **collection** contient tous les **documents** ~~~ table
- Un **document** est l'entité qui contient les données ~~~ enregistrement
- Un sous-document (imbriqué) est un document **à l'intérieur** d'un document parent
- Un **field / champs** est un attribut ou une propriété du document -> colonne

MongoDB - SQL :

- Un **document** est un enregistrement

- Une `Collection` est une table
- Un `Field` est une colonne

MongoDB	SQL base de données
base de données	base de données
Collection	Table
Document	Record/Row
Field	Column
Embedded Document	Foreign Key
<code>_id</code>	Primary Key
<code>\$lookup</code>	JOIN

Un `index` reste un `index`

Schema-less - schema dynamique

Dans quelles situations les données changent elles si souvent qu'il nous faudrait un type spécial de base de données ?

L'exemple le plus courant d'application NoSQL est celui d'un réseau social.

- profils utilisateurs
- les posts contiennent toutes sortes de contenu
- timeline, followers, etc

Yuka et les produits alimentaires

Prenons l'exemple d'une start-up comme [Yuka](#)

Make the right choices for your health

Yuka deciphers product labels and analyzes the health impact of food products and cosmetics.



Selon leurs propres mots : *Yuka déchiffre les étiquettes des produits et analyse l'impact sur la santé des produits alimentaires et cosmétiques.*

Sa base de donnée sous-jacente est la [open food facts base de données](#), une base de donnée de produits alimentaires faite par tous, pour tous, avec plus de 3,5 millions de produits alimentaires.

Regardez par exemple les informations pour [Nutella](#) et celles pour... [Baguette](#)

Voir aussi cet article qui explore le dataset avec python pandas :

<https://medium.com/@achrafelkhanjari99/a-deep-dive-into-the-open-food-facts-dataset-56259b162ac5> (disponible en pdf dans la repo Github)

Avec autant de produits, les informations disponibles et les informations associées (Packaging, Impact Carbone) ainsi que la diversité des réglementations (UE, US, UK, ... etc) varient constamment.

Les données sont constamment mises à jour alors qu'il faut conserver l'historique des changements et des nouveautés.

- de nouvelles données deviennent disponibles au fur et à mesure que les acteurs mettent en place la collecte de données. Pensez traçabilité, sécurité, etc
- les nouvelles réglementations imposent plus de données
- l'actualité, les tendances sociales et les centres d'intérêt changent rapidement (sans gluten, thon et mercure, pesticides, ...)

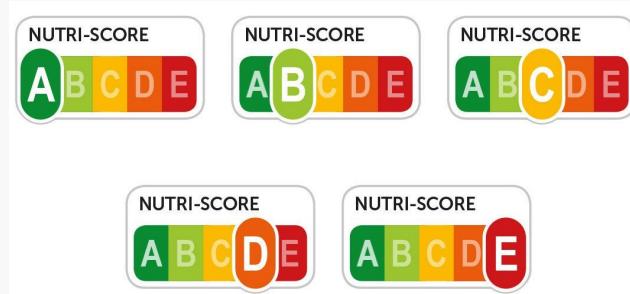
Vous commencez donc votre base de données avec un schema simple qui inclut :

- nom, définition, image, description
- valeurs nutritionnelles
- ingrédients

Mais le schéma devient de plus en plus complexe à mesure qu'évoluent les données , les produits et les services de l'entreprise.

Nutriscore

Prenons par exemple, le label Nutriscore :



Le [Nutriscore](#) a récemment évolué avec une nouvelle version plus stricte. Vous avez donc besoin des nouveaux labels Nutriscore tout en gardant l'ancien car tous les produits n'implémentent pas le nouveau Nutriscore. Certaines entreprises ont même complètement abandonné l'étiquetage.

Vous avez commencé avec une table Nutriscore dans une base de données SQL :

```
product_id: key  
nutriscore_label : array[A,B, ..., E]
```

SQL

donc votre table Nutriscore nécessite une nouvelle colonne :

```
product_id: key  
nutriscore_label: array[A,B, ..., E]  
nutriscore_new_label: array[A,B, ..., E]
```

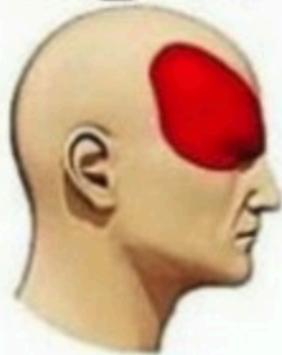
SQL

Cependant, la plupart des produits n'ont pas encore de nouveau label nutriscore.

