L'informatique des entrepôts de données

Daniel Lemire

SEMAINE 14
NoSQL

14.1. Présentation de la semaine

On construit souvent les entrepôts de données en utilisant des systèmes de bases de données relationnels conventionnelles. Il existe cependant de nombreuses alternatives. On désigne souvent l'ensemble de ces alternatives par le terme NoSQL [3]. Cette semaine, nous ferons un survol de ces alternatives.

14.2. Une grande diversité

Dans une certaine mesure, les systèmes tels que MySQL, VoltDB, SQL Server ou Oracle sont interchangeables. Certains systèmes sont plus puissants et permettent plus aisément de traiter des bases de données volumineuses. D'autres, comme MySQL, sont peu coûteux. Mais ils

reposent tous sur les mêmes principes mis de l'avant par Codd [2] dans les années 1970. Tout d'abord, ils permettent de construire des bases de données relationnelles. Les données prennent donc nécessairement la forme de rangées dans des colonnes. De plus, elles tentent le plus possible de satisfaire aux contraintes ACID qui permettent d'assurer l'intégrité des données. Les propriétés ACID (Atomicité, Cohérence, Isolation, Durabilité), essentielles aux systèmes de bases de données relationnelles, garantissent la fiabilité des transactions. L'**Atomicité** assure qu'une transaction est exécutée entièrement ou pas du tout; la **Cohérence** maintient l'intégrité des données en respectant les contraintes définies; l'**Isolation** garantit que les transactions s'exécutent indépendamment les unes des autres; et la **Durabilité** assure que les modifications validées sont définitivement enregistrées, même en cas de panne. Ces propriétés, inspirées des travaux de Codd, sont fondamentales pour les applications critiques, comme les systèmes bancaires ou de gestion d'inventaire.

- L'**atomicité** garantit qu'une transaction est exécutée en totalité ou pas du tout, évitant les états intermédiaires incohérents. Si une partie de la transaction échoue, toutes les modifications sont annulées (rollback).
  Exemple : Un client transfère 500$ de son compte A (solde initial : 1000$) vers son compte B (solde initial : 200$). La transaction comprend deux étapes : débiter 500$ du compte A et créditer 500$ sur le compte B. Si le système plante après le débit mais avant le crédit, l'atomicité annule le débit, restaurant le solde initial de A (1000$) et B (200$), évitant une perte d'argent.
- La **cohérence** assure que chaque transaction amène la base de données d'un état valide à un autre, en respectant toutes les contraintes, règles et intégrités définies (comme les clés primaires ou les soldes non négatifs).
  Exemple : Dans le même transfert de 500$, une contrainte stipule que le solde d'un compte ne peut pas être négatif. Si le compte A n'a que 400$, la transaction est rejetée, car débiter 500$ violerait la contrainte (solde de -100$). La cohérence empêche l'exécution de la transaction, maintenant la validité des données.
- L'**isolation** garantit que les transactions s'exécutent indépendamment les unes des autres, même si elles sont effectuées simultanément. Une transaction partiellement exécutée n'est pas visible pour les autres jusqu'à ce qu'elle soit terminée.
  Exemple : Deux transactions simultanées sont lancées : la

première transfère 500$ de A à B, et la seconde consulte le solde de A. Sans isolation, la seconde transaction pourrait voir un solde intermédiaire de A (500$ après débit, avant crédit sur B). L'isolation utilise des verrous pour s'assurer que la seconde transaction voit soit le solde initial (1000$), soit le solde final (500$), mais jamais un état incohérent.

- La **durabilité** garantit que, une fois une transaction validée (commit), ses modifications sont enregistrées de manière permanente, même en cas de panne du système.
  Exemple : Après le transfert de 500$ de A à B, la transaction est validée, mettant à jour les soldes (A : 500$, B : 700$) dans la base de données. Si le serveur tombe en panne immédiatement après, la durabilité assure que ces nouveaux soldes sont sauvegardés (par exemple, sur un disque non volatil) et seront disponibles au redémarrage, évitant toute perte de données.

