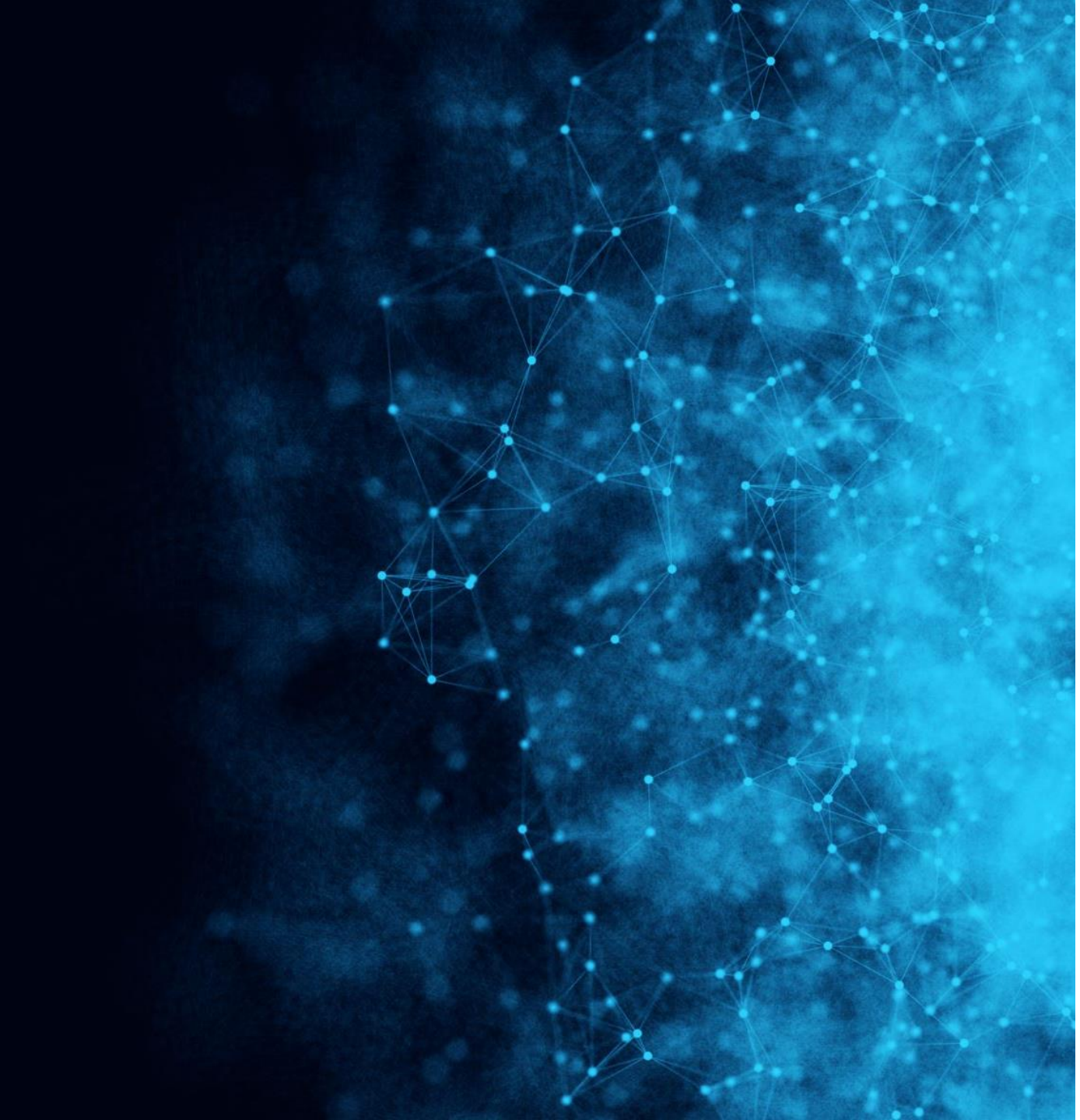


MongoDB

Ali Koudri – ali.koudri@gmail.com

Plan

- Introduction au NoSQL
- Présentation générale de MongoDB
- Concepts de MongoDB
- Installation de MongoDB
- Shell MongoDB
- Cycle de vie de la donnée (CRUD)
- Agrégations
- Transactions
- Schémas de données
- Données géospatiales
- Index
- Performance
- Sharding
- APIs
- Conclusion

Prérequis

- Afin de suivre les travaux pratiques, veuillez clôner le dépôt git suivant:
  - https://github.com/akoudri/mongo-training.git
- Vous aurez également besoin de
  - docker et docker-compose
  - java 17
  - Maven 3.8.X

NoSQL

Définition

- Le terme NoSQL désigne une famille de systèmes de gestion de bases de données ne reposant pas sur le modèle relationnel traditionnel.

- Le nom "NoSQL" signifie littéralement "Not Only SQL", soulignant que ces bases peuvent offrir des modes de fonctionnement alternatifs ou complémentaires au SQL classique.

Motivations

- **Scalabilité horizontale** : Les bases relationnelles traditionnelles peinent à gérer la croissance du volume de données distribués sur plusieurs serveurs.
- **Flexibilité du schéma** : Les modèles NoSQL permettent des structures de données dynamiques s'adaptant facilement à des données hétérogènes.
- **Performance et rapidité** : Optimisées pour certaines opérations spécifiques (lecture/écriture rapide, gestion de grands volumes, etc.), ces bases répondent aux besoins d'applications modernes en temps réel.
- **Adaptation au Big Data et au Cloud** : Les nouvelles architectures applicatives, avec des besoins de traitements massifs et géographiquement répartis, nécessitent un paradigme plus souple que le SQL traditionnel.

Histoire

- **Années 1970-2000** : Les bases de données relationnelles dominent le marché, avec des standards comme SQL et ACID.
- **Fin des années 2000** : Émergence de grandes entreprises Internet (Amazon, Google, Facebook) qui rencontrent les limites du modèle relationnel pour gérer d'immenses volumes de données.
- **2009** : Popularisation du terme "NoSQL" lors d'une conférence destinée à regrouper ces nouveaux systèmes.
- **Aujourd'hui** : NoSQL regroupe une grande diversité de solutions adaptées à des besoins variés et coexiste souvent avec les bases relationnelles classiques dans les architectures hybrides.

Principes

- **Schéma flexible** : Absence de schéma fixe, tolérance des variations entre différents enregistrements.
- **Haute disponibilité et tolérance aux pannes** : Les systèmes sont conçus pour maintenir l'accès aux données malgré des pannes ou déconnexions.
- **Répartition et distribution** : Données stockées sur plusieurs machines (sharding, réplication).
- **Modèles variés de données** : Orientés clé/valeur, document, graphe ou colonne plutôt que le tableau relationnel unique.

Cas d'usage

- **Applications web à grande échelle** : Réseaux sociaux, plateformes e-commerce, jeux en ligne massivement multi-joueurs.
- **Stockage de logs et analyse de flux de données** : Ingestion et agrégation rapide d’évènements massifs.
- **Gestion de catalogues** produits ou profils utilisateurs avec attributs très variés.
- Systèmes nécessitant une **haute disponibilité**, avec **réplication géographique**.

Exemples

- **MongoDB** : Stockage et consultation de documents semi-structurés pour des sites web dynamiques.

- **Cassandra** : Gestion de bases de données distribuées pour la messagerie instantanée ou la collecte de logs.

- **Redis** : Mise en cache et traitement de sessions utilisateurs en temps réel.

- **Neo4j** : Analyse de graphes sociaux ou recommandations personnalisées.

Bases clé-valeur

- **Principe** : Les données sont stockées sous forme de paires clé/valeur. La clé est unique et permet d'accéder rapidement à la valeur correspondante, qui peut prendre n'importe quelle forme (texte, objet sérialisé, etc.).
- **Avantages** :
  - Modèle très simple et rapide pour les accès directs (lookup).
  - Scalabilité et performances excellentes, particulièrement adaptée au cache et gestion de sessions.
  - Flexible : pas de schéma imposé.
- **Inconvénients** :
  - Difficulté à effectuer des requêtes complexes ou multi-attributs.
  - Peu adapté à la gestion de relations entre données.
  - Fonctionnalités limitées pour les analyses avancées.
- **Exemples** : Redis, DynamoDB.

Bases orientées document

- **Principe** : Les données sont stockées sous forme de documents (souvent JSON, BSON ou XML) pouvant avoir une structure flexible et hétérogène. Chaque document est adressé par une clé unique et peut être indexé sur plusieurs champs.
- **Avantages** :
  - Schéma flexible : facile à adapter aux évolutions du modèle de données.
  - Indexation puissante, requêtes riches possibles sur le contenu des documents.
  - Très bon compromis pour des applications web, gestion de profils, catalogues, etc..
- **Inconvénients** :
  - Moins efficace pour les requêtes transactionnelles complexes ou inter-documents.
  - Risque de duplication des données et nécessité de bien concevoir les modèles pour limiter la redondance.
- **Exemples** : MongoDB, CouchDB.

NoSQL – Bases orientées colonnes

- **Principe** : Les données sont organisées par colonnes et non par lignes. Chaque colonne est stockée séparément, ce qui optimise les opérations analytiques sur de grands volumes de données. Ce modèle hérite du paradigme BigTable de Google.
- **Avantages** :
  - Efficace pour les requêtes analytiques et l'agrégation sur de très grandes quantités de données.
  - Excellent pour la scalabilité horizontale.
  - Adapté à l'écriture massive et à la lecture en batch.
- **Inconvénients** :
  - Peu intuitif pour les développeurs venant du relationnel.
  - Moins adapté aux traitements transactionnels ou nécessitant des jointures complexes.
  - Structure de données moins souple pour les besoins variés.
- **Exemples** : Cassandra, HBase, ScyllaDB.

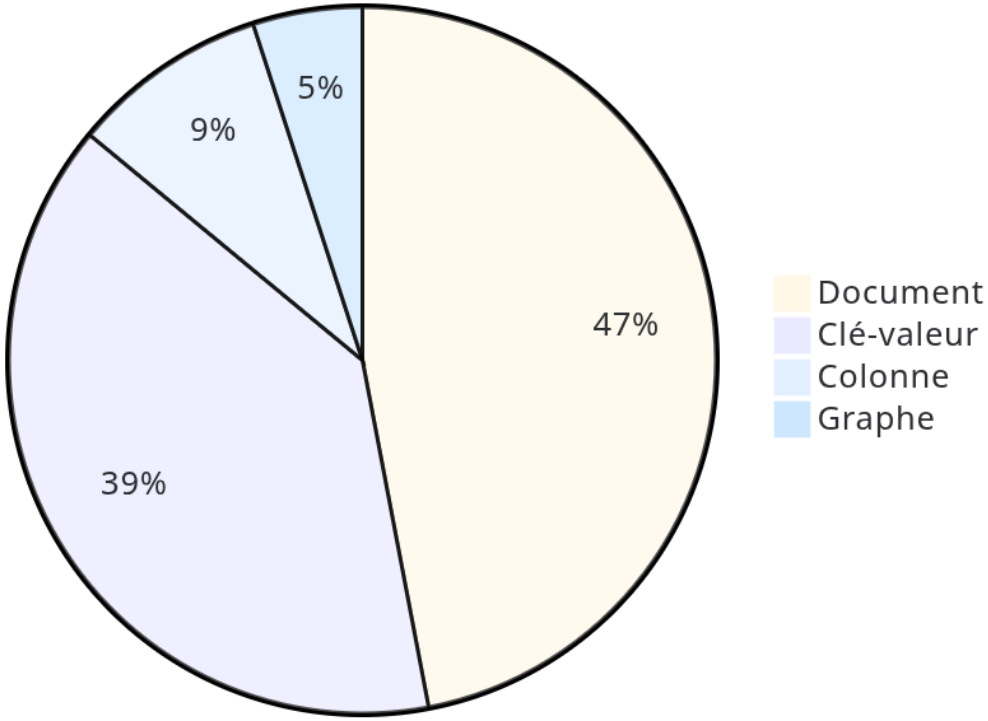
Bases orientées graphes

- **Principe** : Les données sont structurées sous forme de nœuds et de relations (arêtes), permettant de représenter facilement des réseaux complexes (relations sociales, topologies, etc.). Optimisées pour explorer les connexions et chemins entre entités.
- **Avantages** :
  - Idéal pour les données très relationnelles et les requêtes de type graphe (traversées, plus court chemin).
  - Haute performance pour des requêtes impliquant plusieurs niveaux de relations.
  - Naturel pour des domaines comme la recommandation, le social, le réseau.
- **Inconvénients** :
  - Moins adapté au stockage de grandes quantités de données non relationnelles.
  - Modèle spécialisé, pas optimal pour les besoins généralistes.
  - Scalabilité horizontale parfois plus complexe que d'autres types NoSQL.
- **Exemples** : Neo4j, Amazon Neptune.

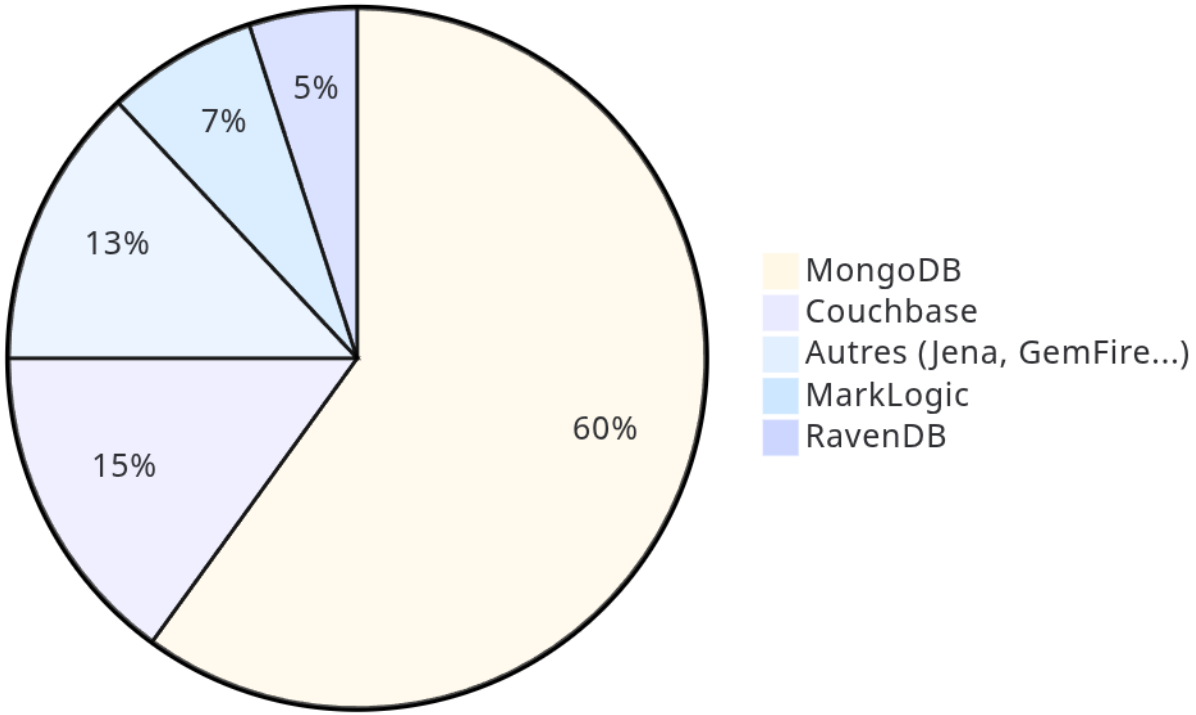
Récapitulatif

Type de base	Principe	Avantages	Inconvénients	Exemples
Clé-valeur	Stockage par paires clé/valeur	Simplicité, rapidité, scalabilité	Limite des requêtes complexes	Redis, DynamoDB
Document	Documents (JSON, XML, BSON...)	Flexibilité, requêtes avancées	Gestion des relations limitée	MongoDB, CouchDB
Colonnes	Organisation par colonnes	Performances analytiques, big data	Complexité, peu adapté au transactionnel	Cassandra, HBase
Graphe	Nœuds + relations	Requêtes relationnelles puissantes	Cas d'usage spécialisés	Neo4j, Neptune

Parts de marché



NoSQL



Documents

À retenir

- Le choix d’une base NoSQL dépend toujours des besoins fonctionnels, de la structure des données et des exigences de performance.

- Chaque type présente ses propres avantages pour des usages ciblés, mais aussi ses limitations structurelles et opérationnelles.

- **Aucune solution n’est universelle** : il est essentiel de bien définir les attentes et les contraintes du projet avant d’opter pour un modèle NoSQL particulier.

MongoDB - Présentation

Introduction

- MongoDB est une base de données NoSQL orientée documents.

- Elle se distingue par le stockage de données sous la forme de documents JSON (BSON en interne), offrant ainsi une grande souplesse et une facilité d'adaptation aux besoins des applications modernes.

- Son approche permet de manipuler des données de structures variées, sans schéma strict imposé, ce qui est particulièrement adapté aux environnements agiles et à l'évolution rapide des données.

Motivations

- **Scalabilité horizontale** : Capacité à distribuer les données sur plusieurs serveurs facilement (sharding).

- **Souplesse du schéma** : Facilite l'évolution et l'hétérogénéité des données sans migration complexe.

- **Performance** pour les lectures et écritures massives : Particulièrement utile pour le big data, le temps réel et le cloud.

- **Simplicité du développement** : Correspondance naturelle avec les objets manipulés dans les langages modernes, évitant les ORM.

Histoire

- **2007** : Création à New York par Dwight Merriman, Eliot Horowitz et Kevin Ryan (alors à DoubleClick).
  - Au départ, le projet portait sur une plateforme PaaS, mais l'équipe a bifurqué vers la conception d'une base de données dédiée pour mieux répondre aux nouveaux enjeux de stockage de données massives et non structurées.

- **2009** : Première version publique sous licence open-source.

- **2013** : Conversion de la société "10gen" en "MongoDB Inc.".

- **2017** : Introduction en bourse sur le NASDAQ (MDB), consolidant sa place en tant que leader NoSQL mondial.

Principes

- **Stockage orienté document** : Les données sont regroupées dans des collections de documents structurés en BSON. Chaque document peut posséder des champs différents, ce qui garantit la flexibilité.

- **Absence de schéma fixe** : La structure d'un document n'est pas imposée, et peut évoluer indépendamment des autres documents.

- **Indexation puissante** : Support d'index complexes (texte, géospatiaux, composés...).

Principes

- **Requêtes riches** : Recherche, agrégation, et manipulation de documents par un langage de requête spécifique (MQL).

- **Haute disponibilité et scalabilité native** : Réplication automatique (Replica Sets), architecture distribuée, sharding intégré pour la montée en charge horizontale.

- **Distribution Géographique** : Conçue pour gérer la répartition et la synchronisation des données à grande échelle, notamment dans des environnements cloud/distribués.

Comparatif

Critère	MongoDB	CouchDB	Amazon DocumentDB	RavenDB
Schéma	Flexible, non imposé	Flexible, non imposé	Compatible MongoDB	Flexible, support schéma
Modèle de données	Document (JSON/BSON)	Document (JSON)	Document (BSON/JSON)	Document (JSON)
Requêtes riches	Oui (MQL, agrégation)	Moins puissant	Presque identique à MQL	Oui
Indexation	Avancée (texte, géo, etc.)	Moins étendue	Index limité VS MongoDB	Bonne

Comparatif

Critère	MongoDB	CouchDB	Amazon DocumentDB	RavenDB
Scalabilité	Sharding natif et auto-scale	Réplication simple	Scalabilité gérée AWS	Scalabilité horizontale
Haute disponibilité	Replica set, multi-site	Réplication master-master	Réplica automatique AWS	Réplication native
Écosystème	Large, intégration cloud (Atlas)	Moins développé	AWS natif	Spécifique (.NET, C#)
Licence	SSPL (open core)	Apache 2.0 (open source)	Propriétaire AWS	Apache 2.0

Comparatif - Synthèse

- **MongoDB** s'impose par sa puissance d'indexation, un langage de requête robuste, une documentation très riche, un large écosystème, une intégration cloud (MongoDB Atlas), et une grande communauté.
- **CouchDB** privilégie la simplicité et la réplication master-master mais offre moins de possibilités d'indexation ou d'agrégation avancée.
- **Amazon DocumentDB** vise la compatibilité API mais ne possède pas toutes les fonctionnalités avancées de MongoDB natif, en particulier concernant le sharding, la recherche textuelle intégrée et certains types d'indexation.
- **RavenDB** vise principalement la communauté .NET mais reprend l'essentiel des concepts de l'orientation document.