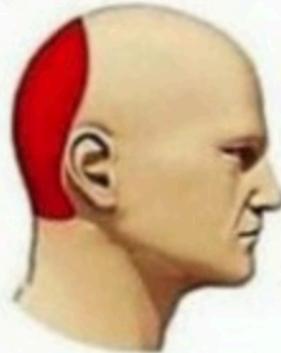
Et vous vous retrouvez avec beaucoup de null values dans cette colonne `nutriscore_new_label` et les null values sont à éviter 😠😠😠.

Types of Headaches

Migraine



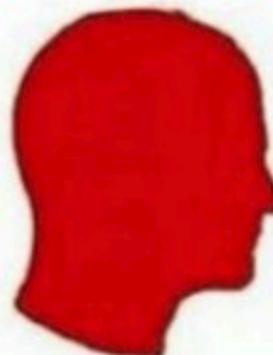
Hypertension



Stress



is **NULL**



Vous pouvez aussi normaliser la table et introduire une colonne version du Nutriscore pour aider avec les Null values.

```
product_id: key  
nutriscore_label: array[A,B, ..., E]  
nutriscore_version: Int
```

SQL

Dans les deux cas, vous devez changer toutes vos requêtes SQL dans votre base de code .

Douleur, soucis, migraines, bugs et coûts supplémentaires \$ \$ \$.

Introduction à la Flexibilité du schéma

La **Flexibilité du schéma** dans MongoDB et d'autres bases de données NoSQL fait référence à la capacité de stocker des données sans nécessiter un schéma prédéfini. Cela signifie que les documents d'une même collection peuvent avoir différents champs / attributs, structures et types de données.

La Flexibilité du schéma aide à gérer les **unknown unknowns** dans un monde qui change rapidement.

Présence et type de données

Dans MongoDB : Vous pouvez simplement ajouter un nouvel élément Nutriscore aux produits alimentaires :

Pas de Nutriscore

```
{  
  "product_id": 198273,  
  "name": "Chocapic",  
}
```

JavaScript

Le Nutriscore est ajouté, il suffit d'ajouter un field au document du produit

```
{  
  "product_id": 198273,  
  "name": "Chocapic",  
  "Nutriscore": "C"  
}
```

JavaScript

Une nouvelle version du Nutriscore arrive, il suffit d'ajouter le label Nutriscore comme un dictionary avec les versions comme keys :

```
{  
  "product_id": 198273,  
  "name": "Chocapic",  
  "Nutriscore": {  
    "v1": C,  
    "v2": D,  
  }  
}
```

JavaScript

Plusieurs représentations peuvent donc **coexister** dans la même base de données :

- Pas de nutriscore
- Un seul nutriscore comme *string*
- Un dictionary nutriscore comme document nested / embedded

Données imbriquées

- Pour être efficace, une base de donnée SQL doit être normalisée. Pour les données complexes, on risque de se retrouver avec beaucoup de tables.
- MongoDB permet d'**imbriquer** les données de façon naturelle

Un bon exemple est celui de l'adresse d'une personne

On peut avoir dans la même base, des personnes qui n'ont pas d'adresse, une adresse ou plusieurs. Et ces plusieurs adresses ont des rôles différents : résidence principale, secondaire, etc
....

Si on utilise un format JSON pour représenter ces 3 cas, on a naturellement

```
// Une personne sans adresse enregistrée
{
    "_id": "1",
    "name": "Anita Sharma",
    "age": 29,
    "email": "anita.sharma@example.com"
}

// Une personne avec une seule adresse comme simple dictionnaire
{
    "_id": "2",
    "name": "Rahul Verma",
    "age": 42,
    "email": "rahul.verma@example.com",
    "address": {
        "type": "home",
        "street": "12 MG Road",
        "locality": "Indiranagar",
        "city": "Bengaluru",
        "state": "Karnataka",
        "pincode": "560038",
        "country": "India"
    }
}

// Une personne avec plusieurs adresses comme liste de dictionnaires
{
    "_id": "3",
    "name": "Priya Singh",
    "age": 35,
    "email": "priya.singh@example.com",
    "addresses": [
        {
            "type": "home",
            "street": "45/2 Lajpat Nagar",
            "locality": "Central Market",
            "city": "Delhi",
            "state": "Delhi",
            "pincode": "110002",
            "country": "India"
        }
    ]
}
```

```

        "city": "New Delhi",
        "state": "Delhi",
        "pincode": "110024",
        "country": "India"
    },
    {
        "type": "work",
        "street": "4th Floor, Tower B",
        "locality": "DLF Cyber City",
        "city": "Gurugram",
        "state": "Haryana",
        "pincode": "122002",
        "country": "India"
    }
]
}

```

En SQL, il faudrait avoir une table adresse et une relation many-to-many entre la table personne et la table personne, donc une table intermédiaire pour la jointure.