En pratique, il n'est pas toujours nécessaire ou souhaitable de stocker les données en utilisant un modèle relationnel. Dans certains cas, par exemple, un modèle en graphe peut être plus approprié. Et les contraintes ACID ne sont pas toujours nécessaires. Par exemple, il n'est pas toujours essentiel, que lorsque de nouvelles informations ont été saisies dans le système, toutes les requêtes subséquentes en tiennent compte. Dans la vie courante, les systèmes informatiques sont plutôt éventuellement cohérents plutôt que strictement cohérents. Lors d'un achat avec une carte de crédit, par exemple, il peut arriver que vous soyez, temporairement, facturé deux fois, ou pas du tout. L'important est que le système puisse corriger l'anomalie suffisamment rapidement. En somme, on peut donc utiliser d'autres types de systèmes. Il en existe une grande variété, et le bon choix dépend de vos données et de l'utilisation que vous voulez en faire.

Les bases de données NoSQL (*Not Only SQL*) sont des systèmes de gestion de données conçus pour répondre aux besoins des applications modernes nécessitant une grande scalabilité, flexibilité et performance avec des volumes massifs de données non structurées ou semi-structurées. Contrairement aux bases de données relationnelles traditionnelles basées sur des tables et le langage SQL, les bases NoSQL adoptent divers modèles comme les bases clé-valeur (ex. : Redis), document (ex. : MongoDB), colonne (ex. : Cassandra) ou graphe (ex. : Neo4j), permettant une manipulation aisée de données hétérogènes. Elles privilégient souvent la disponibilité et la partition (conformément au théorème CAP) sur la cohérence stricte, ce qui les rend idéales pour des applications comme les réseaux sociaux, l'analyse en temps réel ou l'Internet des

objets. Cependant, elles peuvent sacrifier certaines garanties ACID au profit de performances élevées et d'une architecture distribuée.

14.3. Bases de données orientées document

Dans une base de données orientées document, on remplace les rangées de la base de données relationnelle par un document. Un document n'est qu'un objet ayant un nombre variable d'attributs. On peut penser par exemple à un système stockant des courriels. Chaque courriel peut avoir plusieurs attributs, dont le destinataire ou l'objet. Certains attributs peuvent n'apparaître que dans certains documents.

On représente souvent les documents sous forme de *JSON* (JavaScript Object Notation). JSON (*JavaScript Object Notation*) est un format de données léger et textuel utilisé pour structurer, stocker et échanger des informations entre systèmes, notamment dans les applications web et les API. Basé sur une syntaxe simple inspirée des objets JavaScript, il représente les données sous forme de paires clé-valeur (objets) et de listes ordonnées (tableaux), supportant des types comme les chaînes, nombres, booléens, *null*, et des structures imbriquées. Sa lisibilité humaine, sa compatibilité multi-langages (Python, Java, etc.), et sa faible surcharge en font un standard universel, remplaçant souvent XML. Par exemple, un utilisateur peut être représenté comme `{"name": "Alice", "age": 30}`. JSON est largement adopté pour les configurations, les bases de données orientées document, et les communications client-serveur, grâce à sa simplicité et sa flexibilité.

Les bases orientées documents stockent les données comme des collections de documents, où chaque document est une unité autonome contenant des paires clé-valeur (similaires à des objets JSON). Cette approche permet :

- **Flexibilité** : Les documents d'une même collection peuvent avoir des structures différentes, sans schéma prédéfini.
- **Scalabilité** : Conçues pour des architectures distribuées, elles supportent le partitionnement et la réplication.
- **Requêtes riches** : Elles permettent des interrogations sur les champs des documents, souvent via des API ou des langages spécifiques.

Ces bases sont particulièrement adaptées aux applications nécessitant une évolutivité horizontale et une gestion de données variées, comme les réseaux sociaux ou l'Internet des objets.

Apache CouchDB, lancé en 2005, est une base de données orientée documents open-source qui privilégie la disponibilité et la tolérance au

partitionnement (selon le théorème CAP). Elle utilise JSON pour les documents, HTTP pour l'API RESTful, et MapReduce pour les vues (requêtes).

- **Réplication bidirectionnelle** : CouchDB excelle dans la synchronisation entre bases, même hors ligne, grâce à son protocole de réplication, ce qui le rend idéal pour les applications mobiles.
- **Vues MapReduce** : Les requêtes sont définies via des fonctions JavaScript MapReduce, stockées comme vues indexées.
- **ACID par document** : Chaque document est manipulé avec des garanties ACID, mais la cohérence globale est éventuelle (modèle "éventual consistency").
- **API RESTful** : Toutes les opérations (CRUD) sont accessibles via des requêtes HTTP standard.

