# Atelier 1

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

### Bases de données relationnelles (SQL)

1. Modèle de données structuré

- Les bases de données relationnelles organisent les données en (tables). Chaque table a des colonnes (qui décrivent le type d'information, comme "Nom", "Âge", "Adresse") et des lignes (ou enregistrements), qui contiennent les valeurs spécifiques de chaque colonne.

- Les tables peuvent être reliées entre elles grâce à des clés (ex. : une table "Clients" peut être liée à une table "Commandes" par l'identifiant du client).

2. Utilisation du langage SQL

- SQL (Structured Query Language) est un langage pour interroger et manipuler les données. Par exemple :

  - Sélectionner des données : `SELECT Nom FROM Clients WHERE Âge > 30`

  - Ajouter une nouvelle entrée : `INSERT INTO Clients (Nom, Âge) VALUES ('Alice', 25) `

- SQL est utilisé pour effectuer des recherches, mettre à jour et supprimer des informations dans la base de données.

3. Conformité ACID

- ACID est un ensemble de règles pour que les transactions (actions dans la base de données) soient fiables et sûres. Les quatre éléments d'ACID sont :

  - Atomicité : chaque transaction est "tout ou rien" (elle réussit entièrement ou échoue).

- Cohérence : les données restent dans un état valide après chaque transaction.

- Isolation : les transactions sont indépendantes les unes des autres.

- Durabilité : une fois une transaction validée, elle est sauvegardée de façon permanente.

4. Exemples de bases SQL

- Les bases de données relationnelles les plus courantes sont MySQL, PostgreSQL, et Oracle. Elles sont populaires pour des applications qui nécessitent une forte intégrité des données (ex. : gestion bancaire, gestion d'inventaire).

## Bases de données NoSQL

1. Les différents types de bases de données NoSQL

- Contrairement aux bases SQL, NoSQL offre plus de flexibilité pour des données moins structurées. Voici les principaux types :

- Clé-valeur : Stocke des paires clé-valeur, comme un dictionnaire. Exemple : Redis.

- Colonnes : Organise les données par colonnes plutôt que par lignes, optimisé pour les grandes quantités de données. Exemple : Cassandra.

- Documents : Stocke des documents (souvent au format JSON), chaque document peut avoir des champs différents. Exemple : MongoDB.

- Graphes : Conçu pour des données avec des relations complexes, comme les réseaux sociaux. Exemple : Neo4j.

2. Flexibilité du schéma et scalabilité horizontale

- Les bases NoSQL sont flexibles : elles n'imposent pas de structure fixe, ce qui est utile pour des données qui changent souvent.

Scalabilité horizontale : Elles peuvent être facilement étendues en ajoutant de nouveaux serveurs, ce qui est idéal pour des applications avec beaucoup de données (ex. : réseaux sociaux).

3. Conformité BASE

- Les bases NoSQL utilisent souvent les principes BASE, qui sont moins stricts que l'ACID. Cela signifie :

- Disponibilité de base (Basic Availability) : les données sont toujours disponibles, mais pas toujours exactes.

- État mou (Soft State) : les données peuvent changer en fonction des mises à jour.

- Cohérence éventuelle (Eventual Consistency) : les données finissent par être cohérentes, mais pas nécessairement immédiatement.

4. Exemples de bases NoSQL

- Parmi les bases NoSQL les plus populaires, on trouve MongoDB (bases de données de documents), Cassandra (colonnes), et Redis (clé-valeur). Elles sont souvent utilisées pour des applications qui nécessitent de la flexibilité et des mises à jour rapides (ex. : applications en temps réel).

## Partie 2 : Choix d'un cas d'usage

### Cas d'usage : Boutique en ligne

### Exigences métiers

En se basant sur les informations du dataset : Online Retail disponible sur Kaggle (utilisé dans mes projets SQL et NoSQL), voici une définition détaillée des exigences métier pour une application visant à gérer ces données.

