# Atelier 1

**Partie 1 : Recherche sur les bases de données relationnelles SQL et NoSQL**

**1. Bases de données relationnelles (SQL)**

Les bases de données relationnelles sont fondées sur un modèle structuré, organisé en tables avec des colonnes et des lignes. Elles sont adaptées aux systèmes où l’intégrité des données et les relations entre elles sont primordiales.

* **Modèle de données structuré** :
  + Dans les bases de données relationnelles, les données sont stockées dans des **tables** qui définissent un schéma rigide avec des colonnes (champs) et des lignes (enregistrements).
  + Les **relations** entre les tables sont établies via des clés primaires et des clés étrangères, permettant de lier des données entre différentes tables. Par exemple, une table "Clients" pourrait être liée à une table "Commandes" via une clé étrangère.
* **Utilisation du langage SQL** :
  + **SQL (Structured Query Language)** est un langage de manipulation des données utilisé pour gérer et interroger les données dans une base de données relationnelle.
  + Il permet d’effectuer des opérations telles que la **création** (CREATE), la **lecture** (SELECT), la **mise à jour** (UPDATE), et la **suppression** (DELETE) de données. Les requêtes SQL peuvent être complexes, avec des **jointures** pour lier plusieurs tables, des filtres pour affiner les données, et des **transactions** pour garantir l’intégrité.
* **Conformité ACID** :
  + Les bases de données relationnelles sont conçues pour respecter les propriétés **ACID** qui garantissent l’intégrité et la cohérence des données :
    - **Atomicité** : Toute opération (ou transaction) est réalisée dans son intégralité ou pas du tout. Si une transaction échoue, les modifications sont annulées.
    - **Cohérence** : Les transactions respectent les règles et contraintes définies, assurant que la base de données reste dans un état valide.
    - **Isolation** : Les transactions concurrentes s'exécutent indépendamment, sans interférence les unes avec les autres.
    - **Durabilité** : Une fois une transaction validée, les changements sont conservés même en cas de panne système.
* **Exemples de bases de données relationnelles** :
  + **MySQL** : Populaire pour les applications web, il est souvent utilisé pour les sites web, les applications e-commerce, et les plateformes de gestion de contenu.
  + **PostgreSQL** : Connue pour sa conformité stricte à ACID et ses fonctionnalités avancées (comme les types de données personnalisés), elle est couramment utilisée dans les applications nécessitant une forte intégrité des données.
  + **Oracle** : Utilisée dans des environnements d’entreprise, elle est réputée pour sa robustesse, ses fonctionnalités avancées de sécurité et sa gestion de gros volumes de données.

**2. Bases de données NoSQL**

Les bases de données NoSQL sont conçues pour des données moins structurées et des schémas flexibles. Elles sont idéales pour les applications nécessitant de grandes quantités de données non structurées ou semi-structurées, avec une forte scalabilité et des besoins de performance spécifiques.

* **Différents types de bases NoSQL** :
  + **Clé-valeur** : Les données sont stockées sous forme de paires clé-valeur, où une clé unique est associée à une valeur. Cela permet un accès rapide aux données et est souvent utilisé pour la mise en cache.
    - Exemple : **Redis**.
  + **Colonnes** : Ces bases organisent les données en colonnes au lieu de lignes, facilitant le traitement en parallèle de grandes quantités de données. Elles sont idéales pour les analyses en temps réel et le traitement de données massives.
    - Exemple : **Cassandra**.
  + **Documents** : Les données sont stockées sous forme de documents (souvent au format JSON), permettant une grande flexibilité des schémas. Elles sont adaptées aux applications nécessitant une structure de données changeante.
    - Exemple : **MongoDB**.
  + **Graphes** : Ces bases de données sont optimisées pour représenter les relations entre les entités, utiles pour les réseaux sociaux, la recherche de parcours, ou les systèmes de recommandation.
    - Exemple : **Neo4j**.
* **Flexibilité du schéma et scalabilité horizontale** :
  + **Flexibilité du schéma** : Les bases NoSQL permettent de modifier la structure des données sans affecter le reste de la base, ce qui est utile pour les applications évolutives avec des structures de données changeantes.
  + **Scalabilité horizontale** : La plupart des bases NoSQL sont conçues pour une **scalabilité horizontale** (ajout de nouveaux serveurs pour répartir la charge), ce qui permet de gérer de grandes quantités de données et de trafic sans affecter la performance.
* **Conformité BASE** :
  + Contrairement aux bases relationnelles qui suivent ACID, les bases NoSQL adoptent souvent le modèle **BASE**, qui met l'accent sur la disponibilité et la tolérance aux pannes dans les systèmes distribués :
    - **Basic Availability** : Disponibilité garantie même en cas de panne.
    - **Soft state** : L’état de la base peut changer au fil du temps, même sans nouvelles entrées.
    - **Eventual consistency** : Les données deviennent cohérentes à terme, ce qui signifie que les mises à jour seront visibles de manière progressive pour les utilisateurs.
* **Exemples de bases de données NoSQL** :
  + **MongoDB** : Adaptée aux données semi-structurées, elle est souvent utilisée pour les applications nécessitant de la flexibilité dans la structure des données.
  + **Cassandra** : Conçue pour gérer de grandes quantités de données avec une haute disponibilité et une tolérance aux pannes.
  + **Redis** : Base clé-valeur rapide, utilisée pour des applications nécessitant des accès rapides, comme la mise en cache.

