

Chapitre BD NoSQL:

Chapitre 1: Introduction aux BD NoSQL.

Bases de données NoSQL:

Définition:

BD non-relatielles et largement distribuées.
collection de données éparses en plusieurs places.

NoSQL (Not Only SQL)

Rôle: une analyse et organisation rapide des données
volumes très grande volume.

Exploration des données (augmentation exponentielle)

générée par les utilisateurs et leurs applications:

Apparition des
BD NoSQL.

Atouts et caractéristiques:

• Données structurées (tables, colonnes).

• Données Non Structurées (texte, Image).

Atouts

Evolutivité

Tolérance aux fautes

Disponibilité

Caractéristique

Node de données
sans schéma

Architecture
distribuée

NoSQL orientée clé/valeur:

• Stockage simple et direct d'un tableau associatif (hashTable).

• Accès via clé primaire \Rightarrow Interrogation univiseuse pour la clé

• Ex: Table Etudiant

Id	Nom	Age	Adresse
12	John	22	Paris
13	Ali	null	Bruxelles
14	Aladin	null	null

\Rightarrow Clé : Etud-12.

• Valeur: Nom -> John // Age -> 22 //
Adresse -> Paris.

• Rôle: Sauvegarder des données sans définir de schéma.

• Stocker des données en rapport avec un id.

• Opérations: Put: donner une valeur à un id. Delete: effacer la clé et sa valeur.

• Get: récupérer une valeur d'une clé

• Exemples: DynamoDB (Amazon), Azure Table Storage, Redis ...

NoSQL orienté document:

• Système clé-valeur: • Valeur \rightarrow structure connue par le système \Rightarrow peut la parcourir.

• Document \rightarrow un objet \rightarrow 1 ou plusieurs champs \rightarrow chaque champ contient un type

• Avantage: récupérer, via un simple req., un ensemble d'informations \leftarrow structurene
pas de jointure de BD Relationnelle \leftarrow de manière
hierarchique.

• Exemples: MongoDB, Couch DB, Raven DB.

Id	Nom	Age	Adresse
12	John	22	Paris
13	Ali	null	Bruxelles
14	Aladin	null	null

\Rightarrow { {Nom: "John", Age: 22, Adresse: "Paris"}, {Nom: "Ali", Adresse: "Bruxelles"}, {Nom: "Aladin"} }

NoSQL orienté document-colonnes:

• Evolution de la BD clé/valeur.

• Réserve de la SGBDR, mais: • un nombre de colonnes dynamiques.

different d'un enregistrement sur un autre (pour des utilisateurs NDB)

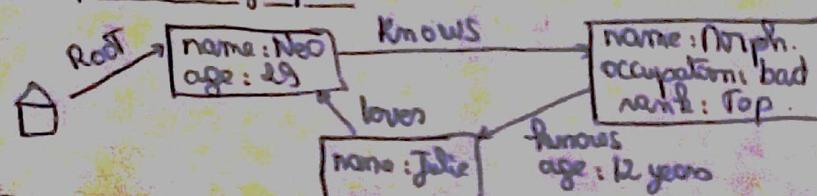
Exemples: Hbase (Hadoop), Cassandra, BigTable ...

ID	Nom	Age	Adresse	
12	John	22	Paris	⇒
13	Alw	Null	Bruxelles	13 Nom/Alw Age/22 Adresse/Bruxelles
14	Alain	Null	Null	14 Nom/Alain

NoSQL orienté Graphes :

- Pas de stockage d'ensemble de données → Stockage de Relations
- S'appuie sur les notions de **nœuds** et **relations**.
- Difficulté de stockage d'un graphe sur plusieurs serveurs (Graphe répartie sur plusieurs serveurs ⇒ problème de performance même pour un simple calcul de chemin).

Exemple d'un graphe:



Adapté aux traitements de Réseaux Sociaux.

Comparaison:

Modèle	Performance	Évolutivité	Flexibilité	Complexité	Fonctionnalité
Cle/Value	•	•	•	•	Variété
Orienté colonnes	•	•	•	•	Métrique
Orienté document	•	•	•	•	variable
Orienté graphe	•	•	•	•	théorie de Graphe

NoSQL vs Relationnel:

Propriétés ACID:

- BD R → supportent ACID (permettent le respect des contraintes d'intégrité)
 - **A → Atomicité:** Toute transaction (requete SELECT par exemple) doit être complète (s'exécute entièrement sans erreur) ou ne s'exécute pas.
 - **C → Consistance / Cohérence:** BD doit être toujours dans état cohérent entre deux transactions. Cohérence assurée par les contraintes d'intégrité.
 - **I → Isolation:** modifications d'une transaction ne sont pas visibles par les autres tant qu'elles n'a pas été validée.
 - **D → Durabilité:** validées → données sont permanentes jusqu'à prochaine modification.