Comparatif - Conclusion

- MongoDB occupe une position de leader des bases de données orientées documents grâce à sa souplesse, sa puissance de requêtage, sa montée en charge horizontale native et sa large adoption.
- Ses points forts résident dans sa capacité à s'adapter à des scénarios évolutifs, à gérer de larges volumes, et à offrir une expérience de développement rapide et moderne.
- La concurrence existe (CouchDB, DocumentDB, RavenDB...), mais MongoDB reste la référence du secteur pour la plupart des environnements nécessitant agilité et performance.

Théorème CAP

- Le théorème CAP (ou théorème de Brewer) est un concept fondamental en architecture de systèmes distribués. Il stipule qu'un système de gestion de données distribué ne peut garantir simultanément que deux des trois propriétés suivantes :
  - C – **Cohérence** (Consistency) : Tous les nœuds voient la même donnée au même instant. Après une écriture, toute lecture retourne la dernière valeur écrite (ou une erreur si ce n'est pas possible).
  - A – **Disponibilité** (Availability) : Tout nœud non défaillant renvoie une réponse à chaque requête, même si cette réponse risque de ne pas refléter la version la plus récente des données, en cas de panne réseau.
  - P – **Tolérance** au partitionnement (Partition Tolerance) : Le système continue de fonctionner correctement même lorsqu'une perte de communication survient entre certaines parties du réseau (partition).

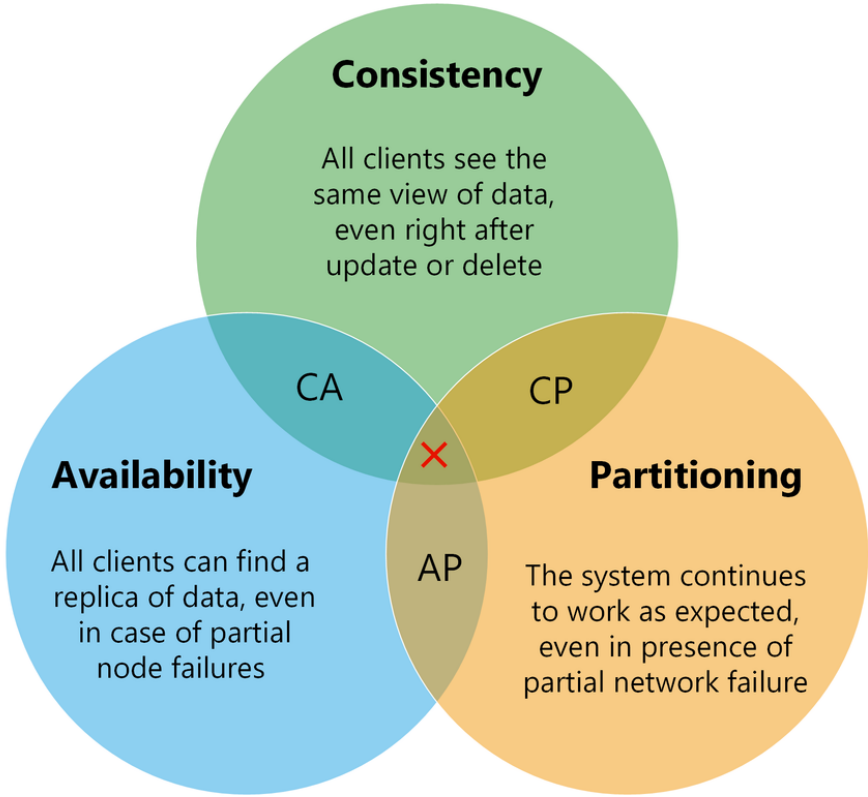
Théorème CAP

- Conséquence : face à une partition réseau (P), un choix s'impose :
  - Sacrifier la cohérence pour maintenir la disponibilité (AP)
  - Ou sacrifier la disponibilité pour assurer la cohérence (CP)

- La plupart des systèmes distribués modernes doivent tolérer les partitions (P). Dans la pratique, les architectes choisissent entre « C » et « A » selon les besoins applicatifs.

Théorème CAP – Position des Bases

Combinaison	Propriété assurée	Propriété sacrifiée	Exemples
CP	Cohérence + Partition	Disponibilité	MongoDB (consistance stricte), HBase
AP	Disponibilité + Partition	Cohérence	Cassandra, CouchDB
CA	Cohérence + Disponibilité	Partition (théorique)	Systèmes non répartis



CAP vs ACID

- **Nature et contexte** :
  - CAP pose les limites des propriétés atteignables dans un système distribué, en insistant sur les compromis à arbitrer (aucun n'est absolu).
  - ACID décrit les exigences à respecter pour garantir la fiabilité des transactions d'une base de données .
- « **Cohérence** » : deux acceptions distinctes
  - Cohérence CAP : tous les nœuds partagent la même version de donnée à l'instant T, quelle que soit l'origine de la demande.
  - Cohérence ACID : application stricte des règles d'intégrité ; le passage d'un état valide à un autre est garanti, mais pas nécessairement synchronisé entre nœuds dans un cluster distribué.
- **Approche transactionnelle**
  - Bases ACID (classiques, SQL) : privilégient la cohérence absolue, acceptant éventuellement une indisponibilité temporaire lors de partitions réseau (orientation CP).
  - Bases NoSQL (ex : MongoDB, Cassandra) : certains systèmes choisissent la disponibilité et la tolérance aux partitions, quitte à sacrifier la cohérence immédiate et à privilégier une cohérence éventuelle (orientation AP ou CP selon configuration).

CAP vs ACID - Comparatif

	CAP Theorem	**ACID Properties**
Finalité	Compromis dans les systèmes distribués	Gestion fiable d'une transaction
Dimension clé	Cohérence (pair node), disponibilité, partition	Atomicité, cohérence (intégrité), isolation, durabilité
Possibilité de tout avoir	Non (seulement 2 sur 3 simultanément)	Oui (en environnement non distribué ou sans partition réseau)
Popularité	NoSQL, systèmes distribués	Bases relationnelles traditionnelles

CAP vs ACID : Récapitulatif

- ACID garantit la robustesse des transactions (souvent dans un contexte centralisé ou faiblement distribué).

- CAP informe sur les compromis à faire dans tout système distribué : il est généralement impossible d'assurer en même temps cohérence forte, disponibilité continue et tolérance aux partitions.

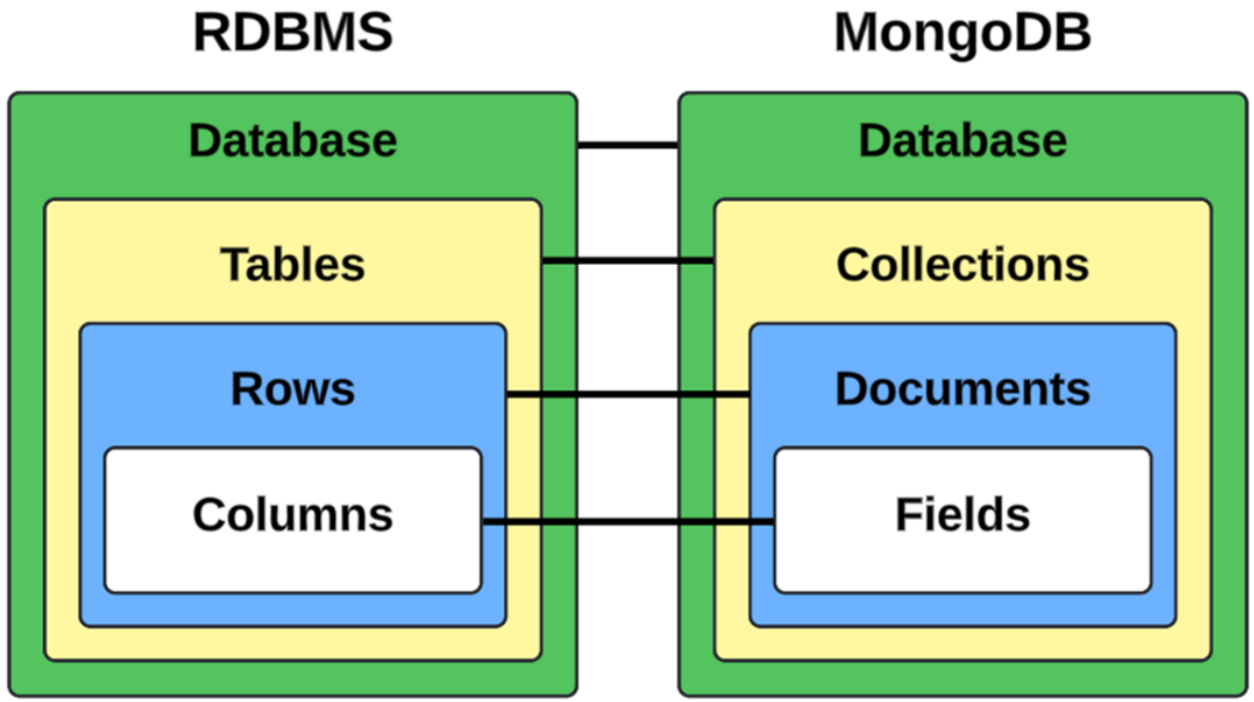
- **Leur bonne compréhension est essentielle pour concevoir des architectures fiables et performantes, adaptées à la nature de l'application et du réseau.**

MongoDB - Concepts

Concepts

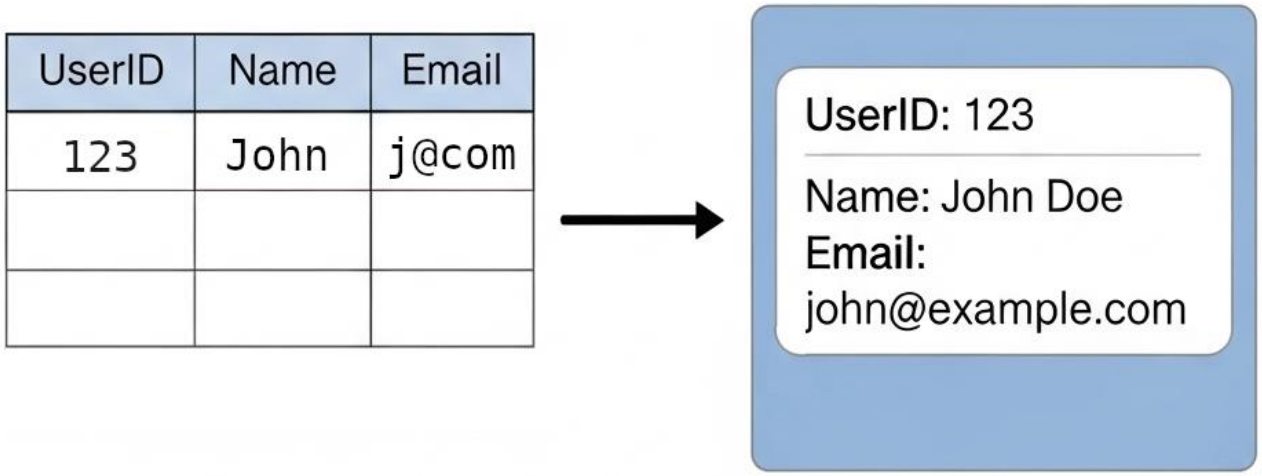
- **Base de données** : Ensemble logique de collections, équivalent à une base dans d'autres systèmes.
- **Collection** : Conteneur de documents, similaire à une table, mais sans schéma fixe : chaque document peut avoir une structure différente.
- **Document** : Unité principale de stockage, structurée en BSON (clé/valeur, imbriquée). Chaque document possède un identifiant unique (_id).
- **Champ** : Élément de donnée d'un document, comparable à une colonne mais chaque document peut avoir des champs différents.
- **Index** : Structures accélérant les requêtes sur un ou plusieurs champs (index simple, composé, texte, géo...).

Concepts - Illustration



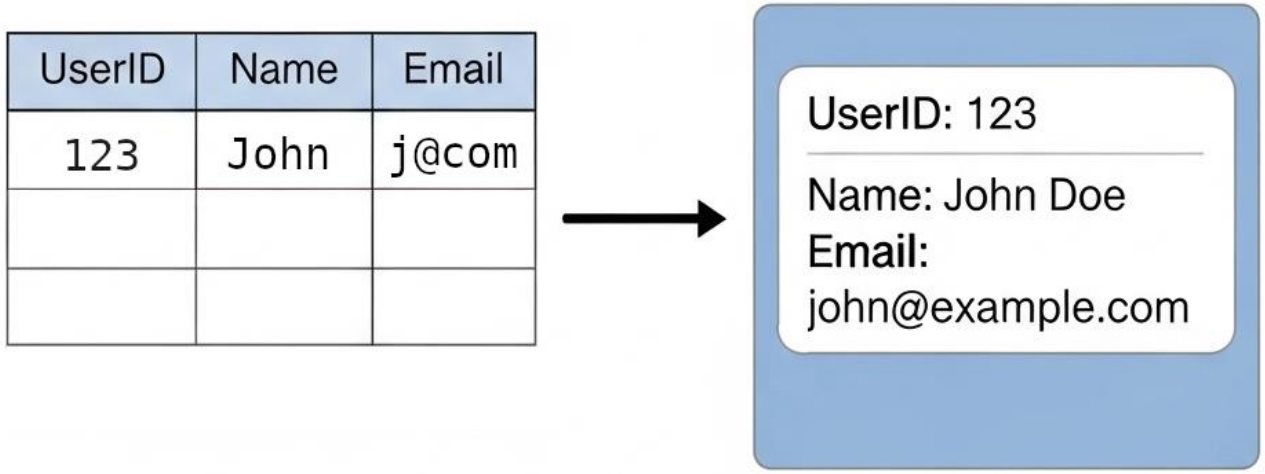
Verticalisation de la donnée

- **Centralisation des attributs** : Toutes les informations liées à une entité (par exemple, un utilisateur) sont regroupées en un seul document ou objet, plutôt que d'être dispersées sur plusieurs lignes ou tables.
- **Hiérarchisation** : L'organisation devient imbriquée ou structurée, intégrant éventuellement des sous-objets ou des listes directement à l'intérieur du même enregistrement.
- **Enrichissement dynamique** : On peut facilement ajouter de nouveaux attributs ou champs sans modification complexe du schéma global.
- **Renforcement de la cohérence du contexte** : Toutes les données pertinentes d'un même utilisateur sont immédiatement accessibles, l'ensemble formant un « bloc vertical » autonome.



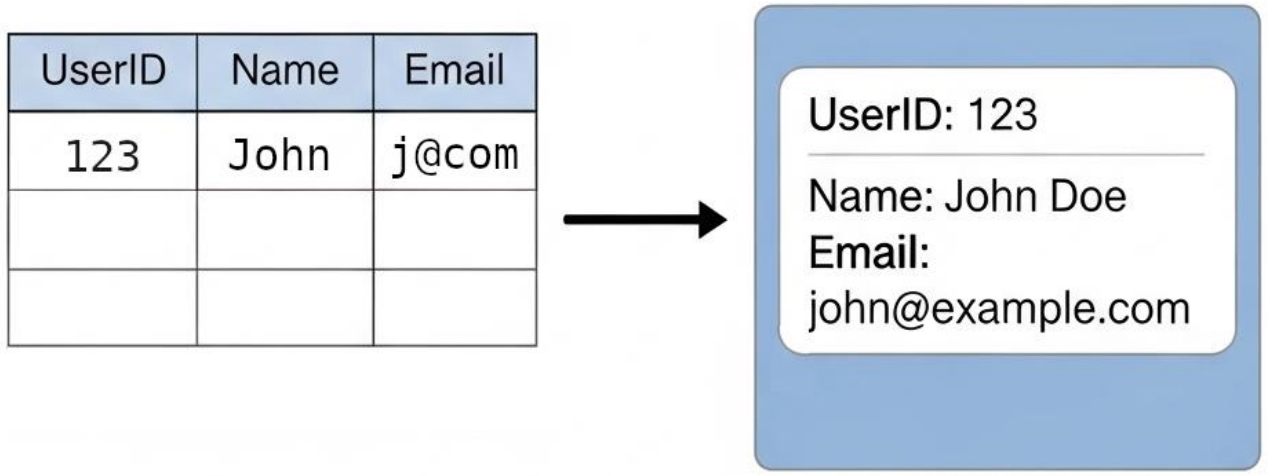
Verticalisation de la donnée

- **Simplicité d'accès:** Un seul document contient toutes les informations sur une entité, ce qui évite les jointures et simplifie les requêtes de lecture.
- **Performance accrue**: Les lectures et consultations sont souvent plus rapides, car il n'est pas nécessaire de combiner plusieurs tables ou lignes. Toutes les données sont déjà localisées au même endroit.
- **Souplesse de modélisation**: Il devient facile d'ajouter ou de retirer des champs selon les besoins métiers, sans migration lourde ou modification universelle du schéma.



Verticalisation de la donnée

- **Réduction du risque d'incohérences**: Puisque l'ensemble des informations pertinentes sont ensemble, il y a moins de défaillances possibles lors des modifications (pas d'anomalies de jointure, moins de doublons cachés).
- **Meilleure correspondance aux structures modernes**: Les documents structurés ainsi représentent mieux les objets applicatifs (objets métier complexes), ce qui facilite l'intégration avec le développement agile et l'itération rapide.



Concepts

- **Requête** : Les requêtes utilisent un langage propre (MQL) permettant de filtrer, trier, agréger les documents.
- **Aggregation Pipeline** : Mécanisme puissant pour transformer et analyser des ensembles de documents, fonctionnant par étapes (stages), chacune réalisant un traitement spécifique (filtre, groupement, tri...).
- **Replica Set** : Mécanisme de réplication pour la haute disponibilité (plusieurs copies synchronisées sur différents serveurs).
- **Sharding** : Répartition horizontale des données sur plusieurs nœuds pour la scalabilité.

Concepts

- **Vue** : Collection virtuelle qui présente les résultats d'une requête d'agrégation prédéfinie. Elle n'enregistre aucune donnée physique : à chaque accès, elle exécute dynamiquement la pipeline d'agrégation définie sur une ou plusieurs collections sous-jacentes.

- **Capped Collection** (collection à taille fixe) : Lorsque la limite est atteinte, les documents les plus anciens sont automatiquement supprimés pour faire place aux nouveaux. Les documents sont stockés dans l'ordre d'insertion et la taille ainsi que le nombre maximal de documents sont configurés lors de la création.

Concepts – Comparaison avec SQL

Concept MongoDB	Concept SQL	Description
Base de données	Database	Ensemble logique de données
Collection	Table	Groupe de documents / lignes, mais sans schéma fixe
Document	Ligne (Row)	Donnée individuelle, structurée (clé/valeur JSON)
Champ	Colonne (Column)	Propriété d'un document / attribut d'une ligne
Index	Index	Accélère les recherches, trie ou filtre
Vue	Vue	collection virtuelle qui présente les résultats d'une requête d'agrégation prédéfinie
_id (MongoDB)	Clé primaire (Primary Key)	Identifiant unique du document / de la ligne
Requête (find)	SELECT	Lecture/filtrage des données

Concepts – Comparaison avec SQL

Concept MongoDB	Concept SQL	Description
Insert	INSERT	Ajout d'un document / d'une ligne
Update	UPDATE	Mise à jour d'un ou plusieurs documents/ lignes
Delete	DELETE	Suppression d'un ou plusieurs documents / lignes
Aggregation Pipeline	GROUP BY + Fonctions d'agrégat	Transformation et analyse avancée des données
Replica Set	Réplication/master-slave	Haute dispo, synchronisation sur plusieurs serveurs
Sharding	Partitionnement / Sharding	Répartition horizontale des données

Différences essentielles

- **Schéma** : MongoDB ne force aucun schéma, chaque document d'une collection peut différer des autres ; SQL impose un schéma structuré et homogène pour toutes les lignes d'une table.
- **Requêtes et agrégation** : Le langage MongoDB s'appuie sur une syntaxe JSON et un pipeline d'agrégation, alors que SQL utilise des mots-clés (SELECT, JOIN, GROUP BY...) plus traditionnels.
- **Relations** : Les jointures complexes sont natives et puissantes en SQL ; en MongoDB, elles existent (opérateur $lookup) mais sont moins centrales, au profit d'une modélisation plus imbriquée ou dénormalisée.
- **Scalabilité** : MongoDB facilite la scalabilité horizontale nativement ; le SQL traditionnel repose souvent sur la scalabilité verticale ou des architectures plus complexes pour le passage à l'échelle.