1. Nature des données

- Type de données :

Les données du dataset sont structurées, avec des colonnes bien définies telles que :

- `InvoiceNo` : Identifiant unique des factures.
- `StockCode` : Code unique des produits.
- `Description` : Description des articles.
- `Quantity` : Quantité d'articles vendus.
- `InvoiceDate` : Date et heure de la transaction.
- `UnitPrice` : Prix unitaire des articles.
- `CustomerID` : Identifiant unique des clients.
- `Country` : Pays d'origine des clients.

Ces données sont parfaitement adaptées à un modèle relationnel (SQL) où chaque colonne correspond à un champ bien défini.

- Flexibilité du schéma :

Bien que le schéma soit principalement structuré, certaines colonnes (comme `Description`) pourraient varier en contenu ou format. Si l'on envisage d'ajouter des métadonnées sur les produits ou des informations supplémentaires non définies à l'avance (comme des avis ou des données promotionnelles), le besoin pourrait évoluer vers un modèle semi-structuré (NoSQL).

2. Volume des données

- Volume estimé :

Le dataset contient environ 406 829 lignes selon vos documents. Cela correspond à une application de gestion de données à moyenne échelle pour une boutique en ligne.

- Scénarios de croissance :

Si l’entreprise ajoute de nouveaux produits, marchés ou clients, les données pourraient atteindre une grande échelle, nécessitant une solution capable de scalabilité horizontale (comme un système NoSQL). Cependant, pour un usage local ou limité à un seul point de vente, le volume peut rester gérable avec un système relationnel comme SQLite.

3. Exigences en matière de performance

- Besoin de rapidité :

L’application doit permettre une consultation rapide des données pour des rapports courants comme :

- Identifier les ventes par pays ou par produit.

- Générer des rapports sur les ventes au cours d’une période donnée.

- Calculer les stocks restants et la valeur des ventes totales.

- Évolutivité :

Si le système s’étend à de multiples boutiques ou marchés, la capacité d’évoluer horizontalement devient critique. Une solution NoSQL comme TinyDB (ou MongoDB pour des cas plus complexes) permettrait d’étendre facilement les capacités de gestion des données.

- Transactions complexes :

Pour des transactions nécessitant une forte cohérence (comme la mise à jour simultanée des stocks ou le suivi précis des paiements), une base SQL est préférable, car elle garantit des propriétés ACID.

4. Modifications fréquentes des schémas

- Scénarios actuels :

Les données actuelles sont assez stables, avec un schéma fixe correspondant à des informations de vente standard. Un système SQL est donc parfaitement adapté.

- Scénarios futurs :

Si l'application doit inclure des fonctionnalités évolutives comme :

- Suivi des avis clients.

- Ajout de champs pour des promotions spécifiques.

- Intégration de données non structurées (comme des images de produits ou des documents d'expédition), le besoin de flexibilité de schéma pourrait justifier l'utilisation d'un système NoSQL.

Résumé des exigences métier**

| Critères | Exigences métiers | Recommandations |
|---|---|---|
| Nature de données | Structurées (actuellement), avec possibilité d'intégrer des données semi-structurées à l'avenir. | SQL (SQLite) pour le schéma actuel ; NoSQL si flexibilité future nécessaire. |
| Volume de données | Moyenne échelle (406 829 lignes), avec possibilité de croissance. | SQL pour les données actuelles ; NoSQL pour la scalabilité horizontale. |
| Exigences de performance | Rapports rapides, évolutivité pour plusieurs marchés, gestion de transactions cohérentes. | SQL pour la cohérence des transactions ; NoSQL pour des lectures/écritures rapides. |
| Modifications fréquentes | Rare actuellement, mais potentiellement fréquentes avec des données évolutives (avis clients, etc.). | SQL (stable actuellement) ou NoSQL (pour flexibilité future) |

Conclusion

Pour le cas actuel choisi, où les données sont principalement structurées et d'échelle moyenne, un système relationnel (SQL) comme SQLite est parfaitement adapté pour répondre aux besoins métiers. Toutefois, si l'entreprise prévoit une forte croissance ou l'intégration de données plus dynamiques et non structurées, une transition vers un système NoSQL pourrait être envisagée.