Un document dans CouchDB pour un utilisateur :

```
{
    "_id": "user123",
    "name": "Alice Dupont",
    "email": "alice@example.com",
    "roles": ["admin", "editor"]
}
```

Pour récupérer ce document via HTTP :

```
GET /mydb/user123 HTTP/1.1
Host: localhost:5984
```

MongoDB, lancé en 2009, est une base de données orientée documents open-source qui met l'accent sur les performances, la scalabilité et la facilité d'utilisation. Elle stocke les documents au format BSON (JSON binaire) et utilise un langage de requêtes riche et intuitif.

- **Scalabilité horizontale** : MongoDB supporte le sharding (partitionnement des données) et la réplication pour gérer de grands volumes de données.
- **Requêtes flexibles** : Les requêtes utilisent une syntaxe proche de JavaScript, avec des opérateurs puissants pour filtrer, agréger et trier.
- **Cohérence forte** : Par défaut, MongoDB offre une cohérence forte pour les lectures et écritures sur un nœud primaire, mais permet des lectures sur des répliques secondaires avec cohérence éventuelle.

- **Agrégation** : Un pipeline d'agrégation permet des transformations complexes, comme des jointures ou des calculs statistiques.

Exemple: Un document dans MongoDB pour un produit :

```
{
    "_id": "prod456",
    "name": "Smartphone XYZ",
    "price": 599.99,
    "categories": ["electronics", "mobile"]
}
```

Pour trouver tous les produits de la catégorie "electronics" :

```
db.products.find({ "categories": "electronics" })
```

- **Modèle de cohérence** : CouchDB privilégie la cohérence éventuelle pour la synchronisation, tandis que MongoDB offre une cohérence forte par défaut.
- **Requêtes** : MongoDB propose un langage de requêtes plus riche et intuitif, tandis que CouchDB repose sur MapReduce, moins flexible.
- **Synchronisation** : CouchDB excelle dans la réplication bidirectionnelle pour les scénarios hors ligne, contrairement à MongoDB, plus orienté vers les performances en ligne.
- **Cas d'usage** : CouchDB est idéal pour les applications décentralisées (ex. : apps mobiles avec synchronisation), tandis que MongoDB convient aux applications web à fort trafic ou aux analyses en temps réel.

Les utilisateurs d'AWS (Amazon Web Services) voudront sans doute prendre en considération DocumentDB qui offre une interface compatible avec MongoDB. Elle permet de stocker, de gérer et de requêter des données semi-structurées, telles que les documents JSON, avec une faible latence.

14.4. Bases de données en graphe

Dans les bases de données en graphe, on remplace l'unité de base des bases de données relationnelles (le tuple ou rangée) par le nœud dans un graphe. Si vos données correspondent, par exemple, à un réseau social ou à un ensemble de pages web, une base de données en graphe peut être mieux adaptée à vos besoins.

Un graphe est une structure de données mathématique composée de **nœuds** (ou sommets) reliés par des **arêtes** (ou liens), utilisée pour modéliser des relations entre entités dans des domaines comme les réseaux sociaux, les réseaux de transport ou l'informatique. Les graphes peuvent être **orientés** (arêtes avec une direction, comme un sens unique) ou **non orientés** (arêtes bidirectionnelles), **pondérés** (arêtes avec un poids, comme une distance) ou **non pondérés**. Les concepts clés incluent le **degré** d'un nœud (nombre d'arêtes connectées), les **chemins** (séquences de nœuds reliés), les **cycles** (chemins revenant au point de départ), et la **connexité** (capacité à atteindre tous les nœuds depuis un point). Les graphes sont fondamentaux pour résoudre des problèmes comme la recherche du plus court chemin (algorithme de Dijkstra) ou l'analyse de réseaux complexes, et sont souvent implémentés via des matrices d'adjacence ou des listes d'adjacence.

Neo4j est une base de données orientée graphe, conçue pour stocker, gérer et interroger des données sous forme de nœuds (entités) et d'arêtes (relations), ce qui la rend idéale pour modéliser des réseaux complexes comme les réseaux sociaux, les systèmes de recommandation ou les analyses de fraude. Contrairement aux bases relationnelles ou documentaires, Neo4j excelle dans le traitement des relations, permettant des requêtes rapides sur des structures interconnectées grâce à son moteur optimisé pour les traversées de graphes. Elle utilise le langage Cypher pour les requêtes, qui permet de décrire des motifs de graphes de manière intuitive. Par exemple, dans un réseau social, Neo4j peut rapidement identifier les amis d'amis ou les chemins entre utilisateurs. Sa scalabilité, son support pour les propriétés ACID et son intégration avec des outils d'analyse en font un choix puissant pour les applications nécessitant une compréhension approfondie des relations.