Point:

- Données (plus en plus volumineuses + non régulières)
- Besoin d'un système évolutif
- BD distribuée sur le réseau (évolution horizontale)

} Difficulté de appliquer ACID sur BD NoSQL.

Théorème CAP: Voir cours

Cours BD NoSQL

Introduction

BD NoSQL \Rightarrow collection de données réparties sur plusieurs places.

Ces dernières années, on a eu une explosion de données (une quantité de données massives) de plusieurs formats (texte, image, ...).



MongoDB est orienté document (Non Relationnelle).

Propriétés ACID:

BD Relationnelle \Rightarrow ACID

A \rightarrow Atomicité : la résultat d'une requête SELECT doit être complet : s'exécute entièrement ou ne s'exécute pas.

C \rightarrow ~~consist~~ cohérence : ^{de cohérence}

I \rightarrow Isolation : je dois valider pour passer à une autre chose.

D \rightarrow durabilité : la modification doivent être enregistrée.

BD non relationnelle \Rightarrow données non organisée \Rightarrow C'est difficile de appliquer les propriétés ACID \Rightarrow Appliquer CAP

C \rightarrow Consistance : tous les nœuds doivent avoir la même info

A \rightarrow Availability : Disponibilité \rightarrow je dois avoir mes données disponibles pour chaque requête \rightarrow une réponse même en cas de panne.

P \rightarrow Partage et faire Partitionnement \rightarrow mes données doivent être réparties sur plusieurs supports.

Dans le non Relationnelle, on ne peut pas avoir tous les propriétés dans chaque outil : MongoDB \rightarrow CP | Cassandra \rightarrow AP. \Rightarrow théorème CAP.

\Rightarrow BD Relationnelle \Rightarrow CA.

Notion de Scalabilité :

Saturation des données \Rightarrow Non tenir en charge verticale = Scalabilité verticale
Augmenter taille de matériau.

Scalabilité horizontale \Rightarrow distribution de données sur plusieurs sites.

\hookrightarrow utile en BD non relationnelle.

NB: Pour les BD relationnelles, on utilise la scalabilité verticale car les données sont reliées entre eux.

Base de données orientée documents

(MongoDB)

MongoDB → divers documents sous format JSON (Binang JSON)
 → Non relationnelle → distribution.

BDR (SQL) vs MongoDB

BDR	MongoDB
BD	BD
Table	Collection
Ligne	document
Colonnes	Champs

BDRMongoDB

Perso	name	phone	Age
J	NathR	Agné	28.

Perso

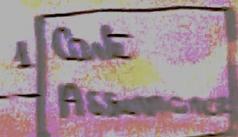
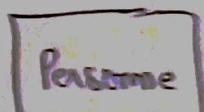
{ name : "NathR",
 phone : "Agné", }

Représentation Jointure (NoSQLisation)

No décomposition

Imbrication:

Mettre les données du tableau ~~compte Assurance~~ dans la table ~~Personne~~
 Y Personne → en seul fichier JSON.

Référence

Ex:

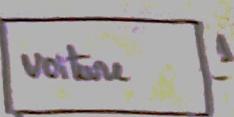
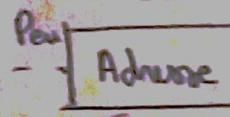


Personne
 { name : ---,
 phone : --- }

Compte Assurance
 ID : ---
 { }

Imbrication

Personne
 { name : ---,
 phone : ---,
 adresse : [] }

Référence

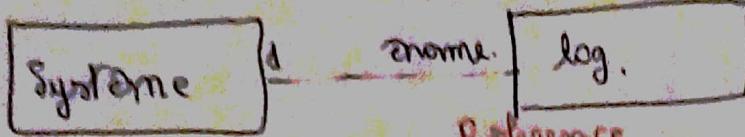
Voiture
 { model : ---,
 Composants : [] }
 id : ---
 nroque : ---