JSON

- JSON (JavaScript Object Notation) est un format léger d'échange de données, à la fois lisible par l'humain et facilement parseable par les machines. Il sert principalement à la sérialisation et à l'échange de structures de données dans les applications web et entre systèmes.
- Caractéristiques principales de JSON :
  - Syntaxe basée sur des objets et des tableaux (hérité de JavaScript).
  - Clés de type chaîne de caractères ; valeurs de type chaîne, nombre, booléen, tableau, objet ou null.
  - Lisibilité élevée, facile à générer et à manipuler dans la plupart des langages de programmation.

BSON

- BSON (Binary JSON) est le format binaire utilisé par MongoDB pour stocker les documents. Il a été conçu pour enrichir les possibilités du JSON standard, tout en optimisant la performance du stockage et des opérations côté serveur.
- Caractéristiques principales de BSON :
  - Encodage binaire (binaire, compact, mais non lisible directement par l'humain).
  - **Supporte davantage de types de données que JSON** : entiers 32/64 bits, flottants, dates, objets binaires, etc.
  - **Prise en charge native d'un identifiant** (ObjectId) et d'éléments propres aux usages MongoDB.
  - Permet la traversée rapide des documents lors des requêtes et des indexations.

BSON – Types de donnée

Type BSON	Usage principal	Remarques
String	Chaînes de texte	UTF-8
Int32/Int64	Entiers	Choix selon la plage de valeurs
Double	Nombres à virgule flottante	Standard IEEE 754
Decimal128	Haute précision numérique	Calcul financier, scientifique
Boolean	Valeur booléenne	true / false
Null	Absence de valeur	

BSON – Types de donnée

Type BSON	Usage principal	Remarques
Object	Document (structuré, clés/valeurs)	Imbrication possible
Array	Séquence ordonnée de valeurs	Valeurs de types mixtes possibles
Binary data	Fichiers binaires	Images, fichiers, cryptographie
ObjectId	Identifiant unique	12 octets, clé primaire
Date	Dates et heures	Millisecondes (depuis 1970 UTC)
Timestamp	Ordonnancement d'événements	Usage interne MongoDB

BSON – Types de donnée

Type BSON	Usage principal	Remarques
RegExp	Expressions régulières	Format PCRE
JavaScript	Code JS dans les documents	
JS with scope	Code JS + variables de contexte	
Symbol	Symbole (obsolète)	Compatibilité
MinKey / MaxKey	Valeur la plus basse/haute	Usage pour tri/requêtes
Undefined	Valeur non définie (obsolète)	
DBPointer	Référence (obsolète)	

JSON vs BSON

Aspect	JSON	BSON
Format	Texte (lisible humain)	Binaire (optimisé machine)
Types de données	Limité	Étendu (timestamp, binaire...)
Utilisation	Échange web, API, config...	Stockage et transfert interne MongoDB
Performance	Simple, léger	Rapide pour lecture/écriture interne

Document MongoDB

- Dans MongoDB, le document est l'unité fondamentale de stockage : il s'agit d'un enregistrement structuré selon le format BSON mais manipulé côté client en JSON ou objets natifs selon le langage.
- Caractéristiques clés d'un document MongoDB :
  - Constitué de paires clé/valeur, potentiellement imbriquées.
  - Peut contenir des tableaux, des objets imbriqués, divers types de valeurs (nombres, chaînes, dates, identifiants, booléens...).
  - Chaque document possède automatiquement une clé unique _id (typiquement un ObjectId généré automatiquement).
  - Les documents d'une même collection peuvent avoir des structures différentes (pas de schéma imposé).

Documents MongoDB - À retenir

- MongoDB stocke ses données sous forme de documents semblables à du JSON mais enrichis et optimisés grâce au format BSON.

- Cela permet de concilier flexibilité de modélisation et efficacité de traitement au sein de bases massivement distribuées.

MongoDB - Installation

Windows

- Rendez-vous sur le site officiel de MongoDB pour télécharger l'installateur Windows au format msi.
- Lancez l'installateur et suivez les étapes de l'assistant pour choisir les modules à installer et configurer le service MongoDB.
- Option d'installation en "Complete" (par défaut) ou "Custom" (personnalisée).
- Possibilité de lancer MongoDB en tant que service Windows (recommandé) ou manuellement.

MAC OS

- brew tap mongodb/brew
- brew update
- brew install mongodb-community
- brew services start mongodb-community

Linux (debian / ubuntu)

- sudo apt update
- sudo apt install mongodb-org
- sudo systemctl start mongod
- sudo systemctl enable mongod

Choix de la version EE

- Environnements soumis à des exigences de conformité (banques, santé, secteur public...).
- Applications critiques devant garantir haut niveau de sécurité, traçabilité, disponibilité.
- Organisations nécessitant du support dédié, de la maintenance proactive et des SLA contractuels.
- Gestion à grande échelle, automatisée et centralisée de multiples clusters.

CE vs EE

Fonctionnalité	Community Edition	Enterprise Edition
Licence	Open Source (SSPL)	Licence commerciale
Authentification LDAP/Kerberos	Non	Oui
Chiffrement au repos	Non	Oui (avec KMIP)
Auditing	Non	Oui
Ops/Cloud Manager	Non	Oui
Moteur en mémoire	Non	Oui
BI Connector/Intégrations	Limité	Oui
Monitoring avancé	Limité	Oui
Support	Forums/communautaire	Pro, 24/7, SLA
Sauvegarde avancée	Non	Oui (point-in-time, etc.)
Certifications	Non	Oui (OS, conformité...)

MongoDB - Shell

Présentation

- **mongosh** est le shell interactif officiel pour MongoDB, permettant d'interagir en ligne de commande avec une base de données MongoDB.
- Il remplace l'ancien shell mongo et offre une expérience améliorée, notamment grâce à la compatibilité avec JavaScript moderne et une meilleure intégration avec les outils d’administration.
- Utilisé pour :
  - Explorer et manipuler les données.
  - Gérer les collections, index, utilisateurs.
  - Automatiser des tâches d’administration (scripts).

Configuration

- Le prompt de mongosh affiche des informations utiles comme le nom de la base sélectionnée.
- Personnalisation possible via des fichiers de configuration (par exemple, .mongoshrc.js pour charger du code au démarrage).
- Historique des commandes, **autocomplétion** et coloration syntaxique sont activés par défaut.
- Il est possible de modifier le prompt en définissant la variable prompt dans le shell, par exemple : *prompt = () => `db:${db.getName()} > `;*

Aide

- Commandes intégrées :
  - help : affiche l'aide générale.
  - db.help() : aide spécifique sur les méthodes de l'objet db.
  - db.collection.help() : méthodes disponibles pour une collection.
- Accès selon le contexte :
  - Démarrage interactif avec mongosh.
  - Depuis un script avec mongosh script.js.
  - Utilisable dans des environnements différents : terminal local, client distant via SSH, ou via le MongoDB Atlas en ligne.
- Documentation externe :
  - Taper help ou demander l'aide sur une commande retourne souvent un lien vers la documentation officielle.

Fonctions utiles

- **Exploration des bases et collections** :
  - *show dbs* : liste les bases de données.
  - *show collections* : liste les collections de la base courante.
  - *use <db>* : change de base de données.

- **Administration** :
  - *db.createCollection("nom")* : crée une nouvelle collection.
  - *db.coll.drop()* : supprime la collection
  - *db.dropDatabase()* : supprime la base courante.

Fonctions utiles

- **Manipulation des documents** :
  - *db.collection.find()* : recherche de documents.
  - *db.collection.findOne()* : recherche un seul document.
  - *db.collection.insertOne({...})* : insère un document.
  - *db.collection.updateOne({...}, {...})* : met à jour un document.
  - *db.collection.deleteOne({...})* : supprime un document.

- **Scripts et automatisations** :
  - Possibilité d'écrire des scripts JavaScript complexes et de les exécuter directement.

MongoDB - CRUD

Présentation

- Les actions du CRUD représentent le cycle de vie d'une donnée dans une application ou une base.
- Dans MongoDB, elles sont réalisées principalement au niveau des collections (équivalent des tables en SQL) où chaque document peut être créé, lu, modifié ou supprimé via des commandes dédiées.
- L'objectif du CRUD : assurer la gestion complète des données, de leur création jusqu'à leur éventuelle suppression, tout en garantissant leur intégrité, leur lisibilité et la simplicité d'accès pour les développeurs ou administrateurs.
- Il structure la quasi-totalité des interactions avec une base de données, que ce soit pour le développement d'applications, l'analyse, ou l'administration.

Base de données

- Dans MongoDB, il n’existe pas de commande explicite pour créer une base de données. Il suffit d’utiliser la commande suivante : *use nom_de_la_base*

- Pour lister bases de données:
  - *show dbs*
  - *show databases*
  - *db.adminCommand('listDatabases') //affichage plus détaillé*

- Suppression d'une base de données: *db.dropDatabase()*

Collections

- Création: *db.createCollection("nomCollection")*
- Insertion: *db.nomCollection.insertOne({/*document*/}) //création auto de la collection*
- Lister les collections de la base courante:
  - *show collections*
  - *db.getCollectionNames()*
- Statistiques : *db.nomCollection.stats()*
- Suppression: *db.nomCollection.drop()*
- Index:
  - *db.nomCollection.createIndex({champ: 1})*
  - *db.nomCollection.dropIndex("nomDeLindex")*

Méthodes d'insertion

- **InsertOne**
  - Permet d’insérer un seul document dans une collection => *db.collection.insertOne(document)*
  - Retourne un objet indiquant le succès de l’insertion, incluant l’identifiant du document inséré.
  - Prend en charge certaines options (writeConcern, bypassDocumentValidation...).
- **InsertMany**
  - Permet d’insérer efficacement plusieurs documents en une seule opération (meilleures performances que des insertions individuelles).
  - Retourne un objet avec un tableau d’identifiants (_ids) des documents insérés.
  - Option: ordered (booléen) : Définit si les documents doivent être insérés dans l’ordre du tableau (par défaut à vrai). Si une erreur survient, le processus s’arrête ou continue selon la valeur de cette option.

```
db.posts.insertMany([
   { title: "Post 2", content: "...", author: "A" },
   { title: "Post 3", content: "...", author: "B" }
])
```

Méthodes d'insertion - Options

Option	Description	Méthodes
writeConcern	Spécifie le niveau d’accusé de réception demandé (acknowledgment level).	insertOne, insertMany
ordered	Définit si les documents sont insérés dans l’ordre (true par défaut), et si l’insertion s’arrête au premier échec ou continue.	insertMany
bypassDocumentValidation	Ignore la validation de schéma définie sur la collection si défini à true.	insertOne, insertMany
comment	Permet d’ajouter un commentaire pour le profilage ou l’audit.	insertOne, insertMany

Méthodes d'insertion – Champs _id

- **Obligatoire et spécial** : Chaque document inséré doit contenir un champ _id unique dans la collection.
- **Génération automatique** : Si non fourni, MongoDB génère un identifiant unique de type ObjectId.
- **Personnalisable** : On peut définir la valeur de _id soi-même (string, nombre, UUID...), à condition que chaque valeur soit unique.
- **Caractéristiques** :
  - Sert de clé primaire (indexée).
  - Immuable après insertion.
  - Format par défaut (ObjectId) : 12 octets représentant :
    - 4 octets : secondes depuis l'époque Unix
    - 3 octets : identifiant machine
    - 2 octets : PID du process
    - 3 octets : compteur incrémental aléatoire.

```
db.collection.insertMany(
   [
      { _id: 123, nom: "Bob", age: 25 },
      { _id: 124, nom: "Carol", age: 41 },
   ],
   { ordered: false, comment: "Import massif" }
);
```

Recherche: find

- **Objectif** : Récupérer un ou plusieurs documents correspondant à une condition dans une collection.
- **Syntaxe de base** : db.collection.find(query, projection, options)
  - query (facultatif) : critère de sélection (objet JSON). {} ou omission = tous les documents.
  - projection (facultatif) : précise les champs à inclure ou exclure (ex : {nom:1, age:1}).
  - options (facultatif) : tri (sort), pagination (limit, skip), etc.
- **Résultat** : Retourne un **curseur** sur tous les documents trouvés. On peut parcourir les résultats, appliquer des méthodes comme forEach(), ou convertir en tableau (toArray()).

```
db.users.find({age: {$gt: 18}})
db.users.find({}, {nom:1, ville:1, _id:0})
db.users.find({sexe: "F"}).sort({age: -1}).limit(5)
```

Recherche: findOne

- **Objectif** : Récupérer le premier document qui correspond à la condition spécifiée.
- **Syntaxe de base** : db.collection.findOne(query, projection, options)
- **Options**:
  - **sort** : Permet de définir l'ordre des documents examinés et donc celui qui sera retourné. Par exemple : { sort: { age: -1 } } pour obtenir le document avec l'âge le plus élevé.
  - **maxTimeMS** : Définit la durée maximale (en millisecondes) d'exécution de la requête avant qu'elle soit interrompue.
  - **collation** : Pour gérer les règles de comparaison (ex : respect de la casse ou accentuation) selon la langue.
  - **hint** : Suggère un index à utiliser pour la requête.
  - **readConcern / comment / showRecordId** : Autres options avancées pour la gestion de la consistance de lecture, l'ajout de commentaires à la requête ou l'affichage des identifiants internes.

Recherche: Documents imbriqués

- Recherche directe sur un sous-document entier : Il est possible de filtrer sur un document imbriqué en fournissant son arborescence exacte. Le moindre changement d'ordre ou de présence de champs fait échouer la recherche.
- Recherche sur champs spécifiques avec la notation pointée ("dot notation") : Le moyen standard d'interroger un ou plusieurs champs au sein d'un document imbriqué consiste à utiliser la syntaxe : "champImbriqué.sousChamp".
- Exemples:
  - *db.utilisateurs.find({ "adresse.ville": "Paris" })*
  - *db.utilisateurs.find({ "adresse": { "ville": "Paris", "codePostal": "75001" } })*

Recherche: tableaux

- Correspondance simple d'un élément du tableau: pour rechercher les documents dont un tableau contient une valeur précise, il suffit de cibler le champ avec la valeur => *db.collection.find({ tags: "electronics" })*
- Correspondance de plusieurs éléments spécifiques avec **$all**: pour trouver les documents dont le tableau contient plusieurs valeurs spécifiques (l'ensemble, pas forcément dans le même ordre ni exclusivement celles-ci) => *db.collection.find({ tags: { $all: ["electronics", "portable"] } })*
- Recherche sur les éléments d'un tableau (conditions): pour filtrer sur les éléments d'un tableau qui respectent une ou plusieurs conditions :
  - Un seul critère => *db.collection.find({ dim_cm: { $gt: 25 } })*
  - Plusieurs critères sur au moins un élément =>

```
db.collection.find({
   scores: { $elemMatch: { $gt: 80, $lt: 90 } },
});
```

Recherche: tableaux

- Recherche sur plusieurs sous-champs

```
db.collection.find({
    addresses: {
        $elemMatch: { city: "Paris", zipCode: "75001" },
    },
});
```

- Recherche sur une position précise => *db.collection.find({ "tags.0": "electronics" })*
- Recherche sur la longueur d'un tableau => *db.collection.find({ tags: { $size: 3 } })*
- Recherche sur la présence d'un élément issu d'une liste => *db.collection.find({ tags: { $in: ["portable", "desktop"] } })*

Itération avec Cursor

- Le résultat d'une requête n’est pas une liste directe de documents, mais un objet "cursor" (curseur).
- Ce curseur représente un pointeur sur le jeu de résultats et permet d’itérer efficacement (document par document), sans surcharger la mémoire, même sur de très gros volumes de données.
- Il fonctionne de manière similaire à un curseur en base de données relationnelle.
- Principes
  - Le curseur itère en avant : chaque appel récupère le document suivant jusqu’à la fin du jeu de résultats.
  - Gestion optimisée de la mémoire : tous les résultats ne sont pas chargés d’un coup, mais par lots (batch).
  - Expiration : un curseur inactif est automatiquement fermé par MongoDB après 10 minutes d’inactivité afin de libérer les ressources.

Cursor: Méthodes

- **cursor.hasNext()** : Indique s'il reste un document à lire.
- **cursor.next()** : Retourne le document suivant.
- **cursor.forEach(function)** : Exécute une fonction pour chaque document.
- **cursor.toArray()** : Regroupe tous les résultats dans un tableau.
- **cursor.limit(n), cursor.skip(n), cursor.sort({champ:1})** : Permettent de contrôler, paginer et trier les résultats en amont.
- **cursor.close()** : Ferme le curseur et libère les ressources.

Cursor: Exemples

```
var cursor = db.collection.find({
   /* critère */
});
// Parcours manuel
while (cursor.hasNext()) {
   printjson(cursor.next());
}
```

```
db.collection
.find({
   /* critère */
})
.forEach(function (doc) {
   printjson(doc);
});
```

```
var tab = db.collection.find({/* critère */}).toArray();
tab.forEach(printjson);
```

Cursor: Avantages

- **Performance** : Les curseurs gèrent efficacement les gros volumes de données.

- **Contrôle** : Vous pouvez itérer, trier, limiter ou modifier à la volée la façon dont vous parcourez les résultats.

- **Flexibilité** : Possibilité de traitement synchrone ou asynchrone (ex : forEach() ou logique personnalisée dans while/next).

Opérateurs de projection

Nom	Description	Exemple
$	Projette le premier élément d'un tableau qui correspond à la condition de la requête.	*db.notes.find({scores: {$gte: 15}}, {"scores.$": 1})* renvoie uniquement la première note ≥15 du tableau.
$elemMatch	Projette le premier élément du tableau qui correspond à toutes les conditions de $elemMatch.	*db.notes.find({}, {scores: {$elemMatch: {type: "math", value: {$gte: 10}}}})* ne projette que les sous-documents validés.
$meta	Projette des métadonnées par-document (score de texte plein, index, etc.).	db.articles.find({}, {title: 1, score: {$meta: "textScore"}}) ajoute un champ score basé sur la pertinence texte.
$slice	Limite le nombre d'éléments renvoyés d'un tableau, avec possibilité de sauter un nombre donné d'éléments.	*db.courses.find({}, {students: {$slice: 3}})* affiche les trois premiers inscrits à chaque cours.

Opérateurs de projection

Nom	Description	Exemple
Inclusion	Projette (garde) uniquement les champs spécifiés.	*db.users.find({}, {name: 1, age: 1})* ne retourne que name, age et _id.
Exclusion	Supprime les champs spécifiés du résultat.	*db.users.find({}, {password: 0, email: 0})* retourne tous les champs sauf password et email.
Dot notation	Permet de cibler un champ imbriqué dans un document ou un tableau.	*db.orders.find({}, {"customer.address.city": 1})* retourne seulement la ville de l'adresse client.