14.5. Bases de données orientées objet

Les bases de données orientées objet ne sont pas une invention récente, elles sont nées en même temps que l'adoption des langages de programmation orienté-objet. En somme, l'idée est relativement simple : il s'agit de stocker non pas des relations, mais des objets (au sein de la programmation orientée-objet).

Les bases de données orientées objet (BDOO) combinent les principes de la programmation orientée objet, comme l'encapsulation, l'héritage et le polymorphisme, avec la gestion de données, permettant de stocker des objets complexes directement sans les convertir en structures relationnelles. Elles sont particulièrement adaptées aux applications nécessitant des modèles de données riches, comme les systèmes CAD, les jeux

vidéo ou la bioinformatique. Ces dernières années, les développements récents dans les BDOO, notamment avec des systèmes comme **ObjectBox** et **Realm** (désormais intégré à MongoDB), mettent l'accent sur l'optimisation pour les environnements mobiles et IoT, offrant une synchronisation efficace, une faible empreinte mémoire et des performances élevées sur des appareils à ressources limitées. De plus, l'intégration avec les bases NoSQL hybrides et les avancées dans la sérialisation des objets (par exemple, via JSON) renforcent leur adoption dans les architectures *cloud-native*, tout en maintenant la flexibilité des schémas dynamiques et la compatibilité avec les paradigmes modernes de développement.

14.6. Bases de données de type clé-valeur

Il arrive parfois que l'on n'ait besoin que de retrouver des valeurs étant donné une clé. Par exemple, si votre application a besoin de retrouver le dossier d'un étudiant en n'utilisant que son code étudiant, alors une base de données de type clé-valeur sera idéale. Elles tendent à être plus rapides et plus simples que les bases de données relationnelles. On les utilise, notamment, pour stocker des pages web précalculées afin d'accélérer certains sites web. Dans un tel cas, la clé peut être, tout simplement, l'hyperlien de la page.

Les bases de données clé-valeur sont des systèmes NoSQL qui stockent des données sous forme de paires clé-valeur, où chaque clé unique mappe directement une valeur, offrant une simplicité et une performance élevées pour des accès rapides. Elles sont idéales pour des applications nécessitant une faible latence, comme le caching, la gestion de sessions ou le stockage de données non structurées. Contrairement aux bases relationnelles, elles n'imposent pas de schéma rigide, ce qui les rend flexibles mais moins adaptées aux requêtes complexes. Tokyo Cabinet, Membase (aujourd'hui Couchbase) et Redis sont des exemples emblématiques, chacun avec des caractéristiques distinctes adaptées à des cas d'usage spécifiques.

Une base de données clé-valeur stocke des données comme un dictionnaire, où une clé (généralement une chaîne) est associée à une valeur (chaîne, nombre, objet JSON, etc.). Leur simplicité permet des opérations rapides (lecture/écriture) avec une complexité proche de $\mathcal{O}(1)$. Les principales caractéristiques incluent :

- **Performance** : Accès direct par clé, minimisant la latence.
- **Scalabilité** : Support du partitionnement et de la réplication pour gérer de grands volumes de données.
- **Flexibilité** : Absence de schéma, permettant de stocker des données hétérogènes.

- **Limitations** : Moins adaptées aux requêtes relationnelles ou analytiques complexes.

Elles sont utilisées pour le caching (ex. : Redis), la gestion de sessions, ou le stockage de données temporaires.

Tokyo Cabinet est une bibliothèque de base de données clé-valeur intégrée (embedded), écrite en C, développée par Mikio Hirabayashi et sponsorisée par Mixi. Conçue pour la vitesse et l'efficacité, elle supporte plusieurs types de stockage, comme les tables de hachage, les arbres B+ et les tables fixes.

- **Types de stockage** : Hash (rapide, accès direct), B+Tree (support des requêtes par plage), Fixed-length (optimisé pour les clés numériques) et Table (émulation de colonnes).
- **Compression** : Support natif pour les algorithmes Lempel-Ziv ou BWT, réduisant la taille des données jusqu'à 25%.
- **ACID** : Transactions avec propriétés ACID via journalisation (write-ahead logging) pour le mode Table.
- **Concurrence** : Accès concurrent avec verrouillage optimiste.