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

| Critères | SQL (SQLite) | NoSQL (TinyDB) |
|---|---|---|
| Type de modèle | Relationnel (basé sur des tables et des relations) | Orienté document (basé sur des documents JSON) |
| Gestion des relations et des dépendances | Gère bien les relations complexes entre tables grâce aux jointures. Chaque table a un schéma fixe. | Moins structuré, pas de relations complexes. Les données sont stockées dans des documents indépendants. |
| Flexibilité du schéma | Schéma rigide : il faut définir les relations et types de données à l'avance. | Schéma flexible : les documents peuvent varier d'une entrée à l'autre (absence de structure fixe). |
| Performance pour des opérations complexes | Excellente pour des requêtes complexes avec jointures et agrégations. Exemples : calculs des ventes par pays, jointures multiples entre tables. | Optimisé pour des requêtes simples mais moins performant pour les jointures complexes. |
| Capacité de scalabilité | Scalabilité verticale : augmentation des ressources sur un seul serveur, avec des limites de performance sur de gros volumes de données. | Scalabilité horizontale : peut être étendu sur plusieurs serveurs, adapté pour gérer de très grands volumes de données. |
| Transactions | Transactions ACID garantissant l'intégrité des données, notamment en cas de panne. Idéal pour des applications nécessitant des garanties fortes sur les données. | Cohérence BASE (Basically Available, Soft state, Eventual consistency) : moins stricte, permet une plus grande disponibilité et scalabilité, mais avec une certaine incohérence temporaire. |
| Exemples de cas d'usage | Gestion de données structurées : systèmes de comptabilité, applications nécessitant des rapports détaillés avec des relations complexes. | Applications avec des volumes de données massifs ou des besoins en disponibilité : systèmes de gestion de logs, réseaux sociaux, systèmes de recommandation, données non structurées. |
| Coût de maintenance | Relativement élevé en termes de gestion et maintenance pour de très | Relativement faible, surtout pour des petites et moyennes bases de |

| | | |
|---|---|---|
| | grandes bases (exigences en matière de ressources pour les opérations complexes). | données, mais peut nécessiter des ressources supplémentaires pour gérer des clusters répartis. |
| Exemple de requête | SELECT Country, COUNT(*) AS Total_Sales FROM retail GROUP BY Country ORDER BY Total_Sales DESC LIMIT 10; | query = Query(); high_price_items = db.search(query.UnitPrice > 10) |
| Intégrité des données | Assure une cohérence stricte des données grâce au modèle ACID, idéal pour des transactions critiques. | Moins strict en termes de cohérence immédiate, ce qui permet une plus grande tolérance aux pannes et une plus grande disponibilité des données. |

## Partie 4 : Conclusion et recommandations

Réponses aux questions clés

Quelle base de données est la plus adaptée au volume de données ?

SQL (SQLite) est suffisant pour le volume actuel de données (406 829 lignes) et peut encore gérer une croissance modérée.

NoSQL (TinyDB) est préférable si les données atteignent une grande échelle ou si l'on prévoit une croissance rapide et une gestion distribuée.

Quelle base de données est la plus performante pour les requêtes complexes ?

SQL (SQLite) est clairement plus performante pour les requêtes complexes impliquant des relations entre données, comme les agrégations sur les ventes par pays ou par période.

NoSQL (TinyDB) ne prend pas en charge les relations complexes, rendant les requêtes agrégées moins performantes.

Quelle base est plus flexible pour des besoins changeants ?

NoSQL (TinyDB) est plus flexible grâce à sa capacité à gérer des schémas dynamiques (par exemple, ajouter des avis clients ou des données promotionnelles sans modifier l'existant).

SQL (SQLite) est rigide et nécessite une refonte du schéma si les données évoluent significativement.

2. Recommandation finale

Recommandation : Une combinaison des deux (modèle polyglotte).

Justification :

Base SQL (SQLite) :

Utilisée pour gérer les données structurées actuelles (ventes, produits, clients) où la rigueur des transactions et les relations complexes entre données sont essentielles. Par exemple :

Calcul des ventes par pays.

Gestion des stocks en temps réel avec cohérence ACID.

Appropriée pour les analyses et rapports détaillés.