Opérateurs de comparaison

Nom	Description	Exemple d'utilisation
$eq	Sélectionne les documents dont la valeur du champ est égale à la valeur spécifiée.	*{ age: { $eq: 30 } }* ou simplement *{ age: 30 }*
$ne	Sélectionne les documents dont la valeur du champ est différente de la valeur donnée.	*{ status: { $ne: "active" } }*
$gt	Sélectionne les documents dont la valeur du champ est strictement supérieure à celle indiquée.	*{ score: { $gt: 18 } }*
$gte	Sélectionne les documents dont la valeur du champ est supérieure ou égale à celle indiquée.	*{ salary: { $gte: 3000 } }*

Opérateurs de comparaison

Nom	Description	Exemple d'utilisation
$lt	Sélectionne les documents dont la valeur du champ est strictement inférieure à celle indiquée.	*{ age: { $lt: 40 } }*
$lte	Sélectionne les documents dont la valeur du champ est inférieure ou égale à celle indiquée.	*{ quantity: { $lte: 100 } }*
$in	Sélectionne les documents dont la valeur du champ correspond à l'une des valeurs du tableau.	*{ status: { $in: ["active", "pending"] } }*
$nin	Sélectionne les documents dont la valeur du champ ne correspond à aucune des valeurs du tableau.	*{ category: { $nin: ["sold", "obsolete"] } }*

Opérateurs logiques

Nom	Description	Exemple d'utilisation
$and	Toutes les conditions doivent être vraies ; prend un tableau de requêtes à évaluer avec ET logique (AND).	*{ $and: [{ age: { $gte: 18 } }, { age: { $lte: 25 } }] }*
$or	Au moins une des conditions doit être vraie ; prend un tableau de requêtes à évaluer avec OU logique (OR).	*{ $or: [{ ville: "Paris" }, { ville: "Lyon" }] }*
$nor	Aucune condition ne doit être vraie ; retourne les documents qui ne vérifient aucune des requêtes données.	*{ $nor: [{ ville: "Paris" }, { age: { $lt: 18 } }] }*
$not	Inverse le résultat d'une expression de requête (NON logique) ; à combiner généralement avec d'autres opérateurs.	*{ age: { $not: { $gte: 18 } } }*

Opérateurs "élément"

Nom	Description	Exemple d'utilisation
$exists	Vérifie si un champ est présent ou absent dans un document.	*{ email: { $exists: true } }* sélectionne les documents contenant le champ email
$type	Sélectionne les documents où le champ est d'un type BSON précis (nombre, string, etc.)	*{ age: { $type: "int" } }* sélectionne les documents où age est un entier
$jsonSchema	Permet de valider les documents selon un schéma JSON défini (validation avancée).	*{ $jsonSchema: { properties: { age: { type: "number" } } } }*
$mod	Sélectionne les documents où la valeur d'un champ numérique satisfait un modulo donné	*{ age: { $mod: 5 } }* tous les documents dont age est multiple de 5

Opérateurs "Tableau"

Nom	Description	Exemple d'utilisation
$all	Sélectionne les documents dont le tableau contient toutes les valeurs spécifiées (dans n'importe quel ordre).	*{ tags: { $all: ["portable", "audio"] } }*
$elemMatch	Sélectionne les documents dont au moins un élément du tableau satisfait toutes les conditions données.	*{ scores: { $elemMatch: { type: "math", value: { $gt: 14 } } } }*
$size	Sélectionne les documents où le tableau contient exactement le nombre d'éléments spécifié.	*{ tags: { $size: 3 } }*

Opérateurs "Tableau"

Nom	Description	Exemple d'utilisation
$in	Sélectionne les documents dont au moins un élément du tableau est présent dans une liste de valeurs.	*{ languages: { $in: ["fr", "en"] } }*
$nin	Sélectionne les documents dont aucun élément du tableau n'appartient à une liste de valeurs.	*{ categories: { $nin: ["sold", "archived"] } }*
dot notation	Permet de filtrer sur la présence d'une valeur dans un tableau de sous-documents ou d'accéder à un champ d'objet du tableau.	*{ "addresses.city": "Paris" }*

Opérateurs "Évaluation"

Nom	Description	Exemple d'utilisation
$expr	Permet d'utiliser des expressions d'agrégation dans la requête (comparaison de champs, calculs, etc.).	*{ $expr: { $gt: ["$qtyVente", "$qtyStock"] } }* sélectionne les documents où qtyVente > qtyStock
$regex	Recherche les documents dont la valeur du champ correspond à une expression régulière.	*{ email: { $regex: /@gmail\.com$/ } }*
$text	Réalise des recherches textuelles plein texte sur les champs indexés pour cela.	*{ $text: { $search: "mongodb rapide" } }*
$where	Permet de passer une expression JavaScript personnalisée pour filtrer les documents. Peu performant, à éviter sur de gros volumes.	*{ $where: "this.age < this.score" }*

Recherche - Exercices

- Base de données training
  - Affichez tous 10 premiers documents de la collection `posts`.
  - Trouvez tous les articles (posts) rédigés par l'auteur "machine".
  - Trouvez tous les articles qui contiennent le tag "constitutional".

- Base de données airbnb
  - Trouvez tous les logements (`listingsAndReviews`) à "Brooklyn" qui peuvent accueillir au moins 4 personnes (`accommodates`).
  - Trouvez tous les logements dont le prix (`price`) est inférieur à 100 et qui ont un score de propreté (`review_scores.review_scores_cleanliness`) de 10.

Mise à jour

- **updateOne()**
  - Description : Met à jour le premier document correspondant au filtre.
  - Arguments :
    - Filtre de sélection
    - Opérations de mise à jour (ex : $set, $inc)
    - Options (ex : upsert)
  - Usage courant : Modifier un seul document dans une collection, même s'il en existe plusieurs correspondant au filtre.
- **updateMany()**
  - Description : Met à jour tous les documents qui correspondent au filtre fourni.
  - Arguments :
    - Filtre de sélection
    - Opérations de mise à jour
    - Options (ex : upsert)
  - Usage courant : Appliquer la même modification à plusieurs documents simultanément.

Mise à jour

- **replaceOne()**
  - Description : Remplace complètement le premier document correspondant par un nouveau document.
  - Arguments :
    - Filtre de sélection
    - Nouveau document (remplace l'ancien : toute propriété non précisée disparait)
    - Options (ex : upsert)
  - Particularité : N'accepte pas d'opérateurs ($set, $inc...), la structure doit être entière.
- **Autres opérations**
  - findOneAndUpdate() : Trouve un document, applique la modification, puis retourne soit l'ancien, soit le nouveau document selon les options.
  - findOneAndReplace() : Trouve un document, le remplace entièrement par un autre et retourne (optionnellement) l'ancien ou le nouveau.
  - findOneAndDelete() : Trouve un document, le supprime, et retourne le document supprimé

Mise à jour - Exemples

```
// Mettre à jour un seul document
db.collection.updateOne({ nom: "Jean" }, { $set: { age: 30 } });


// Mettre à jour plusieurs documents
db.collection.updateMany({ pays: "France" }, { $inc: { visites: 1 } });


// Remplacer complètement un document
db.collection.replaceOne({ nom: "Claire" }, { nom: "Claire", age: 32, ville: "Paris" });
```

Mise à jour - Bulk

- L’opération **bulkWrite** permet d’exécuter plusieurs opérations d’écriture (insertion, mise à jour, suppression, remplacement) sur une collection en un seul appel.
  - C’est la méthode privilégiée pour le traitement par lots, améliorant ainsi la performance et l’efficacité.
- Principales caractéristiques
  - Parfaite pour les lots d'opérations (insert, update, delete, replace, etc.).
  - Réduit le nombre d’allers-retours entre l’application et la base de données, limitant ainsi la latence réseau.
  - Peut mélanger les types d’opérations dans le même batch.
  - Supporte l’exécution ordonnée ou non ordonnée :
    - Ordonnée (ordered: true, par défaut) : les opérations sont exécutées dans l’ordre, un échec stoppe la suite.
    - Non ordonnée (ordered: false) : poursuit même en cas d’erreur, ordre non garanti.
  - Résultat : rapport détaillant pour chaque type d’opération le nombre d’inserts, updates, deletes réussis, IDs insérés, etc.

Bulk - Exemple

```
db.students.bulkWrite([
   { insertOne: { document: { nom: "Alice", age: 22 } } },
   { updateOne: { filter: { nom: "Bob" }, update: { $set: { age: 30 } } } },
   { deleteOne: { filter: { nom: "Claire" } } },
   {
      replaceOne: {
         filter: { nom: "David" },
         replacement: { nom: "David", age: 25 },
      },
   },
]);
```

Mise à jour - Opérateurs de champs

- **$set** : Définit la valeur d'un champ (créé le champ s'il n'existe pas).
  - Exemple : *{ $set: { age: 30 } }*
- **$unset** : Supprime un champ d'un document.
  - Exemple : *{ $unset: { age: "" } }*
- **$inc** : Incrémente (ou décrémente) une valeur numérique.
  - Exemple : *{ $inc: { score: 2 } }*
- **$mul** : Multiplie la valeur d'un champ numérique.
  - Exemple : *{ $mul: { prix: 1.2 } }*
- **$rename** : Renomme un champ.
  - Exemple : { $rename: { "nom": "prenom" } }

Mise à jour - Opérateurs de champs

- **$currentDate** : Met à jour la date/heure d'un champ à la date courante (type Date ou Timestamp).
  - Exemple : *{ $currentDate: { lastModified: true } }*
- **$setOnInsert** : Définit la valeur d'un champ uniquement lors de l'insertion (upsert).
  - Exemple : *{ $setOnInsert: { statut: "Nouveau" } }*
- **$min** : Met à jour un champ uniquement si la valeur spécifiée est inférieure à celle existante.
  - Exemple : *{ $min: { age: 18 } }*
- **$max** : Met à jour un champ uniquement si la valeur spécifiée est supérieure à celle existante.
  - Exemple : *{ $max: { score: 100 } }*

Mise à jour - Opérateurs de tableau

- **$push** : Ajoute un élément à la fin d'un tableau.
  - Exemple : *{ $push: { tags: "nouveau" } }*
- **$addToSet** : Ajoute un élément à un tableau seulement s'il n'existe pas déjà.
  - Exemple : *{ $addToSet: { tags: "unique" } }*
- **$pop** : Supprime le premier ou le dernier élément d'un tableau (1=dernier, -1=premier).
  - Exemple : *{ $pop: { tags: -1 } }*
- **$pull** : Supprime toutes les occurrences correspondant à une valeur dans un tableau.
  - Exemple : *{ $pull: { tags: "obsolete" } }*
- **$pullAll** : Supprime toutes les occurrences des valeurs listées dans un tableau.
  - Exemple : *{ $pullAll: { tags: ["foo", "bar"] } }*

Mise à jour - Autres opérateurs

- Modificateurs d’opérateurs sur les tableaux, utilisables en combinaison, notamment avec **$push** :
  - **$each** : Ajoute plusieurs éléments à la fois.
  - **$slice** : Limite la taille du tableau.
  - **$sort** : Trie les éléments ajoutés.
  - **$position** : Précise la position d’insertion dans le tableau.
- Opérateurs particuliers
  - **$bit** : Effectue des opérations bit-à-bit sur des champs entiers (and, or, xor).
  - **$isolated** : Utilisé pour garantir l'isolation d'une opération de mise à jour (usage rare et avancé).

Mise à jour – Exemple de combinaison

```
db.coll.updateOne(
   { _id: 1 },
   {
      $set: { statut: "traité" },
      $inc: { compteur: 1 },
      $push: { logs: { date: new Date(), action: "mise à jour" } },
   }
);
```

Mise à jour - Exercices

- Base de données training
  - Insérez un nouveau document dans la collection `posts` avec un `title`, un `author` et un `body` de votre choix.
  - Mettez à jour l'article intitulé "US Constitution" pour ajouter un nouveau tag "law" à la liste des `tags`.

- Base de données supplies
  - Pour la vente avec l'ID "5bd761dcae323e45a93ccfe8", mettez à jour la quantité de "pens" à 10.
  - Pour l'article "Gettysburg Address", retirez les tags "civil" et "war" de la liste des `tags`.

Bulk - Exercices

- Construisez et exécutez une seule opération `bulkWrite` sur la collection `sales` (base de données supplies) qui accomplit les tâches suivantes dans l'ordre :
  - **`insertOne`** : Insérez une nouvelle vente pour le magasin de "New York" avec au moins deux articles.
  - **`updateOne`** : Pour la vente avec l'ID `"5bd761dcae323e45a93ccfe8"`, le client a mis à jour son niveau de satisfaction. Modifiez la valeur de `customer.satisfaction` à `5`.
  - **`updateMany`** : La direction a décidé qu'un audit est nécessaire pour toutes les ventes réalisées à "Denver". Ajoutez un nouveau champ `audit_required: true` à tous les documents où `storeLocation` est "Denver".
  - **`deleteMany`** : Les données concernant les ventes effectuées par téléphone (`Phone`) sont obsolètes et doivent être supprimées. Supprimez tous les documents où `purchaseMethod` est "Phone".
  - **`replaceOne`** : La vente avec l'ID `"5bd761dcae323e45a93ccff0"` a été enregistrée avec des erreurs. Remplacez l'intégralité du document par un nouveau document correct (vous pouvez inventer les données pour le nouveau document, en gardant le même `_id`).

Suppression – deleteOne()

- **Description** : Supprime le premier document correspondant au filtre spécifié.
- **Syntaxe** : *db.collection.deleteOne({ critère }, options)*
  - Si plusieurs documents correspondent au filtre, seul le premier (dans l'ordre naturel) est supprimé.
- **Options** : writeConcern, collation (comparaison locale), hint (spécification d'index).
- **Retour** : Un objet avec le nombre de documents supprimés (deletedCount).
- **Cas d'usage** : Suppression ponctuelle, par identifiant ou critère unique.

Suppression – DeleteMany()

- **Description** : Supprime tous les documents qui correspondent au filtre spécifié.
- **Syntaxe** : *db.collection.deleteMany({ critère }, options)*
  - Si le filtre est vide ({}), tous les documents de la collection sont supprimés : à utiliser avec grande précaution !
- **Options** : writeConcern, collation, hint.
- **Retour** : Un objet indiquant le nombre de documents supprimés.
- **Cas d'usage** : Opérations de nettoyage, suppression en masse basée sur une règle métier (exemple : logs dépassés, comptes inactifs...).

Suppression - Exemples

```
// Supprimer un document par _id
db.users.deleteOne({ _id: ObjectId("..." ) })


// Supprimer tous les utilisateurs basés en France
db.users.deleteMany({ country: "France" })
```

Suppression - Exercices

- Base de données supplies
  - Supprimez tous les commentaires (comments) de l'auteur "Elizabet Kleine" de tous les articles.
  - Pour toutes les ventes effectuées à "Denver" dans la collection `sales`, supprimez tous les articles ("items") du type "notepad" de la liste des achats.

- Base de données airbnb
  - Dans la collection `listingsAndReviews`, pour le logement dont le `_id` est "1003530", supprimez le champ `neighborhood_overview`.
  - Dans la collection `listingsAndReviews`, supprimez tous les logements qui sont situés au "Brazil" et dont le `price` est supérieur à 500.

MongoDB - Transactions

Présentation

- MongoDB prend en charge les transactions ACID (Atomicité, Cohérence, Isolation, Durabilité) pour les opérations multi-documents et multi-collections depuis la version 4.0 (et pour les clusters shardés depuis la version 4.2).

- Cela permet d'assurer qu'un ensemble d'opérations s'effectue de manière « tout ou rien » : soit toutes les modifications sont appliquées, soit aucune en cas d'échec

Principes

- **Session obligatoire** : Toute transaction doit être démarrée dans le cadre d'une session. Une session ne peut avoir qu'une seule transaction ouverte à la fois. Si la session est interrompue, la transaction en cours est annulée.

- **Déroulement typique** :
  - Création et ouverture d'une session ;
  - Démarrage de la transaction ;
  - Exécution des opérations (insert, update, delete, etc.) ;
  - Validation (commitTransaction) ou annulation (abortTransaction) de la transaction.

Exemple

```
const session = db.getMongo().startSession();
session.startTransaction();
try {
   // Opérations CRUD ici (insert/update/delete)
   session.commitTransaction();
} catch (error) {
   session.abortTransaction();
}
session.endSession();
```

Caractéristiques

- **Atomicité multi-documents/collections** : Les transactions permettent d'enchaîner des modifications sur plusieurs collections ou bases, en assurant une cohérence globale (ex : débiter et créditer deux comptes dans des collections différentes).

- **Performance** : Les transactions impactent la performance, surtout lors de gros volumes ou de longues transactions. Il faut privilégier des transactions courtes.

- **Support cluster** : Requiert l'utilisation d'un replica set ou d'un cluster shardé (non disponible sur les instances standalone).

Caractéristiques

- **Timeout** : Par défaut, une transaction est annulée si elle dure plus de 60s après le démarrage.

- **Write Concern & Read Concern** : Les niveaux de garantie utilisés lors de la transaction (majorité, local...) peuvent être ajustés pour moduler le compromis entre cohérence et performance.

Avantages

- **Garanties ACID** : Les transactions MongoDB offrent les propriétés classiques d'une transaction, comparables à celles des SGBD relationnels, là où auparavant seule l'opération sur un document individuel était atomique.

- **Alignement usage/data** : Permet d'écrire des logiques transactionnelles là où la souplesse du modèle document ne suffisait plus pour garantir l'intégrité applicative.

Bonnes pratiques

- À privilégier si plusieurs modifications interdépendantes doivent être appliquées ou annulées ensemble (ex : traitements bancaires, inventaires critiques, synchronisation de données redondantes).

- À éviter pour des opérations de masse, des traitements analytiques ou toutes les situations où la performance prime sur la stricte cohérence. Le modèle document de MongoDB permet souvent d'éviter les transactions grâce à la « verticalisation » de la donnée, mais elles restent indispensables dans certains scénarios complexes.

Exercices

- La collection `accounts` ne contient pas de champ `balance`. Pour cet exercice, vous devez d'abord choisir deux comptes (par exemple, ceux avec les `account_id` `100238` et `100282`) et leur ajouter un champ `balance` avec une valeur initiale de `5000`.
- Vous devez écrire un script qui réalise les étapes suivantes dans le bon ordre :
  - **Démarrer une session** (`startSession()`).
  - **Démarrer la transaction** (`startTransaction()`).
  - À l'intérieur d'un bloc `try...catch` :
    - **Débiter le compte source :** Mettez à jour le compte `100238` en décrémentant (`$inc`) sa `balance` de `500`. **Important :** Votre filtre de mise à jour doit également vérifier que le solde est suffisant avant le débit (`{ balance: { $gte: 500 } }`).
    - **Créditer le compte de destination :** Mettez à jour le compte `100282` en incrémentant (`$inc`) sa `balance` de `500`.
    - **Valider la transaction :** Si les deux opérations réussissent, validez la transaction avec `commitTransaction()`.
  - Dans le bloc `catch` :
    - **Annuler la transaction :** En cas d'erreur (par exemple, solde insuffisant), annulez toutes les opérations avec `abortTransaction()`.
  - **Terminer la session** (`endSession()`).

MongoDB - Agrégation

Présentation

- L’agrégation en MongoDB permet d’analyser, combiner et transformer des ensembles de documents pour obtenir des résultats synthétiques ou effectuer des analyses avancées.

- Il s’agit d’une fonctionnalité puissante, équivalente à l’utilisation de GROUP BY, SUM, COUNT, etc., dans les bases relationnelles, mais avec une approche plus flexible et modulaire.

Présentation

- L’agrégation consiste à traiter un grand nombre de documents d’une collection afin de produire un jeu de résultats transformés : statistiques, regroupements, tris, manipulations de structures, etc.

- Elle s’effectue principalement via le Framework d’agrégation de MongoDB, dont le cœur est le concept de **pipeline**.