Membase, désormais intégré à Couchbase, est une base de données clé-valeur et orientée documents, conçue pour la haute disponibilité et la scalabilité. Initialement développée pour répondre aux besoins des applications web à fort trafic, elle combine la simplicité des accès clé-valeur avec des fonctionnalités de documents JSON.

- **Cache et persistance** : Stockage en RAM pour des accès rapides, avec persistance sur disque pour la durabilité.
- **Scalabilité distribuée** : Supporte le sharding et la réplication multi-nœuds pour gérer des charges élevées.
- **Compatibilité Memcached** : Protocole compatible, facilitant l'intégration avec les systèmes existants.
- **Requêtes avancées** : Évolution vers Couchbase, qui ajoute des requêtes N1QL (similaires à SQL) pour les documents.

Exemple en Python avec Couchbase pour stocker un utilisateur :

```
from couchbase.cluster import Cluster, ClusterOptions
from couchbase.auth import PasswordAuthenticator

cluster = Cluster('couchbase://localhost',
    ClusterOptions(
     PasswordAuthenticator('username', 'password')))
bucket = cluster.bucket('mybucket')
collection = bucket.default_collection()
collection.upsert('user:1', {'name': 'Alice', 'email':
    'alice@example.com'})
```

Redis (Remote Dictionary Server), créé par Salvatore Sanfilippo en 2009, est une base de données clé-valeur en mémoire, open-source, connue pour sa rapidité et sa polyvalence. Utilisée principalement comme cache, elle supporte des structures de données complexes et est la base NoSQL la plus populaire.

- **Stockage en mémoire** : Toutes les données sont en RAM, offrant une latence inférieure à la milliseconde, mais limitées par la capacité mémoire.
- **Structures de données** : Supporte les chaînes, listes, ensembles, hachages, bitmaps, et streams, permettant des cas d'usage avancés comme les files d'attente ou les classements.
- **Persistance optionnelle** : Snapshots (RDB) ou journalisation (AOF) pour la durabilité, mais avec un risque de perte de données entre snapshots.
- **Haute disponibilité** : Redis Sentinel et clustering pour la réplication et le basculement.

Exemple en Python pour stocker et récupérer des données avec Redis :

```
import redis
r = redis.Redis(host='localhost', port=6379, db=0)
r.set('user:1', 'Alice')
print(r.get('user:1'))  # Affiche: b'Alice'
```

Valkey est un fork open-source de Redis, lancé en mars 2024 sous la licence BSD-3 par la Linux Foundation, en réponse au passage de Redis à des licences non open-source (RSALv2/SSPLv1). Soutenu par d'anciens mainteneurs de Redis (comme Madelyn Olson) et des géants technologiques (AWS, Google, Oracle, Ericsson), Valkey vise à être un remplacement direct de Redis 7.2.4, avec une compatibilité totale des clients et protocoles. Il a atteint la disponibilité générale (GA) en 2024 et planifie des améliorations comme le support de la recherche vectorielle et des performances multi-threadées. Valkey adopte une gouvernance communautaire ouverte, évitant les changements de licence imprévus, et gagne rapidement en popularité comme alternative fiable.

- **Compatibilité Redis** : Conçu comme un remplacement direct, supportant les clients Redis et les commandes sans modification.
- **Gouvernance ouverte** : Géré par un comité technique incluant des contributeurs de Redis, sous l'égide de la Linux Foundation.

- **Performance** : Améliorations prévues, comme une réduction de la surcharge par paire clé-valeur et un multithreading optimisé.
- **Licence BSD-3** : garantit un accès libre et commercial sans restrictions, contrairement aux licences initiales post-2024 de Redis.

- **Tokyo Cabinet** : Intégré, rapide, et efficace pour les applications embarquées. Moins adapté aux systèmes distribués à grande échelle, mais excellent pour des bases compactes avec compression.
- **Membase/Couchbase** : Combine clé-valeur et documents, idéal pour les applications web à fort trafic avec besoins de persistance et de requêtes complexes.
- **Redis** : Leader historique pour le caching et les applications en temps réel grâce à sa vitesse en mémoire et ses structures de données riches, mais les changements de licence (RSALv2/SSPLv1, puis AGPLv3) ont poussé certains utilisateurs vers des forks comme Valkey.
- **Valkey** : Fork open source de Redis, offrant une compatibilité totale et une gouvernance communautaire. Idéal pour ceux qui recherchent une alternative fiable sous licence BSD-3, avec des améliorations techniques en cours.
- **Autres forks** : Redict (sous LGPL-3.0, axé sur la simplicité et la maintenance à long terme) et KeyDB (fork préexistant sous BSD-3, axé sur la performance avec multithreading et réplication active) sont également des alternatives, mais moins adoptées que Valkey.