Cette recherche donne une vue d'ensemble des caractéristiques clés des bases SQL et NoSQL, permettant de mieux comprendre dans quels cas l’une peut être plus adaptée que l’autre pour répondre aux besoins spécifiques d'une application.

**Partie 2 : Cas d’usage**

**Cas métier sélectionné : Boutique en ligne**  
Nous avons choisi d'explorer le cas d'une boutique en ligne, un scénario pertinent pour analyser et comparer les bases de données relationnelles (SQL) et NoSQL. Cette boutique gère des produits, des clients, et des commandes, avec des besoins variés en termes de gestion des données et d'analyse transactionnelle.

**Exigences métier**

1. **Nature des données :**
   * Les données sont **structurées** (clients, produits, commandes) et comprennent également des éléments **semi-structurés** pour des aspects comme les préférences des utilisateurs ou les avis produits.
2. **Volume des données :**
   * La boutique en ligne est supposée gérer un **grand volume de données** en raison d'une large base de clients, un catalogue produit conséquent, et des historiques de commandes.
3. **Exigences en matière de performance :**
   * **Rapidité** : La boutique doit fournir des temps de réponse rapides pour les recherches de produits et la navigation des utilisateurs.
   * **Évolutivité** : L’application doit être capable de gérer une augmentation du trafic, notamment lors de périodes promotionnelles comme le Black Friday.
   * **Transactions complexes** : Les processus d'achat doivent respecter des règles strictes de gestion des transactions (paiement, mise à jour des stocks).
4. **Modifications fréquentes des schémas :**
   * Les structures de données peuvent évoluer régulièrement pour s'adapter à des besoins changeants, tels que l'ajout de nouvelles catégories de produits ou des fonctionnalités spécifiques comme des recommandations personnalisées.

**Résumé des défis à relever**

Le système doit concilier **intégrité des données** pour les transactions sensibles (paiements, commandes) et **flexibilité** pour intégrer des fonctionnalités modernes comme des recommandations basées sur des interactions en temps réel. Cette dualité en fait un excellent cas pour analyser les avantages et les inconvénients des bases relationnelles et NoSQL.

Ce cas sera analysé dans les sections suivantes à travers des expérimentations pratiques et une comparaison détaillée entre SQL et NoSQL.

!pip install pysqlite3

import sqlite3

sqlite3.version

# Importer les bibliothèques nécessaires

import sqlite3

import pandas as pd

# Créer une connexion SQLite en mémoire (ou sur un fichier si besoin)

conn = sqlite3.connect("ecommerce.db")  # Remplacez "ecommerce.db" par ":memory:" pour une base temporaire

cursor = conn.cursor()

print("Connexion à la base de données SQLite établie.")

# Charger le fichier CSV dans un DataFrame pandas

file\_path = "/content/E-commerce Dataset (1).csv"  # Remplacez par le chemin du fichier si différent

df = pd.read\_csv(file\_path)

# Aperçu des données

print("Aperçu des données :")

print(df.head())

# Nettoyage des données (si nécessaire)

df = df.fillna("")  # Remplacer les valeurs nulles par des chaînes vides

Aperçu des données :