Pipeline d'agrégation

- Un pipeline est une succession d’étapes (ou stages) par lesquelles vont passer les documents.
- Chaque étape reçoit en entrée le flux de documents généré par l’étape précédente, effectue une opération, puis transmet le résultat à l’étape suivante.
- On peut combiner autant d'étapes que nécessaire, ce qui permet de composer des requêtes complexes à partir de sous-étapes simples.

```
db.collection.aggregate([
  {
    /* stage 1 */
  },
  {
    /* stage 2 */
  },
  {
    /* ... */
  },
]);
```

Opérateurs de filtrage ou de sélection

Étape	Description	Exemple d'utilisation
$match	Filtre les documents selon des critères (équivalent WHERE)	*{ $match: { statut: "actif", age: { $gte: 18 } } }*
$project	Sélectionne ou modifie les champs à transmettre	*{ $project: { nom: 1, age: 1 } }*
$limit	Limite le nombre de résultats	*{ $limit: 5 }*
$skip	Ignore les premiers N documents	*{ $skip: 10 }*
$sort	Trie les documents	*{ $sort: { date: -1 } }*

Opérateurs de filtrage ou de sélection

Étape	Description	Exemple d'utilisation
$unwind	Déplie un tableau en plusieurs documents	*{ $unwind: "$tags" }*
$lookup	Jointure avec une autre collection	*{ $lookup: { from: "users", localField: "userId", foreignField: "_id", as: "userInfos" } }*
$unionWith	Réalise l'union du flux du pipeline courant avec les résultats d'une autre collection	*{ $unionWith: { coll: "collection2", pipeline: [{ $project: { champ: 1 } }*
$group	Regroupe les documents par une clé	*{ $group: { _id: "$categorie", total: { $sum: 1 } } }*
$addFields	Ajoute de nouveaux champs calculés	*{ $addFields: { ageDecennie: { $multiply: [{ $floor: { $divide: ["$age", 10] } }, 10] } } }*

Opérateurs de filtrage / comparaison

Nom	Description	Exemple d'utilisation
$eq	Égal à	*{ age: { $eq: 30 } }*
$ne	Différent de	*{ statut: { $ne: "inactif" } }*
$gt	Supérieur à	*{ score: { $gt: 80 } }*
$gte	Supérieur ou égal à	*{ age: { $gte: 18 } }*
$lt	Inférieur à	*{ stocks: { $lt: 5 } }*
$lte	Inférieur ou égal à	*{ prix: { $lte: 100 } }*
$in	Inclus dans un tableau de valeurs	*{ pays: { $in: ["FR", "BE", "CH"] } }*
$nin	Non inclus dans un tableau de valeurs	*{ genre: { $nin: ["SF", "Romance"] } }*

Opérateurs de filtrage / comparaison

Nom	Description	Exemple d'utilisation
$exists	Teste l'existence du champ	*{ promo: { $exists: true } }*
$type	Teste le type BSON	*{ age: { $type: "int" } }*
$regex	Correspondance par expression régulière	*{ nom: { $regex: "^A" } }*
$and	Toutes les conditions doivent être vraies	*{ $and: [{ actif: true }, { age: { $gte: 18 } }] }*
$or	Au moins une condition doit être vraie	*{ $or: [{ pays: "FR" }, { pays: "BE" }] }*
$nor	Vrai si toutes les conditions sont fausses	*{ $nor: [{ $eq: ["$statut", "banni"] }] }*
$not	Négation d'une condition	*{ score: { $not: { $gte: 50 } } }*
$cond	Condition classique "if-then-else"	*{ $cond: [{ $gte: ["$score", 60] }, "ok", "à revoir"] }*
$ifNull	Retourne une valeur par défaut	*{ $ifNull: ["$ville", "Inconnue"] }*

Opérateurs de transformation

Nom	Description	Exemple
$set	Ajoute ou modifie un ou plusieurs champs sur chaque document du pipeline (équivalent à $addFields).	*{ $set: { total: { $sum: ["$champ1", "$champ2"] }, marge: 42 } }*
$unset	Supprime un ou plusieurs champs des documents en sortie d'étape.	*{ $unset: ["champ1", "champ2"] }*

Opérateurs d'accumulation

Nom	Description	Exemple
$sum	Calcule la somme des valeurs d’un champ	{ $sum: "$montant" }
$avg	Calcule la moyenne des valeurs d’un champ	{ $avg: "$score" }
$min	Renvoie la valeur minimale d’un champ	{ $min: "$note" }
$max	Renvoie la valeur maximale d’un champ	{ $max: "$prix" }

Opérateurs d'accumulation

Nom	Description	Exemple
$push	Ajoute chaque valeur rencontrée dans un tableau	{ $push: "$nom" }
$addToSet	Ajoute dans un tableau en évitant les doublons	{ $addToSet: "$email" }
$first	Renvoie la première valeur d’un champ (après tri)	{ $first: "$date" }
$last	Renvoie la dernière valeur d’un champ (après tri)	{ $last: "$date" }
$count	Compte le nombre de documents	{ $count: "total" }

Opérateurs arithmétiques

Nom	Description	Exemple
$add	Additionne des valeurs	*{ $add: ["$prix", "$taxes"] }*
$subtract	Soustrait une valeur d'une autre	*{ $subtract: ["$total", "$remise"] }*
$multiply	Multiplie des valeurs	*{ $multiply: ["$quantite", "$prix_unitaire"] }*
$divide	Divise une valeur par une autre	*{ $divide: ["$total", "$nombre"] }*
$mod	Calcule le reste d'une division entière	*{ $mod: ["$score", 10] }*

Opérateurs de chaînes de caractères

Nom	Description	Exemple
$concat	Concatène plusieurs chaînes	*{ $concat: ["$prenom", " ", "$nom"] }*
$substrCP	Extrait une sous-chaîne (compatible Unicode)	*{ $substrCP: ["$code", 0, 2] }*
$toLower	Convertit en minuscules	*{ $toLower: "$nom" }*
$toUpper	Convertit en majuscules	*{ $toUpper: "$ville" }*
$split	Coupe selon un séparateur	*{ $split: ["$email", "@"] }*
$strLenCP	Compte la longueur d'une chaîne (Unicode)	*{ $strLenCP: "$texte" }*
$trim	Supprime les espaces en début et fin	*{ $trim: { input: "$titre" } }*

Opérateurs de tableaux

Nom	Description	Exemple
$size	Renvoie la taille d'un tableau	*{ $size: "$tags" }*
$filter	Filtre les éléments d'un tableau	*{ $filter: { input: "$arr", as: "x", cond: { $gt: ["$$x", 5] } } }*
$in	Vérifie l'inclusion d'une valeur dans un tableau	*{ $in: ["valeur", "$liste"] }*
$arrayElemAt	Renvoie l'élément d'un tableau à un index donné	*{ $arrayElemAt: ["$notes", 2] }*
$isArray	Vérifie si une valeur est un tableau	*{ $isArray: "$tags" }*
$range	Génère un tableau de nombres	*{ $range: [0, 10, 2] }*

Autres opérateurs

Nom	Description	Exemple
$geoNear	Recherche des documents proches d'un point donné	Utilisé dans le stage *$geoNear*
$sample	Sélectionne aléatoirement un sous-ensemble	*{ $sample: { size: 5 } }*
$collStats	Retourne des statistiques sur une collection (nombre de documents, taille, informations de stockage ou de latence, etc.).	*{ $collStats: { count: {}, storageStats: {}, latencyStats: { histograms: true } } }*
$indexStats	Fournit des statistiques d'utilisation sur chaque index de la collection (nombre de lectures, fréquentation, etc.).	*db.collection.aggregate([{ $indexStats: {} }])*

Exemples

```
db.commandes.aggregate([
    { $match: { statut: "actif", montant: { $gt: 100 } } },
    { $project: { _id: 1, client: 1, date: 1, montant: 1 } },
    { $sort: { date: -1 } },
]);
```

```
db.utilisateurs.aggregate([
    { $match: { age: { $gte: 18 } } },
    { $group: { _id: "$pays", moyenneAge: { $avg: "$age" }, nb: { $sum: 1 } } },
    { $sort: { moyenneAge: -1 } },
]);
```

```
db.livres.aggregate([
    { $match: { auteur: "Agatha Christie" } },
    { $project: { titre: 1, annee: 1, genres: 1 } },
    { $sort: { annee: -1 } },
    { $limit: 5 },
    { $unwind: "$genres" },
]);
```

Exercices – Base de données supplies

- Comptez le nombre total de ventes par `storeLocation`.
- Calculez l'âge moyen des clients (`customer.age`).
- Calculez le chiffre d'affaires total pour chaque `storeLocation`. Vous devrez utiliser **`$unwind`** sur le tableau `items` et multiplier le prix par la quantité.
- Regroupez les ventes par niveau de satisfaction client (`customer.satisfaction`) et comptez le nombre de ventes pour chaque niveau.
- Calculez le total des ventes réalisées "In store" pour chaque `storeLocation`.

Exercices – Base de données analytics

- Calculez le nombre total de transactions pour chaque type de produit (`products`) dans la collection `accounts`.
- Calculez le montant total des transactions de type "buy" pour chaque `account_id` dans la collection `transactions`.
- Trouvez les 5 pays (`address.country`) avec le plus grand nombre de logements.
- Pour chaque commentaire de la collection `comments`, ajoutez les informations de l'utilisateur correspondant de la collection `users`.
- Calculez le score de satisfaction moyen (`review_scores.review_scores_rating`) pour les logements qui ont plus de 50 avis (`number_of_reviews`), groupés par `property_type`. Triez les résultats par score moyen décroissant.

Opérateur mapReduce - Présentation

- L’opérateur mapReduce est un outil d’agrégation avancée conçu pour traiter et transformer de très grands ensembles de données dans MongoDB.

- Il s’inspire du paradigme « MapReduce », largement utilisé dans le monde du traitement de données distribuées.

- MongoDB utilise des fonctions JavaScript personnalisées pour :
  - Mapper (map) chaque document d’une collection en une paire clé-valeur.
  - Réduire (reduce) les valeurs associées à une même clé pour produire un résultat global condensé.

Opérateur mapReduce - Fonctionnement

- La fonction map : Analyse chaque document et émet (via emit(key, value)) une ou plusieurs paires clé/valeur.

- La fonction reduce : Pour chaque clé, combine toutes les valeurs associées afin de réduire à un résultat unique (somme, moyenne, maximum, fusion, etc.).

- Les résultats peuvent être :
  - stockés dans une nouvelle collection ;
  - fusionnés ou réduits avec une collection existante ;
  - retournés directement (mode inline avec une limite de taille).

Syntaxe et exemple

```
db.<collection>.mapReduce(
   function() { emit(key, value); }, // map function
   function(key, values) { return ...; }, // reduce function
   {
      out: <sortie>,
      query: <filtre>,
      sort: <tri>,
      limit: <limite>
   }
)
```

```
db.posts.mapReduce(
   function() { emit(this.user_name, 1); },
   function(key, values) { return Array.sum(values); },
   { query: { status: "active" }, out: "post_totals" }
)
```

Exercices

- Base de données posts
  - Compter le nombre d'occurrences de chaque tag dans l'ensemble des articles de la collection `posts`.
- Base de données supplies
  - Calculez la quantité totale vendue pour chaque produit à travers toutes les ventes de la collection `sales`.
- Base de données airbnb
  - Calculez le prix moyen des logements pour chaque `property_type` dans la collection `listingsAndReviews`.

Limites des pipelines

- **Limites structurelles et syntaxiques**
  - **Longueur maximale du pipeline** : Un pipeline d'agrégation peut contenir jusqu'à 100 étapes. Cette limite vise à éviter des pipelines trop complexes ou difficiles à maintenir.
  - **Profondeur de document** : La profondeur maximale dans un document BSON transformé durant le pipeline est de 100 niveaux.

- **Limites de taille mémoire et de ressources**
  - **Limite mémoire par pipeline** : Par défaut, chaque étape du pipeline ne peut consommer plus de 100Mo de RAM. Si cette limite est dépassée (par ex. lors de gros group ou sort), MongoDB renvoie une erreur, sauf si l'option **allowDiskUse: true** est activée. Avec cette option, MongoDB utilise l'espace disque pour déborder la mémoire, mais cela peut impacter les performances.
  - **Limite de taille des documents issus du pipeline** : Un document émis à n'importe quelle étape du pipeline ne peut dépasser 16Mo, la taille limite fixée par le format BSON de MongoDB. Si un pipeline génère des documents dépassant cette taille (par exemple avec **$group** ou **$push** sur de très grands ensembles), une erreur est générée.

Limites des pipelines

- **Restrictions selon les opérateurs et les étapes**
  - **Position des étapes spéciales** : Certaines étapes, comme **$out** ou **$merge**, doivent impérativement être les dernières du pipeline.
  - **Priorité de certaines étapes** : Les étapes **$collStats** et **$indexStats** doivent être placées au début du pipeline.
  - **Usage restreint** : Certaines expressions ou opérateurs ne sont disponibles qu'à l'intérieur de certaines étapes (par exemple, les opérateurs d'accumulation comme **$sum** ou **$avg** ne sont valides que dans **$group**).

- **Limitations d'indexation et de plan d'exécution**
  - Seules les étapes **$match** et **$sort** placées en début de pipeline peuvent utiliser des index efficacement. Les filtres ou tris réalisés après transformation ou ajout de champs ne bénéficient plus des index.
  - Les pipelines complexes ou mal agencés peuvent entraîner des scans complets (COLLSCAN) et des chutes de performances.

Limites des pipelines

- **Limites liées à l'utilisation en cluster**
  - Certains stages comme **$lookup**, **$graphLookup**, **$out** ou **$merge** présentent des restrictions supplémentaires ou des modifications de comportement en environnement sharded cluster (fragmenté).
  - La cohérence ou la tolérance aux pannes peut entraîner des limitations lors de l'utilisation de pipelines complexes sur plusieurs shards.

- **Autres limitations et considérations**
  - **Concurrence et verrouillage** : Bien que la plupart des opérations d'agrégation soient non bloquantes, les étapes finales comme **$out** peuvent verrouiller la collection cible brièvement.
  - **Limite de temps d'exécution** : Un pipeline trop long peut être interrompu par les paramètres de timeout du serveur ou du driver client.
  - **Version du serveur** : Certains opérateurs ou capacités des pipelines ne sont disponibles qu'avec des versions récentes de MongoDB.

MongoDB - Schéma de données

Présentation

- Depuis la version 3.2, MongoDB permet la validation du schéma au niveau des collections :
  - Définition via $jsonSchema : on spécifie types, valeurs, regex, collections de valeurs autorisées.
  - Niveaux de validation : "strict" (toute insertion/modification non conforme est rejetée), "moderate" (seulement certaines opérations reçoivent un avertissement).
  - Application : tous les inserts et updates sont contrôlés ; option pour autoriser ou refuser les écritures non conformes.

```
{
   "validator": {
      "$jsonSchema": {
      "bsonType": "object",
      "required": ["title", "body"],
      "properties": {
         "title": { "bsonType": "string" },
         "body": { "bsonType": "string" },
         "tags": {
            "bsonType": ["string"],
            "description": "Doit être un tableau de chaînes"
         }
      }
   },
   "validationLevel": "strict"
}
```

Présentation

- Un schéma de données décrit la structure logique des données : comment les informations sont organisées, stockées, reliées, et sous quelles contraintes elles peuvent être manipulées dans une base de données.
- Il sert de plan directeur pour concevoir, comprendre et gérer des systèmes d'information, qu'il s'agisse de bases de données relationnelles, NoSQL ou d'autres modèles.
- Rôles principaux d'un schéma de données
  - **Définir la structure** : types de données, champs, relations entre entités (tables, documents, collections...).
  - **Imposer des contraintes** : unicité, non-nullité, typage, valeurs autorisées, etc.
  - **Faciliter la cohérence et l'intégrité** : s'assurer que les données restent fiables et utilisables, même en cas d'évolution du système.
  - **Accompagner l'évolution** : documenter les modifications structurelles au fil du temps pour garantir la compatibilité.

Spécification

- MongoDB propose deux grands types de modélisation :
  - **Schéma dynamique** : par défaut, aucun schéma n'est imposé (collections ouvertes).
  - **Schéma strict** : on peut définir des règles précises (typage, champs obligatoires, valeurs possibles) via le mécanisme de validation ($jsonSchema).

- Types de modélisation
  - **Documents imbriqués** (embedding) : recommandé pour des entités fortement liées (ex : commandes et adresse de livraison dans le même document).
  - **Documents référencés** (reference) : utile pour mutualiser des entités, ou en cas de relations many-to-many (ex : utilisateurs et groupes).

Intérêts des schémas dans MongoDB

- MongoDB se distingue par la flexibilité de son modèle de données : contrairement aux bases SQL traditionnelles, la structure des documents peut varier d'un enregistrement à l'autre dans une même collection.

- Cette approche permet :
  - Une adaptation rapide à l'évolution des besoins applicatifs : ajout/modification de champs sans migrations complexes.
  - Le stockage natif de données structurées, semi-structurées ou non structurées.
  - L'alignement naturel avec les objets métier issus de langages orientés objet (structure JSON/BSON).

Avantages des schémas de MongoDB

- **Schéma dynamique** : les collections n'imposent pas de schéma fixe ; grande souplesse pour les phases de prototypage ou d'évolution continue.
- **Performance** : accès rapide aux documents et forte adaptabilité à la répartition horizontale (sharding).
- **Modélisation adaptée** : permet d'imbriquer ou de référencer des documents pour refléter différents types de relations (one-to-one, one-to-many, many-to-many).
- **Scalabilité** : MongoDB gère nativement des volumes de données massifs et les pics de charge grâce à son architecture distribuée.

Inconvénients des schémas de MongoDB

- **Risque d'incohérence** : l'absence de schéma strict peut entraîner des erreurs, sauf si des règles de validation sont mises en place.
- **Redondance** : l'usage de modèles dénormalisés (données répliquées dans plusieurs documents) peut générer des incohérences ou alourdir la maintenance.
- **Limites techniques** : taille maximale de document (16Mo), profondeur d'imbrication, contraintes sur certaines opérations complexes comme les jointures ou transactions multi-documents.
- **Difficulté pour certains usages analytiques** : moins performant qu'un SGBDR pour les requêtes et reportings relationnels complexes.

Cas d'usage

- **Applications à données évolutives ou non structurées** : CMS, catalogues e-commerce, profils utilisateurs personnalisés.
- **Temps réel & IoT** : stockage de métriques ou logs dont la structure peut varier selon la source ou évoluer dans le temps.
- **Prototypage rapide/Agilité** : contextes où le modèle de données est amené à changer fréquemment (startups, projets innovants).
- **Applications nécessitant haute disponibilité et scalabilité horizontale** : gestion de contenu, systèmes de monitoring, analytics en flux continu.

Exercices

- L'objectif de cet exercice est de définir et d'appliquer un schéma de validation strict pour une nouvelle collection de ventes, en se basant sur la structure des données de la base `supplies`. Cela permettra de s'assurer que seules les données valides et conformes aux règles métier sont insérées.

- Créez une collection nommée `validated_sales` en utilisant la commande `db.createCollection()`. Dans cette commande, vous définirez un validateur `$jsonSchema` qui applique les contraintes d'intégrité suivantes définies dans les diapositives suivantes

Exercices

- **Champs Obligatoires** :
  - Les champs saleDate, items, storeLocation, customer, couponUsed, et purchaseMethod doivent être présents dans chaque document.

- **Contraintes sur les Types de Données** :
  - saleDate doit être de type date.
  - storeLocation doit être de type string.
  - couponUsed doit être de type bool.
  - purchaseMethod doit être une string.

Exercices

- **Contraintes sur le sous-document customer** :
  - customer doit être un objet obligatoire.
  - Il doit contenir les champs gender (string), age (int), email (string), et satisfaction (int).
  - L'âge (age) doit être un entier supérieur à 16.
  - La satisfaction (satisfaction) doit être un entier compris entre 1 et 5 inclusivement.
  - L'email (email) doit correspondre à une expression régulière de format email simple (par exemple, .*@.*\.com).

Exercices

- **Contraintes sur le Tableau items** :
  - items doit être un tableau et ne doit pas être vide (minItems: 1).
  - Chaque élément du tableau doit être un objet contenant :
  - name (string, obligatoire).
  - tags (un tableau de strings).
  - price (decimal, obligatoire).
  - quantity (int, obligatoire et supérieur à 0).

- **Contraintes d'Énumération** :
  - Le champ purchaseMethod ne peut accepter que deux valeurs : "Online" ou "In store".

Exercices

- Une fois la collection créée avec son schéma, essayez d'insérer plusieurs documents :
  - Un document parfaitement valide.
  - Un document où il manque le champ `customer`.
  - Un document avec un `age` de client de 15 ans.
  - Un document avec un tableau `items` vide.
  - Un document avec une valeur de `purchaseMethod` égale à "Phone".

MongoDB - Données géospatiales

Introduction

- Les données géospatiales représentent des informations liées à un positionnement géographique (latitude, longitude, altitude, etc.).

- Elles sont utilisées dans de nombreux domaines : cartographie, analyse urbaine, logistique, géolocalisation de services, etc.

- Dans le contexte des bases NoSQL comme MongoDB, la gestion et la recherche de données géospatiales sont facilitées par des formats adaptés et des opérateurs spécifiques.