Conséquences de la flexibilité du schéma

La flexibilité du schéma impacte chaque étape du cycle de vie d'une base de données

- **Design** : sans règles, tout devient possible. Les choix de conception sont dictés par l'application. La façon dont les données sont *consommées* dicte la structure des données dans la base de données.
- **Development** : avec un schema flexible, les changements peuvent être implémentés plus rapidement.
- **Maintenance** : l'inconvénient est la nécessité de gérer une organisation et des types de données historiques

Une prudence supplémentaire est nécessaire pour éviter le chaos et les **data inconsistencies**. La performance des requêtes peut être affectée si les changements dans la structure des données conduisent à un indexing inefficace ou incohérent.

Avec les databases NoSQL, le coût de l'implémentation des changements dans la nature des données est déplacé de la base de données vers le niveau applicatif.

Cependant, des inconsistencies dans la base de données peuvent toujours survenir si plusieurs applications interagissent différemment avec la même base de données.

En bref, la flexibilité du schéma doit être utilisée avec précaution et uniquement lorsque c'est utile et justifié.

Quand choisir NoSQL (base de données - document) plutôt que SQL ?

Alors quand est-ce qu'une base de données document NoSQL est un meilleur choix que SQL ?

- Vos données correspondent naturellement à une structure de **documents** plutôt qu'à des tables strictes
 - Vous voulez stocker les données liées ensemble plutôt que de les répartir dans des tables pour accélérer la récupération d'information: les requêtes sont plus simples, il y a moins de jointures, et simplicité du code.
- Itération Rapide : Votre schéma doit évoluer rapidement et vous privilégiez la vitesse de développement à la stricte cohérence des données
 - Applications et exigences de données qui changent rapidement
 - Startups en phase initiale où le modèle de données n'est pas encore pleinement compris
- également : A/B testing de différentes fonctionnalités qui peuvent nécessiter différentes structures de données
- Scalabilité et Performance
 - Conçu pour le **horizontal scaling** avec support intégré du **sharding** (distribution des données sur plusieurs serveurs).
 - Adapté à la gestion d'applications à grande échelle, haut débit et géographiquement distribuées.

MongoDB scale out, tandis que PostgreSQL scale up.

- Utilise un modèle de document flexible de type JSON (BSON), le rendant idéal pour les données hiérarchiques ou semi-structurées.
 - Réduit le besoin de joins complexes, car les données liées peuvent être embeddées dans un seul document.

MongoDB excelle dans :

- Les applications avec des données non structurées ou semi-structurées.
- Les charges de travail à haute vitesse nécessitant des changements rapides de schéma.
- Les cas d'utilisation nécessitant un horizontal scaling dans des environnements distribués.

Performances

En termes de performances, la comparaison favorise la plupart du temps PostgreSQL par rapport à MongoDB. voir [MongoDB Vs PostgreSQL: A comparative study on performance aspects](#)

Et cet autre article, [MongoDB vs PostgreSQL: Choosing the Best Database for Your Needs](#), le résume bien :

_MongoDB brille dans les scénarios nécessitant le développement d'applications logicielles qui traitent divers types de données de manière évolutive. Il est particulièrement adapté aux projets qui doivent supporter un développement itératif rapide et faciliter la collaboration de nombreuses équipes.

En bref

- Choisissez MongoDB si votre application a un modèle de données simple et gère un très grand volume de données
- Choisissez PostgreSQL si votre application a une logique métier complexe qui repose sur des transactions.

Qu'en est-il des bases de données graphes ?

Les bases de données SQL sont appelées bases de données relationnelles.

Dans une base de données SQL, la relation entre les tables est explicitement définie par les foreign keys entre les tables.