Order\_Date Time Aging Customer\_Id Gender Device\_Type \

0 2018-01-02 10:56:33 8.0 37077 Female Web

1 2018-07-24 20:41:37 2.0 59173 Female Web

2 2018-11-08 08:38:49 8.0 41066 Female Web

3 2018-04-18 19:28:06 7.0 50741 Female Web

4 2018-08-13 21:18:39 9.0 53639 Female Web

Customer\_Login\_type Product\_Category Product Sales Quantity \

0 Member Auto & Accessories Car Media Players 140.0 1.0

1 Member Auto & Accessories Car Speakers 211.0 1.0

2 Member Auto & Accessories Car Body Covers 117.0 5.0

3 Member Auto & Accessories Car & Bike Care 118.0 1.0

4 Member Auto & Accessories Tyre 250.0 1.0

Discount Profit Shipping\_Cost Order\_Priority Payment\_method

0 0.3 46.0 4.6 Medium credit\_card

1 0.3 112.0 11.2 Medium credit\_card

2 0.1 31.2 3.1 Critical credit\_card

3 0.3 26.2 2.6 High credit\_card

4 0.3 160.0 16.0 Critical credit\_card

# Ouvrir la connexion à la base de données

conn = sqlite3.connect('/content/ecommerce.db')  # Assurez-vous de spécifier le bon chemin de la base de données

cursor = conn.cursor()

# Définir le schéma SQL basé sur les colonnes du DataFrame

columns = df.columns

schema = ", ".join([f"{col} TEXT" for col in columns])  # Toutes les colonnes en TEXT pour simplifier

# Création de la table

table\_name = "ecommerce\_data"

cursor.execute(f"CREATE TABLE IF NOT EXISTS {table\_name} ({schema});")

print(f"Table '{table\_name}' créée avec succès.")

# Insérer les données dans la table SQL

df.to\_sql(table\_name, conn, if\_exists="replace", index=False)

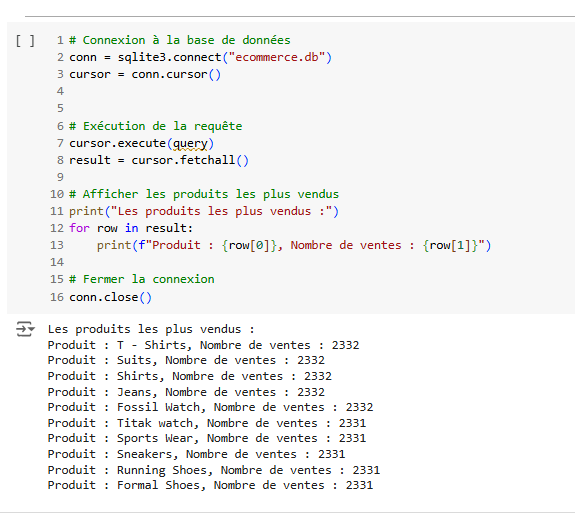
print("Données insérées dans la table SQL avec succès.")

# Fermer la connexion

conn.close()

**résultat :** Table 'ecommerce\_data' créée avec succès.

Données insérées dans la table SQL avec succès.

****

# Ouvrir la connexion à la base de données

conn = sqlite3.connect('/content/ecommerce.db')  # Assurez-vous de spécifier le bon chemin de la base de données

# Nom de la table

table\_name = 'ecommerce\_data'

# Requête pour obtenir les données de la catégorie

query = f"SELECT product\_category, COUNT(\*) AS count FROM {table\_name} GROUP BY product\_category;"

# Exécuter la requête et récupérer les données

Product\_Category\_data = pd.read\_sql\_query(query, conn)

# Vérifier les colonnes et les premières lignes

print(Product\_Category\_data.columns)  # Vérifie les noms des colonnes

print(Product\_Category\_data.head())   # Affiche les premières lignes du DataFrame

# Tracer un histogramme

plt.figure(figsize=(10, 6))

sns.barplot(data=Product\_Category\_data, x="Product\_Category", y="count", palette="viridis")  # Assurez-vous d'utiliser le bon nom de colonne

plt.title("Répartition des catégories de produits", fontsize=16)

plt.xlabel("Catégories", fontsize=14)

plt.ylabel("Nombre de produits", fontsize=14)

plt.xticks(rotation=45)

plt.tight\_layout()

plt.show()

# Fermer la connexion

conn.close()

Une image contenant texte, capture d’écran, diagramme, ligne

Description générée automatiquement

# Ouvrir la connexion à la base de données

conn = sqlite3.connect('/content/ecommerce.db')  # Assurez-vous de spécifier le bon chemin de la base de données

cursor = conn.cursor()

# Définir le schéma SQL basé sur les colonnes du DataFrame

columns = df.columns

schema = ", ".join([f"{col} TEXT" for col in columns])  # Toutes les colonnes en TEXT pour simplifier

# Création de la table

table\_name = "ecommerce\_data"

cursor.execute(f"CREATE TABLE IF NOT EXISTS {table\_name} ({schema});")

print(f"Table '{table\_name}' créée avec succès.")