Cas d'usage

- **Géolocalisation en temps réel** : Suivi de flotte de véhicules, localisation d'utilisateurs ou d'objets connectés.
- **Services de proximité** : Trouver les commerces, points d'intérêt ou événements proches.
- **Analyse territoriale** : Calculer des statistiques dans des zones urbaines ou administratives.
- **Gestion de réseaux** : Identifier des intersections, surveiller les infrastructures.

Représentation

- Format GeoJSON : MongoDB stocke les points, lignes, polygones et géométries complexes selon le format GeoJSON, un standard largement adopté pour les données géographiques.
- Types de géométries supportées :
  - Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, etc.
- Système de coordonnées : Utilisation du CRS EPSG:4326 (WGS84), où les coordonnées sont de la forme [longitude, latitude].

```
{
   "type": "Point",
   "coordinates": [2.3522, 48.8566] // [longitude, latitude]
}
```

Opérateurs

Opérateur	Description	Exemple d'utilisation
$near	Recherche de points proches d'une coordonnée donnée, triés par distance croissante	Lieux à proximité d'un utilisateur
$nearSphere	Version sphérique pour distances terrestres	Calcul de périmètre autour d'un point sur Terre
$geoWithin	Recherche d'objets totalement contenus dans une zone (cercle, polygone, etc.)	Points dans un quartier ou une zone administrative
$geoIntersects	Cherche les objets qui croisent une géométrie spécifiée	Routes traversant une région
$geometry	Représente la forme utilisée pour une requête géospatiale (supporte les types GeoJSON)	Définir un polygone cible pour la recherche

Exemple

```
db.places.find({
    location: {
        $near: {
            $geometry: {
                type: "Point",
                coordinates: [2.3522, 48.8566],
            },
            $maxDistance: 5000 /* in meters */,
        },
    },
});
```

Bonnes pratiques

- Toujours indexer les champs de type géospatial pour des performances optimales.
- Utiliser le format GeoJSON pour compatibilité et évolutivité.
- Prendre en compte la sphéricité de la Terre pour les grandes distances avec l'index 2dsphere.
- Définir précisément le système de coordonnées et la cohérence des données stockées.

Exercices

- Base de données geospatial
  - En utilisant l'opérateur **`$near`**, trouvez les 5 épaves (`shipwrecks`) les plus proches du point de coordonnées `[-74.0, 18.5]`.
  - En utilisant l'opérateur **`$geoIntersects`** avec un **`$geometry`** de type `Polygon`, trouvez toutes les épaves dont le `feature_type` est "Wrecks - Visible" et qui se trouvent à l'intérieur du polygone défini par les points suivants :
    - `[-75.0, 18.0]`
    - `[-73.0, 18.0]`
    - `[-74.0, 19.0]`
    - `[-75.0, 18.0]` (pour fermer le polygone)
- Base de données airbnb
  - En utilisant l'opérateur **`$geoWithin`** avec un **`$centerSphere`**, trouvez tous les logements (`listingsAndReviews`) situés dans un rayon de 2 km autour du point de coordonnées `[-73.9857, 40.7580]` (Times Square).

MongoDB - index

Présentation

- Un index en base de données est une structure de données destinée à accélérer les recherches de données sur une ou plusieurs colonnes, à la manière d'un index à la fin d'un livre.
  - Au lieu d'analyser chaque document ou chaque ligne lors d'une requête (ce qu'on appelle un scan complet), la base de données utilise l'index pour localiser rapidement les enregistrements pertinents.
- Utilité des Index
  - **Accélération des requêtes** : Les index réduisent drastiquement le temps de recherche des données.
  - **Optimisation des tris** : Ils permettent d'exécuter les opérations de tri plus efficacement.
  - **Renforcement des contraintes d'unicité** : Certains index (uniques) garantissent qu'aucune valeur dupliquée n'est présente sur le ou les champs indexés.
  - **Performance des jointures** : Dans les bases relationnelles, les index accélèrent les opérations de jointure entre tables.

Index MongoDB

- **Index par défaut** : Chaque collection possède un index unique automatique sur le champ _id.
- **Index secondaire** : On peut créer des index sur n'importe quel champ ou sous-champ du document.
- **Index simple et composé** : Comme les bases relationnelles, MongoDB permet des index simples (sur un champ) et composés (plusieurs champs).
- **Index multiclés** : Permettent d'indexer les champs contenant des tableaux : MongoDB crée une entrée d'index pour chaque élément du tableau.
- **Index géospatiaux** : Pour les requêtes sur des coordonnées géographiques (2d, 2dsphere).
- **Index texte** : Facilite la recherche plein texte sur des chaînes de caractères.
- **Index Hashed** : Utilisé pour le sharding (répartition des données).
- **Index partiel** : Indexe uniquement les documents qui correspondent à un filtre spécifique.
- **Index TTL** (Time-To-Live) : Permet la suppression automatique des documents après une certaine durée.

Précisions

- **Planification automatique** : MongoDB choisit l'index le plus pertinent au moment de la requête, mais il est possible d'analyser et d'optimiser ces choix via les outils d'analyse (explain()).
- **Absence de jointures** : Le besoin d'indexation est souvent centré sur la rapidité des recherches par clé et l'agrégation, moins sur les jointures complexes.
- **Compatibilité BSON** : Les index fonctionnent sur tout champ d'un document, même imbriqué, grâce au modèle BSON.
- **Maintenance** : Trop d'index dégrade les performances d'écriture ; il est recommandé de ne garder que les index nécessaires.

Index à champs unique

- Présentation
  - Cet index est créé sur un seul champ d'un document.
  - Il accélère les recherches, tris et filtres sur ce champ, et peut être construit en ordre croissant (1) ou décroissant (-1).
  - C'est le type d'index le plus simple et le plus couramment utilisé.
- Cas d'usage
  - Recherche rapide de documents par un identifiant, un nom, ou toute autre propriété individuelle.
  - Tri sur une colonne précise dans une collection importante.
  - Accélération des requêtes de filtrage portant sur un seul champ.
- Exemple
  - *db.users.createIndex({ firstName: 1 })*

Index composé

- Présentation
  - Un index composé est construit sur plusieurs champs d'un même document.
  - Il permet d'optimiser les requêtes faisant intervenir plusieurs critères à la fois.
  - L'ordre des champs dans l'index est crucial : la base trie d'abord sur le premier champ, puis le second, etc.
- Cas d'usage
  - Recherche multi-critères (par exemple, nom et date).
  - Tri combiné sur plusieurs colonnes.
  - Optimisation des agrégations complexes utilisant plusieurs champs.
- Exemple
  - *db.users.createIndex({ lastName: 1, age: -1 })*

Index multiclés

- Présentation
  - Cet index gère les champs qui sont des tableaux dans le document.
  - MongoDB indexe chaque élément du tableau ; il est donc possible d'interroger efficacement tous les éléments composant un même champ de type array.
- Cas d’usage
  - Recherche dans des champs comportant des tags, listes de mots-clés, adresses multiples, etc.
  - Filtrage sur les éléments d'un tableau dans le document.
- Exemple
  - *db.articles.createIndex({ tags: 1 })*

Index texte

- Présentation
  - Un index texte permet d'effectuer des recherches plein-texte sur un (ou plusieurs) champs de type chaîne.
  - Il prend en charge les recherches avancées, comme les expressions, la pertinence, et le « stemming » (reconnaissance des racines de mots).
- Cas d’usage
  - Recherche dans du contenu, par exemple, moteur de recherche pour des articles ou des commentaires.
  - Fonctionnalité d'autocomplétion textuelle.
- Exemple
  - *db.blogPosts.createIndex({ subject: "text", body: "text" })*

Index géospatial

- Présentation
  - Permet d’indexer des données de type coordonnées géographiques.
  - Prend en charge plusieurs types (2d, 2dsphere) pour répondre aux besoins de recherche spatiale et de localisation.
- Cas d’usage
  - Applications mobiles de géolocalisation, recherche de points d’intérêt proches, etc.
  - Calcul de distances, recherches par proximité.
- Exemple
  - *db.places.createIndex({ location: "2dsphere" })*

Index "Hashed"

- Présentation
  - Index basé sur le hachage de la valeur d’un champ, utilisé pour répartir uniformément les données entre plusieurs partitions (sharding).
- Cas d’usage
  - Répartition horizontale (sharding) des collections volumineuses pour équilibrer la charge.
- Exemple
  - *db.users.createIndex({ userId: "hashed" })*

Index TTL

- Présentation
  - Permet la suppression automatique des documents après une durée donnée.
  - L’index est défini sur un champ contenant une date ou un timestamp.
- Cas d’usage
  - Données temporaires (sessions utilisateur, logs, cache).
  - Nettoyage automatique de collections pour ne conserver que les données récentes.
- Exemple
  - Suppression automatique après 3600 s (1h) du champ createdAt dans la collection sessions.
  - *db.sessions.createIndex({ createdAt: 1 }, { expireAfterSeconds: 3600 });*

Index partiel

- Présentation
  - Index qui ne porte que sur une partie des documents, selon un critère précis.
  - Permet d'optimiser la taille de l'index pour des cas spécifiques.
- Cas d'usage
  - Indexation sélective selon l'état (active: true) ou la présence d'un champ.
  - Optimisation des performances si seuls certains sous-ensembles sont interrogés régulièrement.
- Exemple
  - Index partiel sur le champ status, uniquement pour les documents actifs :

```
db.orders.createIndex(
  { status: 1 },
  { partialFilterExpression: { status: "active" } }
);
```

Index unique

- Présentation
  - Ce type d'index impose l'unicité des valeurs sur un ou plusieurs champs.
  - Deux documents ne peuvent avoir la même valeur sur les champs indexés.
- Cas d'usage
  - Identifier de façon unique : email, numéro d'utilisateur, etc.
  - Empêcher les doublons lors de l'insertion de documents.
- Exemple
  - *db.users.createIndex({ email: 1 }, { unique: true })*

Wildcard Index

- **Présentation**
  - Les Wildcard index (ou « index joker ») sont une fonctionnalité avancée de MongoDB introduite à partir de la version 4.2, conçue pour indexer dynamiquement des champs inconnus ou multiples dans des documents ayant une structure potentiellement très flexible.
- **Principe**
  - Un Wildcard Index permet d'indexer tous les champs d'un document, ou tous les champs correspondant à un chemin donné, SANS avoir à connaître leur nom à l'avance.
  - Il s'utilise via la syntaxe joker "$**" dans la définition de l'index.
- **Exemple**: *db.collection.createIndex({ "$**": 1 })*
  - Cela crée un index couvrant tous les champs du document. Si de nouveaux champs apparaissent par la suite dans d'autres documents, ils seront également indexés automatiquement.

Wildcard Index – Cas d'usage

- Collections à schéma très dynamique, où les noms ou la nature des champs varient beaucoup entre documents (ex : applications IoT, logs, enregistrements hétérogènes...).

- Requêtes fréquemment portées sur des champs « dynamiques » ou qui ne sont pas connus lors de la conception initiale du schéma.

- Faciliter l'exploration ou l'interrogation sur des données semi-structurées sans multiplier les index spécifiques.

Wildcard – Fonctionnement et limitations

- **Sélectivité** : Un Wildcard Index indexe tous les champs possibles, ce qui peut générer un index assez volumineux en fonction de la diversité des contenus.

- **Projections/exclusions** : On peut restreindre les champs indexés en utilisant l'option wildcardProjection pour inclure/exclure certains sous-champs.

- **Performances** : Requêtes avec des prédicats sur des champs non anticipés bénéficient de cet index, mais attention : l'utilisation abusive de wildcard peut nuire à la taille et à la maintenance de l'index.

Index - Opérations

Opération / Commande	Description et utilité	Exemple d'utilisation
createIndex()	Crée un nouvel index sur un ou plusieurs champs.	*db.collection.createIndex({ champ: 1 })*
getIndex() / listIndex	Liste l'ensemble des index existants sur une collection.	*db.collection.getindex()*
dropIndex("name")	Supprime un index spécifique d'une collection, identifié par son nom ou sa définition.	*db.collection.dropIndex("index_name")*
dropIndex()	Supprime plusieurs index à la fois (sauf _id), ou tous (hors _id) en une seule commande.	*db.collection.dropIndex("*")*

Index - Opérations

Opération / Commande	Description et utilité	Exemple d'utilisation
totalIndexSize()	Retourne la taille totale occupée par tous les index d'une collection (en octets).	*db.collection.totalIndexSize()*
reIndex()	Reconstruit tous les index d’une collection (opération coûteuse, rarement nécessaire).	*db.collection.reIndex()*
compact	Défragmente la collection et reconstruit ses index (utilisé principalement pour la maintenance).	*db.collection.reIndex()*
validate()	Vérifie l’intégrité des données et des index sur une collection.	*db.collection.validate()*

Règles de bonnes pratiques

- Comprendre les modèles de requête
  - Analyser les requêtes les plus fréquentes de votre application afin de cibler les champs à indexer.
  - Créer des index uniquement sur les champs utilisés dans les filtres, tris, ou jointures.
- Choisir des index appropriés
  - Index à Champ Unique : Idéals pour des recherches simples sur un seul champ fréquemment interrogé.
  - Index Composés : À privilégier pour optimiser les requêtes multicritères ; ordonner les champs selon la règle ESR (Equality, Sort, Range) : d'abord l'égalité, puis le tri, et enfin la plage.
  - Index Multiclés : À utiliser pour interroger des éléments stockés dans des tableaux.
  - Index Texte et Géospatiaux : À adapter si vos requêtes portent sur du contenu textuel ou des données géographiques.

Règles de bonnes pratiques

- Limiter le nombre d’index
  - Évitet la sur-indexation : chaque index consomme de la mémoire, impacte le temps d’écriture et la place disque.
  - Supprimer les index inutilisés ou redondants (hors préfixe d’index composé).
  - En règle générale, ne pas dépasser 50 index par collection, sauf besoin spécifique.
- Optimiser la structure des index
  - Privilégier les index composés couvrant les champs de filtrage et de tri souvent utilisés ensemble.
  - Faire des "covered queries" (requêtes couvertes), c’est-à-dire des requêtes dont tous les champs nécessaires sont contenus dans l’index (y compris ceux à retourner), pour éviter de lire les documents sources.

Règles de bonnes pratiques

- Utiliser les index partiels et sélectifs
  - Les index partiels réduisent la taille de l'index en ne portant que sur certains sous-ensembles de documents : exploiter cette fonctionnalité pour des requêtes ciblées.
  - Favoriser les index sélectifs avec une forte cardinalité pour des performances accrues.
- Garder l'œil sur la performance
  - Surveiller l'utilisation réelle des index via les outils de profilage (**explain()**, **$indexStats()**)
    => Supprimer ceux qui ne sont jamais utilisés.
  - Cacher les index avant suppression définitive pour mesurer l'impact éventuel.
  - Éviter les expressions régulières non ancrées à gauche : elles neutralisent l'intérêt de l'indexation.

Règles de bonnes pratiques

- Adapter les index au sharding
  - Dans un cluster sharded, s'assurer que la clé de sharding figure dans l’index utilisé pour les requêtes critiques.

- Prendre en compte l’impact sur les écritures
  - Plus il y a d’index, plus la base consomme de ressources pour les opérations d’écriture, d’insertion, de modification ou de suppression.
  - Indexer avec parcimonie dans les applications à fort volume d’écritures.

Règles de bonnes pratiques

- Réviser et tester régulièrement
  - Tester les nouvelles configurations d'index avant déploiement en production.
  - Réviser périodiquement les index en fonction de l'évolution des requêtes et du schéma.
- Autres conseils pratiques
  - Éviter d'indexer les champs très volumineux ou faiblement sélectifs.
  - Privilégier la mise en place des index après un important chargement de données pour limiter la charge à l'écriture.
  - Utiliser les wildcard index dans les schémas dynamiques dont les champs varient d'un document à l'autre.

Exercices – Base de données airbnb

- Exécutez la requête `db.listingsAndReviews.find({ property_type: "Apartment" })` en utilisant `.explain("executionStats")` pour analyser ses performances.
  - Notez le temps d'exécution (`executionTimeMillis`) et le nombre de documents scannés (`totalDocsExamined`).

- Créez un **index simple** sur le champ `property_type`.

- Ré-exécutez la même requête avec `.explain("executionStats")`.

- Comparez les nouvelles statistiques avec les anciennes.
  - Vous devriez constater une réduction drastique du temps d'exécution et voir que `totalDocsExamined` est maintenant égal au nombre de documents retournés (`nReturned`).

Exercices – Base de donnée airbnb

- Analysez la requête suivante avec `.explain()` : `db.listingsAndReviews.find({ "address.country": "Spain" }).sort({ price: 1 })`.

- Créez un **index composé** sur les champs `address.country` et `price`. **Réfléchissez bien à l'ordre des champs dans l'index.** Puisque la requête filtre d'abord par pays (`equality`) puis trie par prix (`range`), l'index doit respecter cet ordre pour être optimal.

- Vérifiez avec `.explain()` que votre nouvel index est bien utilisé et qu'il évite une opération de tri en mémoire (`SORT` stage). Le plan d'exécution devrait montrer une lecture de l'index (`IXSCAN`) qui satisfait à la fois le filtre et le tri.

Exercices – Base de données mflix

- Créez un **index composé** sur les champs `title` et `year` de la collection `movies`.

- Exécutez la requête `db.movies.find({ year: { $gt: 2010 } }, { _id: 0, title: 1, year: 1 })`.

- Analysez le résultat de `.explain("executionStats")`.
  - Prouvez qu'il s'agit d'une **requête couverte** en vérifiant que `totalDocsExamined` est égal à `0`. Cela signifie que MongoDB a obtenu toutes les informations nécessaires directement depuis l'index, sans avoir à lire un seul document complet sur le disque.

Exercices – Base de données airbnb

- Créez un **index texte composé** sur les champs `name` et `summary` de la collection `listingsAndReviews`.
- Lors de la création de l'index, assignez un **poids (weight)** de `10` au champ `name` et un poids de `5` au champ `summary`. Cela rendra les correspondances dans le `name` plus pertinentes.
- Utilisez l'opérateur **`$text`** pour rechercher les logements contenant les termes "beautiful" et "cozy".
- Triez les résultats par pertinence en utilisant le score de texte (`{ score: { $meta: "textScore" } }`).
- Projetez les résultats pour n'afficher que le `name`, le `summary`, et le `score` de pertinence, afin de pouvoir constater l'impact de la pondération.

MongoDB - Performance

Présentation

- La performance dans le contexte des systèmes informatiques fait référence à la capacité d’un système à effectuer ses tâches de manière efficace, rapide et stable, tout en utilisant des ressources optimisées.

- C'est un critère essentiel pour garantir la satisfaction des utilisateurs, la scalabilité et la durabilité des applications, notamment lorsqu’elles manipulent de grands volumes de données ou répondent à de nombreuses requêtes simultanées.

Motivations

- **Expérience utilisateur** : des systèmes performants assurent une navigation fluide et des temps de réponse courts.
- **Optimisation des ressources** : de bonnes performances réduisent la consommation en CPU, mémoire, réseau...
- **Scalabilité** : un système performant peut mieux évoluer pour s'adapter à la croissance du trafic et des données.
- **Coûts** : améliorer la performance permet de réduire les besoins matériels et les dépenses d'infrastructure.

Caractéristiques

- **Temps de réponse** : rapidité avec laquelle le système traite une requête.
- **Débit** : nombre de requêtes traitées sur une période donnée.
- **Disponibilité** : capacité à être opérationnel et accessible en continu.
- **Consistance** : aptitude à fournir des résultats fiables et à jour.

Facteurs

- **Modélisation des données**
  - Utilisation de structures adaptées (documents, sous-documents, tableaux).
  - Adéquation avec le type d’accès attendu (lecture/écriture, requêtes fréquentes).

- **Indexation**
  - Création d'index pertinents pour accélérer les recherches et les agrégations.
  - Surveillance des index inutilisés pour éviter la surcharge mémoire.

Facteurs

- **Scalabilité horizontale**
  - Utilisation du sharding (partitionnement) pour distribuer les données sur plusieurs serveurs.