Composante
 id : ---
 nroque : ---

{ }

{ name : ---,
 ville : --- },
 { name : ---,
 ville : --- }] }

Scanned with CamScanner



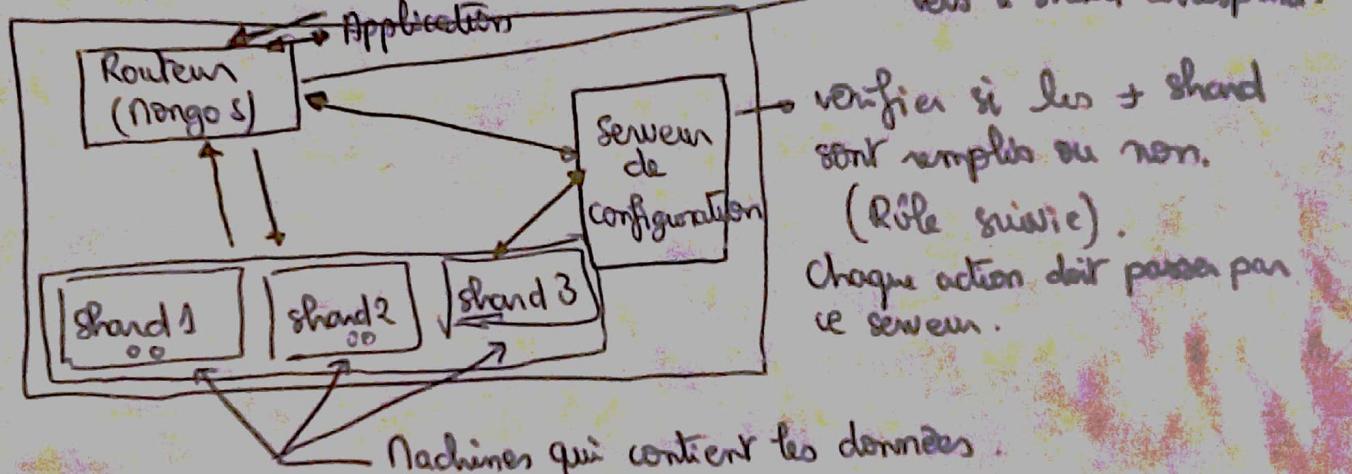
Sharding & RéPLICATION

Sharding: (Partition / Segmentation).

Répartir les données sur + machines (Scalabilité).

Cluster (ensemble de machines) MongoDB.

Application



Faire plusieurs shard (répartir les données) permet de bien aller les requêtes vers le shard contenant les données.

On peut faire un sharding selon un clé (Id, age...) → Clé de sharding.

Ex: Personne

```
{
  nom: "...",
  prénom: "...",
  age: ...
}
```

sharding
selon l'age

⇒ sh 1

sh 2

sh 3

↓
Sarah
Noahmed
30

la requête va directement vers le shard correspond au critère de l'âge.

↳ Minimiser le temps de réponse → Minimiser les champs parcourus.

Le sharding ne garantit pas la tolérance ou en cas de panne, on peut perdre partiellement des données.

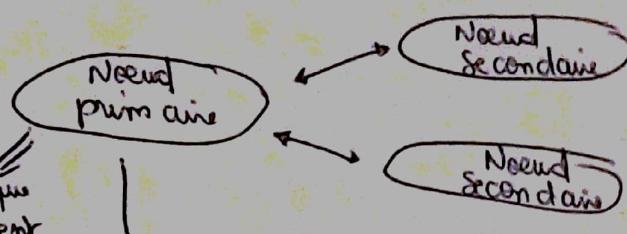
Replication: (Replica set → un ensemble de réplique).

Repliquer → dupliquer → faire des copies de mes données dans des machines (backups).

⇒ Chaque shard est composé de plusieurs instances.

Architecture d'un replica set:

En cas de dommage, lorsque ce nœud revient → revient en tant que nœud secondaire -



on peut écrire que dans le nœud primaire.

Lorsque le nœud primaire est endommagé, l'un des nœuds secondaires devient nœud primaire (mécanisme d'élection).

↳ D'où la présence d'un nœud d'arbitre qui ne peut pas devenir nœud primaire et ne contient pas de données → IP qui revient que lors de l'élection en tant qu'arbitre. (Et: en cas de conflit).

Mise en place d'un cluster MongoDB:

- 1) Crédation des shards (sh1, sh2, sh3)
 - 2) Crédation des répliques (rep1, rep2, rep3).
 - 3) Crédation des serveurs de configurations (cfg1, cfg2, cfg3)
- ⇒ démarrage - cluster - mongo.bat.

II) Manipulation des shards du cluster.

Faire: Atelier 2.

Concentrer sur les ports

Etat "purge / chaining" d'un shard:

l'équilibre (balance) commence à déplacer les segments (chunks) hors du shard manqué (qui est en train de se vider).

1) Crédation du sharded cluster:

- ⇒ Demander à mongo de connaître les shards qu'il va communiquer entre eux.

2) Test sharding:

- c) Le port 37023 correspond à mongo ⇒ Connexion sur mongo.
- ⇒ db.wers ⇒ Crédation de la collection = wers et sa manipulation
- ⇒ sh. shardCollection ⇒ permet de décomposer les données selon l'id (notre id de sharding).