# Insérer les données dans la table SQL

df.to\_sql(table\_name, conn, if\_exists="replace", index=False)

print("Données insérées dans la table SQL avec succès.")

# Fermer la connexion

conn.close()

**NoSQL**

# Installation de Redis et de la bibliothèque Python

!apt update

!apt install redis-server -y

!pip install redis

# Démarrage de Redis

import os

os.system("redis-server --daemonize yes")

print("Redis démarré")

import redis

import pandas as pd

# Connexion au serveur Redis

redis\_client = redis.Redis(host='localhost', port=6379, decode\_responses=True)

# Chargement des données

file\_path = "/content/E-commerce Dataset (1).csv"  # Mettez ici le chemin du fichier

data = pd.read\_csv(file\_path)

# Ajouter les données dans Redis (par exemple, chaque ligne comme un hash)

for idx, row in data.iterrows():

    key = f"order:{idx}"  # Nom unique pour chaque enregistrement

    redis\_client.hset(key, mapping=row.to\_dict())

print("Données chargées dans Redis avec succès.")

# Récupérer une commande spécifique

order\_key = "order:0"  # Exemple : la première commande

order\_data = redis\_client.hgetall(order\_key)

print(f"Données pour {order\_key}:")

print(order\_data)

Données pour order:0:

{'Order\_Date': '2018-01-02', 'Time': '10:56:33', 'Aging': '8.0', 'Customer\_Id': '37077', 'Gender': 'Female', 'Device\_Type': 'Web', 'Customer\_Login\_type': 'Member', 'Product\_Category': 'Auto & Accessories', 'Product': 'Car Media Players', 'Sales': '140.0', 'Quantity': '1.0', 'Discount': '0.3', 'Profit': '46.0', 'Shipping\_Cost': '4.6', 'Order\_Priority': 'Medium', 'Payment\_method': 'credit\_card'}

**Partie 3 : Comparaison des bases de données SQL et NoSQL**

**Analyse des bases relationnelles (SQL)**

**Contexte :** Utilisation de SQLite avec une base de données d'e-commerce pour analyser les ventes, les catégories de produits, les paiements, et d'autres paramètres liés aux commandes et aux clients.

1. **Gestion des relations et des dépendances entre les données :**
   * SQLite gère efficacement les relations entre entités à travers des tables normalisées. Par exemple, la table ecommerce\_data contient des colonnes comme Customer\_Id, Order\_Date, Sales, permettant de regrouper et relier des données grâce à des jointures (ex. Customer\_Id pour regrouper les ventes par client).
   * Les relations sont maintenues via des contraintes explicites (clé primaire, étrangère) garantissant l'intégrité des données.
2. **Performance pour des opérations complexes :**
   * SQLite est performant pour des requêtes complexes comme celles impliquant des jointures multiples, des agrégations (SUM, COUNT) ou des regroupements (GROUP BY), comme observé avec les analyses des ventes par mois ou les produits les plus populaires.
   * Toutefois, les performances peuvent diminuer avec des datasets massifs, en raison de sa scalabilité verticale limitée.
3. **Scalabilité verticale et ses limites :**
   * SQLite est conçu pour des applications mono-utilisateur ou des bases de taille modérée. Les limites deviennent évidentes lorsque la taille des données croît ou en cas de multiples accès simultanés.
4. **Intégrité des données et transactions ACID :**
   * SQLite respecte les principes ACID, garantissant que chaque transaction est atomique et que la cohérence des données est préservée, même en cas d'interruption ou de panne.

**Analyse des bases NoSQL**

**Contexte :** Utilisation de Redis pour gérer des données similaires dans un environnement NoSQL, avec un focus sur la rapidité et la flexibilité.

1. **Flexibilité du schéma :**
   * Redis permet de stocker chaque commande sous forme de hash, ce qui supprime la contrainte de schéma rigide. Cela facilite l’ajout ou la modification de colonnes sans nécessiter une migration complexe de la base.
2. **Capacité à traiter de grands volumes de données en parallèle :**
   * Redis utilise la scalabilité horizontale, permettant de gérer de grands volumes de données via un partitionnement efficace. Ce modèle est idéal pour les applications nécessitant des lectures/écritures rapides, comme les ventes en temps réel.
3. **Requêtes simples mais rapides :**
   * Redis excelle dans des cas d'usage où les requêtes sont simples, comme obtenir les catégories de produits ou les paiements les plus fréquents. Les opérations sont effectuées en mémoire, garantissant des performances élevées.
4. **Compromis avec la cohérence BASE :**
   * Contrairement à SQLite, Redis adopte un modèle BASE (Basically Available, Soft state, Eventual consistency). Cela peut entraîner une cohérence éventuelle, un compromis tolérable pour des systèmes nécessitant rapidité et disponibilité.