- Product -> Vegetables -> Location, Origin
- Product -> Vegetables -> Organic / not Bio
- Product -> nutrition (sugar, fat etc)
- Product -> Nutriscore labels
- Product -> ConsumeBy

Et l'ERD pour une telle base de données indique uniquement la cardinalité de la relation :

- 1 to 1
- 1 to many



Quand vous demandez à un LLM de générer un diagramme pour une base de données de produits, il ajoute naturellement des informations significatives aux relations entre les tables.



[Mermaid diagram](#)

Relationships

Les bases de données graphes comme Neo4j sont centrées sur la *signification* des **relations** entre entités.

Les **relations** sont aussi importantes que les données elles-mêmes et sont stockées explicitement.

Ces relations ont leurs propres propriétés et sont stockées comme des connexions.

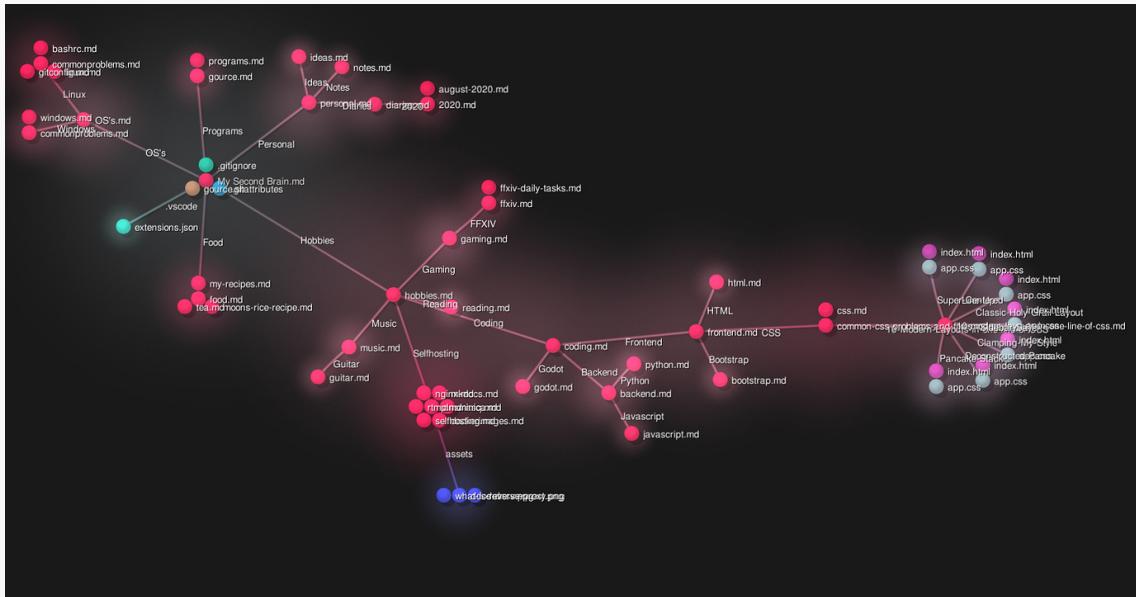
Alors qu'en SQL :

- Les relations sont implicites via des clés étrangères
- Doivent être reconstruites via des JOINs
- Deviennent exponentiellement plus complexes et lentes lorsque vous suivez plusieurs niveaux de connexions

C'est pourquoi Neo4j excelle dans les questions comme :

- "Trouvez tous les amis d'amis qui aiment courir et qui vivent à Paris"
- "Quel est le chemin le plus court entre la personne A et la personne B ?"
- "Qui sont les personnes les plus influentes dans ce réseau ?"

Voici un exemple de graphe de connaissances avec Obsidian



conclusion

Donc les bases de données SQL, dites relationnelles, sont excellentes pour les structures de données qui ne changent pas souvent et où les relations entre objets sont stables.

Les bases de données No-SQL comme MongoDB : excellentes quand les spécifications des données évoluent rapidement ou ne sont pas définitives, schéma flexible, grande échelle

Bases de données graphes : la relation est clé. Ce n'est pas seulement qu'il y a une relation mais aussi quelle est la nature de cette relation.

- NoSQL - Document concerne l'évolutivité et l'évolution des données.
- Graph concerne la réponse à des questions spécifiques, la découverte de différents types de signification et d'aperçus dans les données.

Dans cette session, vous avez appris :

- RDBS et l'histoire des bases de données
- Un aperçu des différents types de bases de données
- Base de données relationnelle vs non relationnelle
- Schéma flexible dans les bases de données NoSQL
- Pourquoi et quand choisir MongoDB plutôt que SQL