- **Gestion de la concurrence**
  - Accès concurrentiels optimisés grâce au modèle lock-free sur la plupart des opérations.

- **Requêtes et agrégations**
  - Analyse des plans d’exécution (explain) pour identifier les goulots d’étranglement.
  - Optimisation des pipelines d’agrégation.

Outils et bonnes pratiques

- **MongoDB Atlas et Ops Manager** : solutions embarquant des tableaux de bord pour surveiller l'utilisation CPU, la mémoire, le réseau, les requêtes lentes, etc.
- **Commandes d'analyse** : db.serverStatus(), db.currentOp(), et le profiler pour diagnostiquer les lenteurs.
- **Maintenance proactive** : ajustement des index, mise à jour des versions, configuration du cache WiredTiger.
- **Éviter les requêtes de scan complet de collection** (COLLSCAN) : privilégier les index adaptés.
- **Limiter la taille des documents** : au maximum 16 Mo, mais favoriser des documents compacts pour des accès plus rapides.
- **Partitionner judicieusement les collections volumineuses** : choix stratégique des clés de sharding.

Profiling - Présentation

- Le profiling a pour objectif d'analyser en détail les opérations effectuées par une base de données afin d'identifier les goulots d'étranglement, d'optimiser les performances et d'assurer la stabilité d'une application à grande échelle.

- Les motivations principales sont :
  - **Détection des requêtes lentes** : repérer les opérations qui dépassent un seuil de temps défini.
  - **Optimisation des performances** : ajuster la modélisation ou l'indexation en fonction des diagnostics collectés.
  - **Audit et traçabilité** : comprendre le comportement des applications lors de pics de charge ou d'incidents.
  - **Amélioration continue** : fournir une base factuelle pour les cycles d'optimisation itératifs.

Profiling - Principes

- Le profiling repose sur la collecte et l'analyse des détails sur l'exécution des requêtes :
  - Niveaux de profiling
    - Désactivé : aucune opération n'est enregistrée.
    - Profiling partiel : seules les opérations lentes (au-delà d'un seuil en ms) sont loguées.
    - Profiling total : toutes les opérations sont enregistrées, quel que soit leur temps d'exécution (à utiliser avec précaution, car générateur de surcharge).
  - Données collectées
    - Type d'opération (lecture, écriture, agrégation...)
    - Index utilisés ou absence d'index (COLLSCAN)
    - Détails du plan d'exécution
    - Temps d'exécution et ressources consommées

Profiling – Outils intégrés

- Profiler intégré (system.profile)
  - Enregistre les opérations selon le niveau configuré via db.setProfilingLevel().
  - Permet des recherches sur l'historique via la collection system.profile.
- Explain (explain())
  - Analyse le plan d'exécution d'une requête : indique si un index est utilisé, le nombre d'étapes, et les potentiels scans complets.
- Profiler graphique
  - MongoDB Compass ou Atlas (offrent des visualisations et la détection automatisée de requêtes lentes).
- Commandes système
  - db.currentOp(), db.serverStatus(), ou le Slow Query Log pour le suivi en temps réel.

Profiling - Exemples

```
// Activer le profiling pour les requêtes de plus de 50 ms
db.setProfilingLevel(1, 50)
// Visualiser les dernières opérations lentes
db.system.profile.find().sort({ ts: -1 }).limit(5)
```

```
db.collection.find({ field: value }).explain("executionStats")
```

Explain - Présentation

- La commande explain dans MongoDB est un outil essentiel pour comprendre comment les requêtes sont exécutées.

- Elle permet d'analyser en détail le plan d'exécution choisi pour une requête donnée, d'identifier les index utilisés, les opérations effectuées et d'obtenir des statistiques d'exécution précises.

- Cette analyse est indispensable pour diagnostiquer les problèmes de performance et optimiser les requêtes.

Explain - Utilisation

- Il existe plusieurs façons d'utiliser explain dans MongoDB :
  - En appelant directement db.collection.explain() puis en passant une requête telle que find, aggregate, etc.
  - En utilisant la méthode explain() sur un curseur : db.collection.find(...).explain().
  - Via la commande shell explicite : db.runCommand({ explain: <commande>, ... }).
- Verbosité:
  - **queryPlanner**: (Défaut) Affiche le plan d'exécution choisi et les plans rejetés, sans exécuter la requête.
  - **executionStats**: Fournit, en plus, des statistiques d'exécution réelles sur le plan retenu: nombre de documents examinés, temps d'exécution, etc.
  - **allPlansExecution**: Affiche les statistiques sur tous les plans candidats exécutés, y compris partiellement. Trés utile pour le diagnostic avancé.

Explain - Sortie

- La sortie renvoyée par explain est un document JSON qui intègre plusieurs sections :
  - **queryPlanner** : Détaille le plan d'exécution sélectionné (winningPlan), les index utilisés, la structure (les étapes, par ex. IXSCAN, COLLSCAN), la liste des plans rejetés.
  - **executionStats** : Affiche les métriques réelles (avec les niveaux executionStats ou allPlansExecution) :
    - *nReturned* : Nombre de documents retournés par la requête.
    - *totalKeysExamined* : Nombre de clés d'index parcourues.
    - *totalDocsExamined* : Nombre de documents réellement examinés.
    - *executionTimeMillis* : Temps total d'exécution en millisecondes.
  - **serverInfo** et **serverParameters** : Informations techniques sur le serveur MongoDB exécutant la commande.

Explain – Analyse d'une sortie

- MongoDB utilise un Query Planner qui génère différents plans et choisit le plus efficace en fonction des index disponibles et de la sélectivité de la requête :
  - Winning Plan (« plan gagnant ») : Le plan sélectionné par l'optimiseur. C'est ce plan qui sera exécuté.
  - Rejected Plans : Les autres plans candidats évalués mais non retenus.
  - Mécanisme multi-planner : MongoDB peut exécuter plusieurs plans en parallèle sur un sous-ensemble des données pour mesurer lequel est le plus efficace, puis il met en cache le plan retenu pour les requêtes similaires ultérieures.


```
{
  "queryPlanner": {
    "winningPlan": {
      "stage": "IXSCAN",
      "indexName": "status_1",
      "inputStage": {
        "stage": "FETCH"
      }
    },
  "rejectedPlans": [...]
  },
  "executionStats": {
    "nReturned": 42,
    "executionTimeMillis": 1,
    "totalKeysExamined": 42,
    "totalDocsExamined": 42
  }
}
```

Explain – Analyse d'une sortie

- Principales étapes d'un plan d'exécution
  - IDLE/COLLSCAN : Parcours complet de la collection (généralement à éviter, appelle la table entière).
  - IXSCAN : Parcours d'un index (beaucoup plus efficace).
  - FETCH : Récupération des documents correspondant aux clés trouvées avec l'index.
  - SORT, LIMIT, SKIP : Étapes additionnelles selon la requête.
  - AND/OR : Combinaison de plusieurs index ou expressions logiques.


```
{
  "queryPlanner": {
    "winningPlan": {
      "stage": "IXSCAN",
      "indexName": "status_1",
      "inputStage": {
        "stage": "FETCH"
      }
    },
  "rejectedPlans": [...]
  },
  "executionStats": {
    "nReturned": 42,
    "executionTimeMillis": 1,
    "totalKeysExamined": 42,
    "totalDocsExamined": 42
  }
}
```

Explain – Analyse d'une sortie

- Grâce à l'analyse des résultats d'explain, vous pouvez :
  - Vérifier l'utilisation des index pour éviter les COLLSCAN.
  - Mesurer le rendement d'une requête (nombre de documents parcourus vs. retournés).
  - Détecter les besoins de création ou de suppression d'index.
  - Comprendre les raisons de lenteur de certaines requêtes et agir en conséquence (modification de la requête, des index ou de la modélisation des données).


```
{
  "queryPlanner": {
    "winningPlan": {
      "stage": "IXSCAN",
      "indexName": "status_1",
      "inputStage": {
        "stage": "FETCH"
      }
    },
  "rejectedPlans": [...]
  },
  "executionStats": {
    "nReturned": 42,
    "executionTimeMillis": 1,
    "totalKeysExamined": 42,
    "totalDocsExamined": 42
  }
}
```

Optimisation des requêtes (Query Tuning)

- **Utilisation d'explain()** : Testez chaque requête avec explain() pour comprendre comment MongoDB l'exécute (index utilisé, nombre de documents parcourus, etc.). Misez sur les plans d'exécution qui minimisent totalDocsExamined et privilégient IXSCAN à COLLSCAN.

- **Projection limitée** : Retournez uniquement les champs nécessaires ; cela réduit l'utilisation du réseau et accélère la requête, surtout sur de grands documents.

- **Tri efficace** : Pour des requêtes triées, construisez vos index en tenant compte de l'ordre de tri ; sans cela, MongoDB doit charger tous les résultats en mémoire avant de les trier.

Optimisation des requêtes (Query Tuning)

- **Limitation des résultats** (*limit, skip*) : Utilisez limit pour réduire la charge en limitant le nombre de documents renvoyés par la requête.

- **Reformulation des requêtes** : Privilégiez des filtres sur des champs indexés ; évitez les expressions régulières non ancrées et les recherches sur des champs non indexés.

- **Gestion des index temporaires** : Pour diagnostiquer ou forcer l'utilisation d'un index particulier, utilisez *$hint*.

Optimisation des pipelines

- **Filtrage précoce avec $match**
  - Placez les étapes *$match* aussi tôt que possible. Cela réduit le nombre de documents traités par la suite et limite l'usage mémoire et CPU.
  - Si possible, tirez parti des index sur les champs utilisés par *$match*, car MongoDB peut exploiter directement les index lors des premières étapes du pipeline.

- **Projection anticipée avec $project / $unset**
  - Utilisez un *$project* ou *$unse*t dès le début pour ne garder que les champs nécessaires.
  - Moins de champs à manipuler dans les étapes suivantes = moins de charge processeur et réseau.

Optimisation des pipelines

- **Utilisation efficace des index**
  - Créez des index sur les champs fréquemment utilisés dans *$match* et *$sort.*
  - Un index bien aligné peut accélérer considérablement le pipeline, notamment devant les jointures (*$lookup*), les triages (*$sort*) ou les regroupements (*$group*).

- **Ordre stratégique des étapes**
  - Effectuez toujours le filtrage (*$match*) et la pagination (*$limit, $skip*) avant les opérations lourdes, comme *$group, $lookup*, et *$sort* (si possible).
  - Mettez *$limit* juste après *$sort* dès qu'il est utilisé pour ne pas trier inutilement un gros ensemble de documents.
  - Minimisez l'usage du *$skip* sur de gros volumes – préférez la pagination basée sur un identifiant (par exemple via *$match* sur un _id supérieur à une valeur donnée).

Optimisation des pipelines

- **Regroupement et agrégation judicieux**
  - N'utilisez *$group* qu'une fois les documents strictement nécessaires sélectionnés : cette étape est coûteuse en ressources.
  - Agrégez plusieurs champs dans un même *$group* plutôt que de multiplier les étapes.

- **Réduction et fusion des étapes**
  - Combinez les étapes là où c'est possible (*$match* + *$project*, fusion de plusieurs *$match* proches) pour que le plan d'exécution soit plus court et efficace.
  - MongoDB dispose d'un optimiseur de pipeline qui réorganise parfois automatiquement les étapes, mais il est toujours préférable de concevoir un pipeline optimal dès le départ.

Optimisation des pipelines

- **Optimisation des jointures et opérations lourdes**
  - Placez les *$lookup* et *$unwind* après le filtrage maximal.
  - Assurez-vous que la collection secondaire pour *$lookup* est indexée sur le champ utilisé pour la jointure.

- **Utilisation de l'option allowDiskUse**
  - Pour les pipelines dépassant la limite mémoire (100Mo), activez *allowDiskUse: true* : cela permet à MongoDB d'utiliser le disque pour les opérations intermédiaires volumineuses.

Optimisation des pipelines - Exemple

```
db.collection.aggregate([
    { $match: { statut: "valide" } }, // Filtre précoce
    { $project: { utilisateur: 1, score: 1 } }, // Projection anticipée
    { $sort: { score: -1 } }, // Tri sur champ indexé
    { $limit: 10 }, // Limitation rapide
    {
        $lookup: {
            from: "users",
            localField: "utilisateur",
            foreignField: "_id",
            as: "infos",
        },
    },
]);
```

Exercices

- Optimiser les deux requêtes suivantes (airbnb & mflix)

```
db.listingsAndReviews.aggregate([
    {
        $sort: { price: -1 }
    },
    {
        $match: { "address.market": "New York" }
    },
    {
    $limit: 10
    },
    {
        $project: {
        _id: 0,
        name: 1,
        price: 1,
        "address.market": 1
        }
    }
])
```

```
db.comments.aggregate([
    {
        $group: {
            _id: "$movie_id",
            commentCount: { $sum: 1 }
        }
    },
    {
        $lookup: {
            from: "movies",
            localField: "_id",
            foreignField: "_id",
            as: "movie_details"
        }
    },
    {
        $unwind: "$movie_details"
    },
    {
        $match: { "movie_details.year": { $gt: 2000 } }
    },
    {
        $project: {
            _id: 0,
            title: "$movie_details.title",
            commentCount: 1
        }
    }
])
```

Bonnes pratiques

- Testez vos index sur un jeu de données représentatif avant la mise en production.
- Revue régulière des plans d’exécution pour s’assurer que les index sont toujours pertinents avec l’évolution fonctionnelle.
- Ne dupliquez pas inutilement les index et vérifiez si certains index apportent effectivement un gain.

Synthèse

- Concevez vos index en fonction des habitudes réelles d'accès aux données.
- Privilégiez des index composés adaptés à vos filtres, tris et plages de données.
- Surveillez, testez et ajustez régulièrement vos index et requêtes à l'aide d'outils tels que explain(), les logs, et les outils graphiques comme Compass ou Atlas.
- Gardez un équilibre entre performance en lecture et coût en écriture/mémoire.

MongoDB - Sharding

Présentation

- Le sharding est une technique d'architecture de bases de données utilisée principalement pour améliorer la capacité de stockage et les performances lors de la gestion de grands volumes de données.
- Dans un contexte NoSQL, notamment avec MongoDB, le sharding consiste à répartir les données sur plusieurs serveurs ou clusters, appelés shards, chaque shard stockant une portion des données totales.
- Cette approche s'oppose à une architecture dite « monolithique » où toutes les données résident sur un seul serveur. Avec le sharding, chaque shard gère une partie spécifique des enregistrements, selon un découpage strict et déterminé (par exemple, basé sur une clé ou un champ de la donnée).

Motivations

- **Scalabilité horizontale (scaling out):** Permet d'ajouter facilement de nouvelles machines pour augmenter la capacité de stockage ou le nombre de requêtes gérées, contrairement à la scalabilité verticale (scaling up) qui implique de remplacer un serveur par un autre plus puissant.

- **Traitement de très gros volumes de données:** Utile lorsque la quantité de données dépasse la capacité de stockage, de traitement ou les limites de performance d'un seul serveur. On retrouve ce besoin dans les applications web avec des millions d'utilisateurs, des analyses big data ou des systèmes SaaS.

Motivations

- **Réduction de la charge sur chaque serveur:** Les opérations sont réparties entre plusieurs shards, ce qui évite qu'un seul serveur ne devienne un point de congestion ou de panne.

- **Haute disponibilité et tolérance aux pannes:** En associant sharding et réplication, on améliore la résilience : la perte d'un shard n'entraîne pas l'indisponibilité du système global.

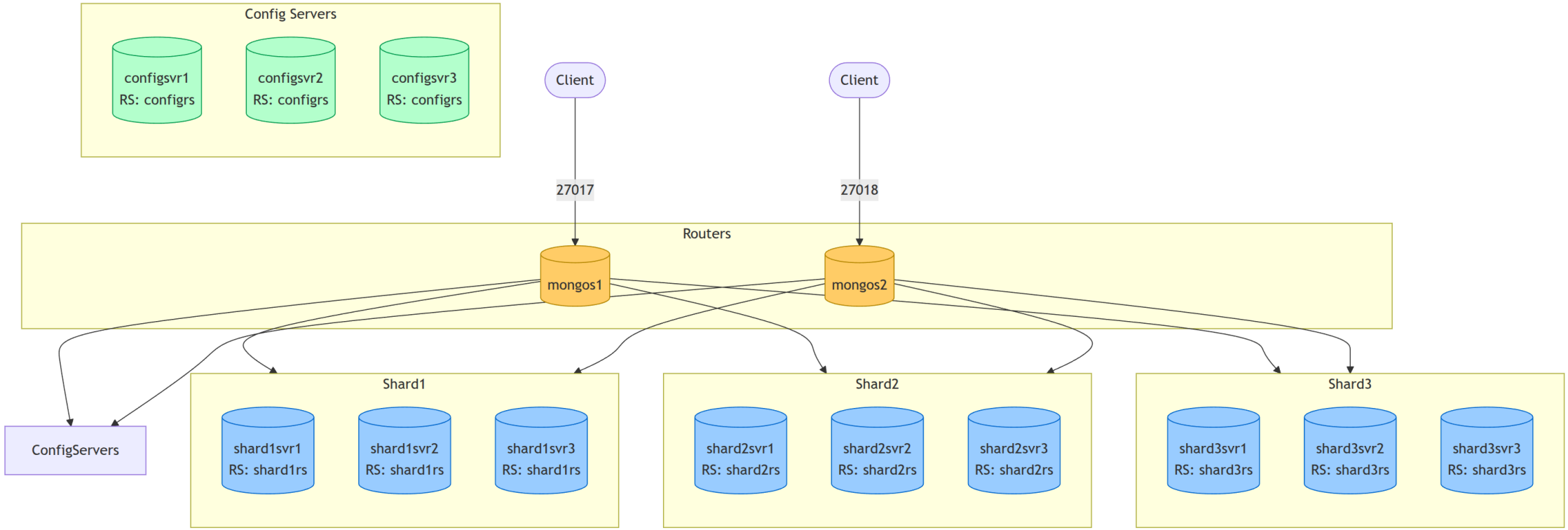
- **Optimisation des coûts:** Il est souvent plus économique de disposer de plusieurs serveurs standards que d'investir dans un unique serveur très puissant.

Architecture

- MongoDB intègre nativement le sharding. Il repose sur trois composants principaux :
  - **Shards** : serveurs contenant les données partitionnées.
  - **Config Servers** : stockent la configuration du sharding et la métadonnée de distribution des données.
  - **Query Routers** (mongos) : servent d'interface entre l'application et les shards, en acheminant les requêtes vers le(s) bon(s) shard(s) selon la clé de sharding.

- Cette architecture permet à MongoDB de supporter de très fortes charges et de traiter des ensembles de données répartis sur plusieurs machines, tout en présentant une interface unifiée pour le développeur ou l'utilisateur.

Architecture - Illustration



Architecture - Shards

- **Présentation** : Les shards sont les serveurs ou ensembles de serveurs (clusters) qui stockent les données partitionnées de la base, chaque shard contenant une fraction des données globales.
- **Rôle** :
  - Héberger physiquement une partie des données.
  - Gérer les opérations de lecture/écriture pour les données dont il est responsable.
- **Responsabilités** :
  - Stocker et répliquer les données selon leur clé de sharding.
  - Exécuter les requêtes transmises par le Query Router, pouvant concerner tout ou partie des documents stockés localement.
  - Assurer la disponibilité, la performance et la sécurité des données sur chaque shard.