**Comparaison SQL / NoSQL et synthèse**

|  |  |  |
| --- | --- | --- |
| Critères | SQL (SQLite) | NoSQL (Redis) |
| Schéma | Rigide | Flexible |
| Relations | Très bien gérés (jointures, clés) | Non relationnel |
| Performance (requêtes complexes) | Optimale pour jointures et agrégations | Limites aux requêtes simples |
| Scalabilité | Verticale (limites de croissance) | Horizontale (parallélisme natif) |
| Cohérence des données | ACID (strict) | BASE (cohérence éventuelle) |
| Cas d’usage idéal | Analyses complexes, intégrité forte | Temps réel, grande volumétrie, flexibilité |

Les bases SQL conviennent mieux pour des analyses complexes et des applications nécessitant une stricte cohérence. Les bases NoSQL sont idéales pour des systèmes où la vitesse et la flexibilité sont primordiales.

**Partie 4 : Conclusion et recommandations**

Synthèse des avantages et inconvénients de chaque type de bases de données SQL et NoSQL.

|  |  |  |
| --- | --- | --- |
| Critères | Bases SQL(SQLite) | Base NoSQL(Redis) |
| Avantages | Gestion robuste des relations grâce aux jointures | Très rapide pour les lectures et écritures (opérations en mémoire ) |
|  | Forte cohérences données grâce aux transactions ACID | Flexible (pas de schéma rigide, adaptation facile aux changements) |
|  | Idéal pour les analyses complexes (ex. ventes par client, relations multitables) | Excellente scalabilité horizontale, capable de traiter de grandes quantités de données |
|  | Outils SQL familiers pour les utilisateurs (requêtes standardisées) | Configuration et maintenance simples pour des opérations courantes |
| Inconvénients | Scalabilité limitée à une machine (verticale uniquement) | Moins performant ou adapté pour des requêtes complexes nécessitant des jointures ou des agrégations avancées |
|  | Schéma rigide : toute modification nécessite une migration des données | Cohérence éventuelle (BASE), peut etre problématique pour des données critiques. |
|  | Performances en déclin avec des datasets très volumineux ou des multiples accès concurrents | Consommation mémoire importante (données stockées en RAM) |

Les bases SQL sont idéales pour l'analyse relationnelle de données structurées, tandis que les bases NoSQL conviennent mieux aux applications en temps réel nécessitant des performances rapides et une grande flexibilité.

1. **Quelle base de données est la plus adaptée au volume de données ?**  
   Redis (NoSQL) est mieux adapté pour gérer de grandes quantités de données grâce à sa scalabilité horizontale. Il permet de distribuer les données sur plusieurs nœuds, réduisant la charge sur un seul serveur. SQLite est limité par la capacité de l’hôte et convient davantage à des datasets modérés.
2. **Quelle base de données est la plus performante pour les requêtes complexes ?**  
   SQLite est plus performante pour des requêtes complexes nécessitant des jointures, des agrégations ou des analyses relationnelles approfondies. Redis est rapide pour les opérations simples, mais il n’est pas conçu pour des opérations complexes impliquant des relations entre données.
3. **Quelle base est plus flexible pour des besoins changeants ?**  
   Redis excelle dans la flexibilité. Son absence de schéma permet de modifier facilement la structure des données sans migration complexe. SQLite, avec son schéma rigide, nécessite une reconfiguration manuelle pour répondre aux changements.

**Recommandations**

* **Pour des analyses relationnelles complexes (exemple : analyse des ventes par client ou par catégorie avec jointures multiples)**, il est recommandé d'utiliser SQLite ou une autre base SQL relationnelle.
* **Pour des applications nécessitant une vitesse d’exécution élevée et une gestion flexible de données massives (exemple : suivi en temps réel des commandes ou gestion d’un système de recommandation)**, Redis ou une base NoSQL similaire est préférable.
* **Dans un environnement mixte**, une architecture hybride combinant SQL pour des données critiques et NoSQL pour des besoins de performance et de scalabilité pourrait être envisagée.

Diagramme comparatif

Une image contenant diagramme, ligne, cercle, texte

Description générée automatiquement