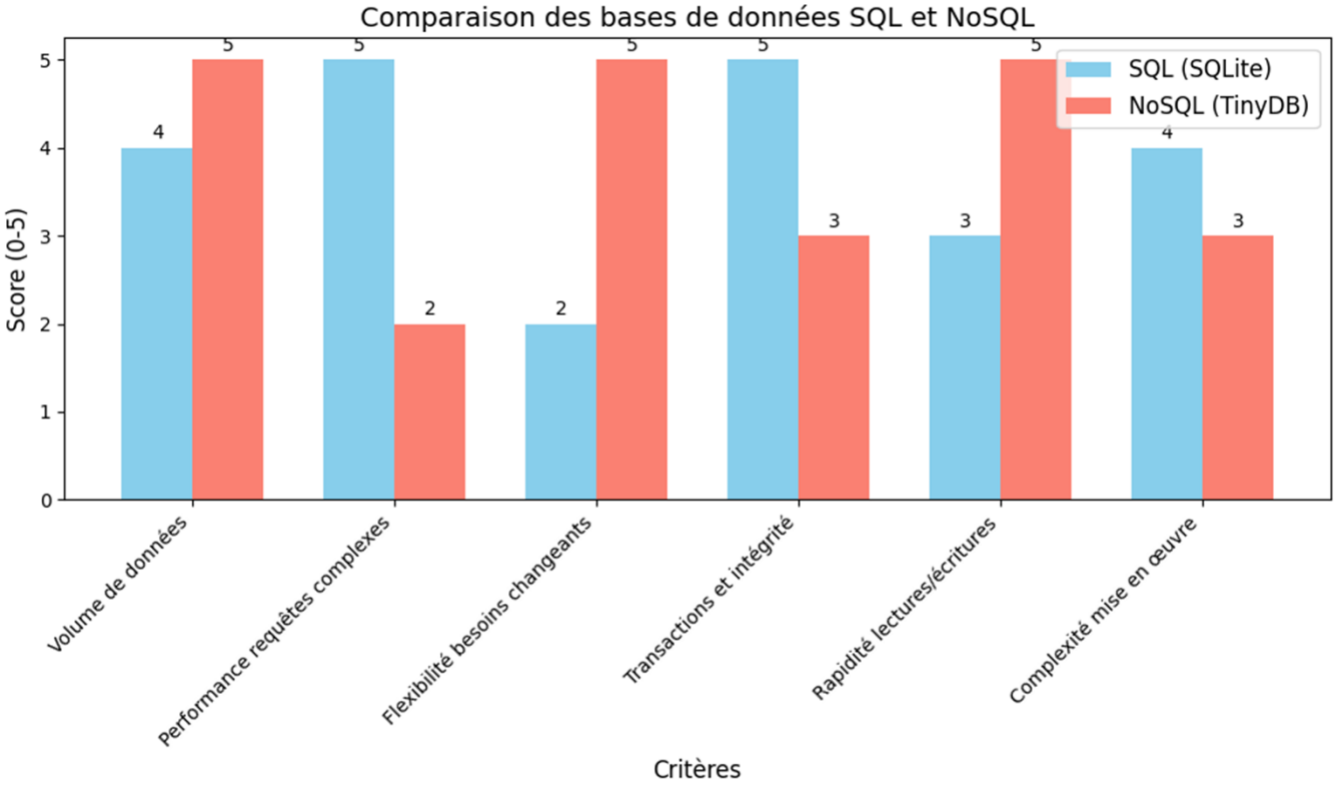
Base NoSQL (TinyDB) :

Utilisée pour des données semi-structurées ou évolutives, telles que :

Ajout de champs dynamiques (avis clients, données de navigation, métadonnées).

Gestion de larges volumes de logs ou données historiques (comme l'intégration future de sources tierces).

Utile dans des cas où la rapidité d'écriture ou de lecture est critique.


Comparaison des bases de données SQL et NoSQL
SQL (SQLite)
NoSQL (TinyDB)
Score (0-5)
Critères
Volume de données
Performance requêtes complexes
Flexibilité besoins changeants
Transactions et intégrité
Rapidité lectures/écritures
Complexité mise en œuvre
0
1
2
3
4
5
4
5
5
2
2
5
5
3
3
5
4
3