Architecture – Config Servers

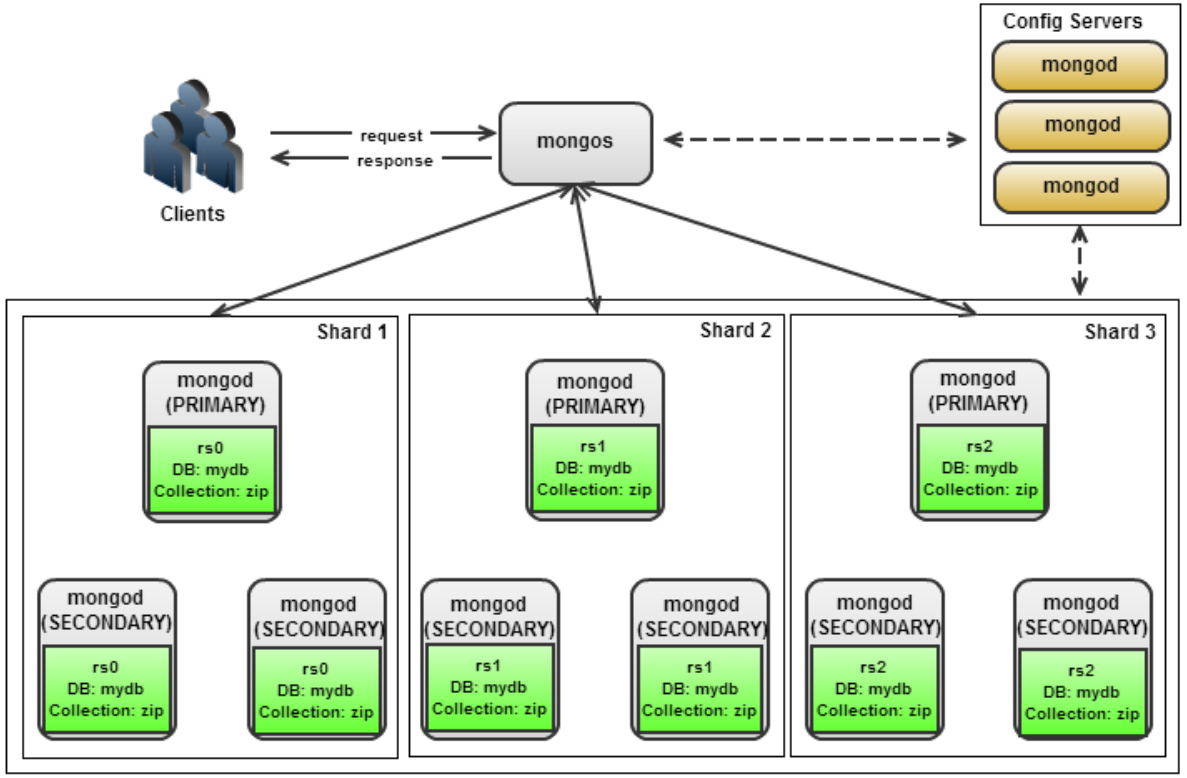
- **Présentation** : Les Config Servers sont des serveurs spécifiques qui stockent la configuration du cluster, en particulier la map (métadonnées) de la distribution des données sur les shards.
- **Rôle** :
  - Maintenir une vue centralisée de la répartition des données.
  - Servir de référentiel pour la topologie et l'état du sharding.
- **Responsabilités** :
  - Enregistrer la structure des chunks (segments de données) et leur mapping vers les shards.
  - Assurer la cohérence et la synchronisation des métadonnées lors des opérations de migration de données entre shards.
  - Sécuriser l'intégrité des informations de partitionnement.

Architecture – Query Routers

- **Présentation** : Les Query Routers (processus mongos) servent d'interface entre les applications clientes et le cluster sharded.
- **Rôle** :
  - Acheminer les requêtes des applications vers le ou les shards appropriés.
  - Agréger les résultats et répondre à l'application cliente.
- **Responsabilités** :
  - Déterminer, grâce aux métadonnées disponibles via les Config Servers, quels shards sont concernés par une requête.
  - Diviser les requêtes complexes si nécessaire, les transmettre aux shards impliqués et combiner les résultats de façon transparente pour le client.
  - Gérer la logique de routage, sans stocker de données, permettant ainsi de scaler horizontalement les points d'accès au cluster.

Sharding - Illustration

- Un shard correspond à un sous-ensemble des données globales.
- Chaque shard est lui-même un replica set MongoDB complet : il contient un nœud PRIMARY (écritures/principales lectures) et plusieurs nœuds SECONDARY (réplicas pour la tolérance aux pannes ou la montée en charge lecture).
- Dans cette illustration, les shards répartissent la collection ‘zip’ de la base mydb en plusieurs morceaux/chunks selon la shard key. Ainsi, l’échelle peut augmenter linéairement.



Sharding par plage

- **Principe** : Les données sont réparties selon des intervalles (plages) de valeurs du shard key (clé de sharding). Par exemple, tous les utilisateurs ayant un identifiant compris entre 1 et 100,000 peuvent être stockés sur un shard, ceux de 100,001 à 200,000 sur un autre, etc.
- **Avantages** :
  - Très efficace pour les requêtes de type « range queries » (sur des intervalles de valeurs).
  - Recommandé pour les données naturellement ordonnées, comme des dates ou des identifiants numériques séquentiels.
- **Inconvénients** : Risque de déséquilibre (hotspot) si la distribution des valeurs de la clé n'est pas uniforme : certains shards peuvent être surchargés tandis que d'autres restent sous-utilisés.

Sharding par hachage

- **Principe** : Une fonction de hachage est appliquée à la valeur du shard key pour déterminer le shard de chaque donnée. Le résultat du hachage répartit les données plus uniformément parmi les shards.
- **Avantages** :
  - Permet une distribution très équilibrée des données, évitant la saturation d'un shard unique.
  - Pratique lorsque la clé de sharding ne présente pas de structure ordonnée ou quand un équilibrage automatique est recherché.
- **Inconvénients** :
  - Requêtes par plages inefficaces, car les valeurs consécutives du champ clé sont réparties sur de multiples shards, nécessitant de consulter chacun d'eux pour couvrir une plage de valeurs.
  - Moins adapté si les requêtes concernent souvent des intervalles.

Sharding par zone géographique

- **Principe** : Les données sont réparties en fonction de critères régionaux, de zones ou de règles métier. Par exemple, tous les enregistrements d'un pays ou d'une région sont affectés à un shard particulier (ce qu'on appelle aussi « zoned sharding » ou « geo-sharding »).
- **Avantages** :
  - Optimise les performances pour les applications nécessitant la localisation géographique des données ou des règles de résidence des données.
  - Réduit la latence en rapprochant les données des utilisateurs finaux selon la géographie.
- **Inconvénients** :
  - Peut amener un déséquilibre si certaines zones sont fortement sollicitées par rapport à d'autres.

Sharding par entités

- **Principe** : Les données fortement liées sont placées ensemble sur un même shard (par exemple, toutes les informations d'un même utilisateur).
- **Avantages** :
  - Réduit les besoins de requêtes traversant plusieurs shards.
  - Optimise les performances pour certaines requêtes applicatives complexes.
- **Inconvénients** :
  - Peut nécessiter une connaissance approfondie du modèle métier ;
  - Déséquilibre possible selon la taille ou l'activité des entités.

Sharding par annuaire

- **Principe** : Un annuaire (table de mapping) conserve la correspondance entre chaque entrée et le shard hébergeant cette donnée. Ce type est rare dans MongoDB mais commun dans certains NoSQL.

- **Avantages** :
  - Très flexible pour des schémas complexes.

- **Inconvénients** :
  - Ajoute une couche de complexité et de gestion à l'infrastructure.

Shard vs ReplicaSet

- Un shard n'est généralement pas un simple serveur, mais également un replica set, constitué d'un nœud primaire (qui reçoit les écritures) et d'un ou plusieurs nœuds secondaires (qui répliquent automatiquement les données du primaire).
- Cela signifie que la gestion de la réplication (synchronisation des données, basculement automatique en cas de panne, redondance) est de la responsabilité de chaque shard pris individuellement.
- Lors d'un déploiement en production, il est recommandé de toujours configurer chaque shard en replica set plutôt qu'en simple instance MongoDB. Cela garantit que la défaillance d'un serveur ne rend pas le shard (et donc le cluster MongoDB) indisponible.

Principales commandes

- Activer le sharding sur une base de données : *sh.enableSharding("nom_de_la_base")*
- Lister les bases shardées : *sh.getShardMap()*
- Sharder une collection par plage: *sh.shardCollection("base.collection", { cle: 1 })*
- Sharder une collection par hash: *sh.shardCollection("base.collection", { cle: "hashed" })*
- Créer une zone et associer un shard: *sh.addShardToZone("nom_du_shard", "nom_zone")*
- Supprimer un shard d'une zone: *sh.removeShardFromZone("nom_du_shard", "nom_zone")*
- Vérifier le statut du sharding: *sh.status()*
- Ajouter un nouveau shard au cluster : *sh.addShard("adresse_serveur")*
- Supprimer un shard: *sh.removeShard("nom_du_shard")*
- Activer /Désactiver l'équilibrage de charge: *sh.startBalancer() / sh.stopBalancer()*
- Afficher l'état du balancer: *sh.getBalancerState()*

Exercices

- Votre entreprise collecte des données de ventes en temps réel depuis des milliers de magasins. La collection `sales` (basée sur `supplies.json`) grandit de plusieurs millions de documents par jour. Pour gérer cette croissance, vous décidez de mettre en place le sharding. Un collègue suggère d'utiliser le champ `saleDate` comme clé de sharding (shard key).
- Questions:
  - **Analyse du problème :** Pourquoi utiliser un champ qui augmente de manière monotone comme `saleDate` (ou `_id` par défaut) est-il une très mauvaise idée pour la clé de sharding dans ce scénario ? Décrivez le phénomène de **"hot shard"** (ou "shard chaude") qui en résulterait.
  - **Impact sur les performances :** Quel serait l'impact de ce phénomène sur les performances d'écriture de votre cluster ?
  - **Proposition de solution :** Proposez une meilleure stratégie de clé de sharding pour distribuer plus uniformément les écritures. Pensez à deux approches :
  - **Partitionnement par hachage (Hashed Sharding) :** Sur quel champ l'appliqueriez-vous et pourquoi ?
  - **Partitionnement par plages (Ranged Sharding) :** Proposez une **clé de sharding composée** qui éviterait le problème de la "hot shard".

Exercices

- Votre application de type Airbnb (`listingsAndReviews.json`) est devenue mondiale et la collection unique est devenue trop lente. La grande majorité des requêtes des utilisateurs consiste à rechercher des logements **à l'intérieur d'un même pays**. Par exemple, "trouver des appartements à Spain" est beaucoup plus fréquent que de faire une recherche mondiale.
  - Votre objectif est de partitionner (shard) la collection `listingsAndReviews` pour optimiser ce cas d'usage très fréquent.
- Questions
  - **Choix de la Clé :** Quelle serait la clé de sharding idéale pour ce scénario ? Justifiez votre choix en expliquant comment elle répond au besoin principal de l'application.
  - **Avantages :** En quoi votre choix de clé permet-il d'éviter les requêtes de type **"scatter-gather"** (où le routeur de requêtes, `mongos`, doit interroger tous les shards du cluster) pour la majorité des recherches ?
  - **Inconvénients potentiels :** Y a-t-il des inconvénients ou des scénarios moins optimaux avec la clé que vous avez choisie ? (Par exemple, que se passerait-il si un pays avait beaucoup plus de logements que tous les autres ?). Comment pourriez-vous potentiellement atténuer ce problème ?

Bonnes pratiques

- **Choix pertinent de la clé de sharding**: Sélectionner une clé de sharding (shard key) qui assure une répartition uniforme des données et de la charge entre les shards. La clé doit aussi bien correspondre aux requêtes les plus fréquentes afin d'éviter les hotspots et d'assurer de bonnes performances.
- **Indexation optimisée**: Il est obligatoire de créer un index sur le champ choisi comme clé de sharding afin d'accélérer les opérations de recherche et de routage des écritures. Penser aussi à intégrer la clé de sharding dans les index composés pour bénéficier de requêtes couvertes, ce qui accélère significativement les accès en lecture dans un cluster sharded.
- **Planification de la montée en charge**: Prendre en compte la croissance future => démarrer avec suffisamment de shards pour absorber le trafic attendu et prévois d'en ajouter au fil de la montée en charge. Anticipe l'évolution du volume de données et la capacité réseau et matérielle nécessaire.

Bonnes pratiques

- **Utilisation de matériel dédié:** Affecter à chaque shard son propre serveur (physique ou virtuel). L'usage de machines dédiées évite la concurrence des ressources et garantit de meilleures performances pour l'ensemble du cluster.
- **Déploiement des Config Servers et des Shards en Replica Set**: Pour garantir la haute disponibilité et la tolérance aux pannes, configurer chaque shard ainsi que les config servers en Replica Set (idéalement au moins trois membres par ensemble).
- **Équilibrage et migration des chunks**: Surveiller régulièrement la répartition des chunks (blocs de données) et s'assurer que le processus de balancer de MongoDB fonctionne correctement. Il doit migrer automatiquement les chunks pour maintenir l'équilibre de la charge entre les shards.

Bonnes pratiques

- **Tests et simulations avant production**: Valider la répartition des données, la performance des requêtes et le bon fonctionnement du routage dans un environnement de pré-production pour éviter les problèmes lors du passage en production.
- **Surveillance et optimisation continue**: Mettre en place une surveillance régulière => observer l'utilisation CPU/mémoire, la distribution des données, les métriques de réplication et d'équilibrage, la latence des requêtes, etc. Réajuster la clé de sharding ou l'architecture si des déséquilibres ou saturations apparaissent.
- **Adoption de mises à jour atomiques ou idempotentes**: Dans un environnement sharded, il est important de minimiser les opérations multi-shards, qui sont plus coûteuses. Structurer les mises à jour pour qu'elles touchent de préférence un seul shard lorsque c'est possible.

Limites

- **Choix du shard key définitif et critique**: Le choix de la clé de sharding est irrémédiable : une fois la collection shardée, il n'est pas possible, en production, de changer la shard key sans opérations lourdes telles que le resharding (migration complète des données), ce qui peut entraîner des interruptions et coûts importants.
- **Complexité accrue de l'architecture**: Un cluster sharded nécessite plusieurs composants (shards, config servers, mongos) à maintenir, surveiller et mettre à jour. Cette complexité impacte la maintenance et augmente le risque d'erreurs ou d'incidents.
- **Limitations sur les index uniques**: Les index uniques ne sont garantis que si la clé de sharding fait partie de l'index (au moins en préfixe). Sinon, MongoDB ne peut pas garantir l'unicité sur l'ensemble du cluster, mais seulement à l'intérieur de chaque shard.

Limites

- **Opérations multi-shards coûteuses**: Les requêtes ou mises à jour qui n'incluent pas la clé de sharding (ou son préfixe dans le cas d'une clé composée) doivent être envoyées à tous les shards (scatter/gather), ce qui dégrade fortement les performances et la latence.
- **Impossibilité de « désharder » une collection**: Une fois qu'une collection est shardée, il n'existe pas de mécanisme natif pour « désharder » : la migration vers une collection non sharded demande des manipulations lourdes et, souvent, une indisponibilité temporaire.
- **Limites sur la taille avant sharding et la migration**: Il est recommandé de sharder une collection avant qu'elle n'atteigne une taille importante (ex : 256 Go), car la création du sharding après coup induit des migrations massives de données susceptibles de surcharger le cluster et d'allonger la durée de la maintenance.

Limites

- **Limites sur la modification du shard key des documents**: Le champ de shard key d'un document n'est pas modifiable après insertion. Pour « changer » la clé, il faut supprimer puis réinsérer le document avec la nouvelle valeur.

- **Certaines opérations ou API partiellement compatibles**: Par exemple, certaines opérations d'agrégation ($lookup, $graphLookup) imposent que la seconde collection soit non sharded selon les versions, et certaines limitations existent avec les transactions distribuées ou l'utilisation d'opérations d'écriture atomiques sur plusieurs shards.

Limites

- **Limites de capacité d'un shard individuel**: Chaque shard (généralement un replica set) est lui-même limité, par exemple à 4 To de données sur MongoDB Atlas. Le sharding permet de dépasser ces limites uniquement en ajoutant de nouveaux shards.

- **Risques de déséquilibre** (hotspots): Si la shard key est mal conçue, certains shards peuvent recevoir la majorité du trafic ou des données (hotspots), annulant les bénéfices de la répartition de charge.

APIs

Présentation

- Selon vos besoins (développement applicatif, automatisation d'administration, intégration légère), vous avez le choix entre :
  - Les drivers officiels disponibles dans la plupart des langages pour de la manipulation fine ;
  - Les APIs REST/HTTP pour Atlas (Data API, Admin API) ;
  - Des frameworks ODM/ORM pour accélérer le développement.
    Le tout s'inscrit dans une démarche d'ouverture et de flexibilité, la plupart des usages programmatiques étant couverts, que vous travailliez en local ou dans le cloud.

Librairies

Langages supportés	Nom du driver	Utilisations typiques
Python	PyMongo	Data science, scripts, applications backend
Node.js	MongoDB Node.js Driver	Web apps, services, serverless
Java	MongoDB Java Driver	Systèmes d'entreprise, frameworks Java
C#, .NET	MongoDB .NET/C# Driver	Applications Microsoft, SaaS
Go	MongoDB Go Driver	Cloud natif, microservices
Ruby	MongoDB Ruby Driver	Rails apps, scripting
PHP, C, C++, Rust...	Drivers ou ORM communautaires/officiels	Divers

Frameworks

- Beaucoup de frameworks proposent leur propre couche d'abstraction, facilitant le mapping objet/documents :
  - Mongoose (Node.js/JS)
  - MongoEngine (Python)
    - Cf. Exemple dans le dossier python
  - Spring Data MongoDB (Java/Spring)

Spring-Boot MongoDB

- Spring Data MongoDB : l'API de référence
- Spring Data MongoDB est la couche d'intégration officielle qui simplifie l'utilisation de MongoDB au sein d'une application Spring :
  - Dépendance : Dans le fichier pom.xml ou build.gradle
  - Configuration: Modification du fichier application.properties
- 2 APIs d'accès: Mongo Repository et Mongo Template

```
org.springframework.boot spring-boot-starter-data-mongodb
```

```
spring.data.mongodb.uri=mongodb://localhost:27017/nom_de_la_base
spring.data.mongodb.auto-index-creation=true
```

Spring-Boot MongoDB - Exercices

- Prenez le temps d'étudier l'exemple dans le dossier src
- Vous devez ajouter une fonctionnalité où un utilisateur peut "liker" un logement, ce qui incrémente un compteur de likes et ajoute l'ID de l'utilisateur à une liste de "fans".
  - **Modifiez votre `ListingAndReview.java`** pour ajouter les champs : `private int likes;` et `private List<String> fans;`.
  - **Créez une méthode `addLike(String listingId, String userId)`** qui utilise `mongoTemplate.updateFirst()`.
  - Dans cette méthode, construisez un objet `Query` pour cibler le bon logement par son ID.
  - Construisez un objet `Update` qui utilise :
    - `$inc` pour incrémenter le champ `likes` de 1.
    - `$push` pour ajouter le `userId` au tableau `fans`.
  - Exposez cette fonctionnalité via le contrôleur `ListingAndReviewController`.

Conclusion

Conclusion

- NoSQL s’est imposé comme une réponse aux limites du modèle relationnel traditionnel face aux besoins de scalabilité, de flexibilité et de performance des architectures modernes. Cette famille regroupe une diversité de modèles (clé-valeur, document, colonne, graphe), chacun répondant à des cas d’usage spécifiques, mais aucun n’étant universel.
- MongoDB, figure de proue des bases NoSQL orientées document, se distingue par sa souplesse de modélisation, la puissance de ses requêtes et son adaptation native aux architectures distribuées. Sa popularité tient à sa capacité à accompagner des projets de toutes tailles, de la startup à la grande entreprise, tout en soutenant l’innovation rapide et l’évolution continue des besoins métiers.

Conclusion

- La connaissance approfondie des concepts clés (schéma flexible/dynamique, indexation avancée, opérations CRUD, agrégation, sharding, réplication, gestion géospatiale, profiling...) est indispensable pour tirer le meilleur parti de MongoDB, sans sacrifier la cohérence des données, la performance ou la maintenabilité du système.
- Le choix du type de base et du modèle de données doit toujours être guidé par l'analyse précise des besoins métier, des contraintes d'évolution, de performance et de volume. La maîtrise des compromis (CAP vs ACID, schéma vs flexibilité, performance en lecture/écriture, scalabilité horizontale vs complexité) s'avère donc déterminante.
- Enfin, la mise en œuvre des bonnes pratiques — de la conception du schéma à l'optimisation des requêtes, en passant par le diagnostic de performance et le choix stratégique du sharding — conditionne la robustesse et la pérennité des solutions basées sur MongoDB.