Les bases de données clé-valeur comme Tokyo Cabinet, Membase (Couchbase), Redis et Valkey offrent des solutions performantes pour un accès rapide et une gestion flexible des données. Tokyo Cabinet excelle dans les environnements embarqués, Couchbase convient aux applications distribuées, Redis reste une référence pour le caching malgré les controverses sur ses licences, et Valkey émerge comme une alternative open source prometteuse, soutenue par une communauté forte et des acteurs majeurs. Les récents forks comme Valkey et Redict reflètent la résilience de la communauté open-source face aux changements de licence, garantissant des solutions fiables pour les développeurs. Le choix dépend des besoins : compacité pour Tokyo Cabinet, scalabilité pour Couchbase, performance historique pour Redis, ou stabilité open source pour Valkey.

14.7. Bases de données orientées colonne

Pour les très grandes tables devant être distribuées sur un réseau de machine, le modèle proposé par Google par sa base de données BigTable [1] a été adopté par plusieurs autres sociétés. Au lieu d'être orienté par rangées, comme la plupart des bases de données relationnelles, BigTable est orienté colonne, ce qui signifie que les données d'une même colonne sont stockées de manière consécutive. Par ailleurs, dans ce modèle, la compression des données joue un rôle important afin d'accélérer le traitement.

HBase et BigTable sont des bases de données NoSQL distribuées, orientées colonnes, conçues pour gérer de grands volumes de données avec une haute scalabilité et une faible latence. **BigTable**, développé par Google en 2004, est une base propriétaire optimisée pour les charges de travail massives, comme l'indexation web ou Google Maps, utilisant une structure de colonnes *sparse* et un stockage hiérarchique avec des tablettes (*tablets*) pour la répartition des données. **HBase**, un projet open-source Apache inspiré de BigTable, fonctionne sur HDFS (Hadoop) et offre une interface similaire, avec un accès aléatoire en temps réel aux données et une intégration étroite avec l'écosystème Hadoop. Les deux systèmes privilégient la cohérence et la disponibilité (selon le théorème CAP), supportent des milliards de lignes et des colonnes dynamiques, et sont adaptés aux applications comme l'analyse de *logs*, la gestion de séries temporelles ou les systèmes de recommandation. Cependant, HBase est plus accessible pour les déploiements personnalisés, tandis que BigTable est limité à l'environnement Google Cloud.

Le système le plus populaire de ce type est sans doute DuckDB [4]. DuckDB est un système de gestion de base de données SQL analytique, orienté colonne, qui fonctionne en mode intégré (embedded). Cela signifie qu'il peut être exécuté directement dans l'application qui l'utilise, sans nécessiter un serveur séparé. DuckDB est conçu pour offrir des performances élevées sur des requêtes analytiques complexes avec de grandes quantités de données, tout en conservant une empreinte mémoire faible. Il est particulièrement adapté pour les environnements où l'analyse de données doit être effectuée rapidement et efficacement, comme dans les outils de science des données, les notebooks Jupyter, ou encore dans des scripts Python ou R. Il supporte une grande partie du standard SQL, ce qui le rend accessible et facile à utiliser pour ceux qui ont déjà une expérience avec d'autres bases de données SQL.

À consulter : `https://duckdb.org`

14.8. Conclusion

Il est impossible de rendre justice à l'ensemble des alternatives aux systèmes de bases de données traditionnelles. C'est un domaine en constante mutation. Par contre, il faut retenir que les bases de données traditionnelles ne sont pas la seule possibilité au sein d'un entrepôt de données.

Votre avis compte !

Chers étudiants,
Si vous repérez une erreur dans ce document ou si vous avez une suggestion pour l'améliorer, nous vous invitons à remplir notre **formulaire anonyme**.
Votre contribution est précieuse pour nous !

Cliquez ici pour accéder au formulaire

BIBLIOGRAPHIE

1. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.
2. E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.
3. W. Khan, T. Kumar, C. Zhang, K. Raj, A. M. Roy, and B. Luo. Sql and nosql database software architecture performance analysis and assessments—a systematic literature review. *Big Data and Cognitive Computing*, 7(2):97, 2023.
4. M. Raasveldt and H. Mühleisen. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1981–1984, 2019.