Remarques

- MongoDB est CP : Consistency → par les sharding replication.
Persistence → par le sharding.

MongoDB est disponible (Available) mais pas hautement disponible car on doit attendre pour avoir notre résultat.

à faire Réviser chapitre 3.

Cours BD NoSQL

Chapitre 3: MongoDB Operations CRUD

Port par défaut du MongoDB → 27017

Mongo Shell:

Démarrage: cmd → mongo d (Mongo daemon: ouvrir une env de mongoBD)

NB: On laisse mongo d s'exécuter en arrière plan.

Connexion: cmd → mongo

Base de données courante: Pour savoir dans quel base on est → db

NB: Par défaut, MongoDB est connecté sur la BD test

Liste BDs: Pour avoir toutes les bases trouvant dans MongoDB par défaut → show dbs

Sélection d'une BD: use <name BD> (Ex: use admin)

Affichage Collections: show collections

Démarrage	Connexion	Arrêt	Sélection BD	Supprimer BD	Le shell est
mongod	mongo	shutdown	use <name BD>	db.dropDatabase() (dropDB(true))	son shell et son côté + la commande
BD courante	list BDs	> use admin > db. show dbs	Affichage Collections	use Admin X use admin ✓ Show dbs X	use Admin X use admin ✓ Show dbs X

Types de données

Type	Exemple	Type	Exemple
int / double	{a: 1}	Array	{d: [1,2,3]}
boolean	{b: true}	Date	{e: ISODate("2023-12-15")}
String	{c: "Hello"}	Object	{g: {"a": 1, "b": 2}}

CRUD:

Syntaxe (Forme Générale): db. <collection>. <méthode> ()

Create	db. collection. insert(<document>) ⇒ La méthode insert va créer la collection "collection" et va l'alimenter par les données du document.
Read	db. collection. find (<query>, <projection>) db. collection. findOne (<query>, <projection>)

Update	<code>db.collection.update(<query>, <update>, <options>)</code>
Delete	<code>db.collection.remove(<query>, <justOne>)</code>

Insertion:

document BSON "Binary JSON"

- Ex: `db.people.insert({ nom: 'Smith', age: 30, profession: 'ingénieur' })`
- la collection 'people' n'existe pas → on va la créer et l'alimenter en même temps.
- pour vérifier que l'insertion a été effectuée et l'afficher : `db.people.find()`
- `find` retourne : { -id: ObjectId ("5f897b25"), nom: 'Smith', age: 30, ... } document JSON

NB: on peut définir un Id → identifiant unique de type ObjectId → clé primaire de la collection que celle de MongoDB :
 On ajoute un champ "id" → n'est pas modifiable → ne pas le préciser.
 mais on va avoir les deux "id" et "-id".
 → MongoDB se charge de le créer.

- On peut avoir une redondance du même document JSON plusieurs fois → la même ligne s'insère plusieurs fois avec des identifiants "-id" différents.
- On peut préciser le type d'un champ : `db.people.insert({ nom: 'Smith', age: NumberInt(30) })`

Lecture:

`find()`:

- `db.people.find()` → BD Relationnel → Select → From people ;
- `find()` → permet de rechercher et afficher des documents.
- `db.people.find({ nom: 'Smith' }, { nom, true, age, true })` → champs qu'on veut afficher.
 - document → critère de recherche
 - BD Relationnelle : WHERE nom = "Smith"
 - document → renvoie tous les champs
 - BDR : Select nom, age
 - En son absence → retourne tous les champs
 - true / 1 → champ affiché
 - false / 0 → champ non affiché
 - si on peut ne pas mentionner le champ

NB: l'ID est toujours retourné, il ne sera pas retourné que lorsque on ajoute :

-id : false ou -id : 0.

• `db.people.find({ nom: 'Smith', -id: 0, nom: 0, age: 1 })`

→ on peut pas avoir deux champs true et false → MongoDB un erreur Bad Value
 ⇒ les champs qu'on veut pas les afficher → on les mentionne pas.

• `db.people.find({ nom: 'Smith' }, { -id: 0, profession: 0 })` → Par défaut, il va afficher les autres champs.

• `db.people.find({ }, { -id: 0, nom: 1, age: 1 })` → Affichage nom et age de tous les documents.

• `db.people.find({ }, { BDR: 1, nom: 1, age: 1 })` → Select nom, age From people ;

• it. → iteration (passer aux résultats suivants → les résultats sont paginés)

• `db.people.find().pretty()` → ajouter un beau affichage.

`findOne()`:

• `db.people.findOne()` → renvoi un document quelconque de la collection.

• `db.people.findOne({ }, { nom: 1, age: 1 })` → renvoie maximum 1 document seulement le nom et l'âge et l'Id.

Utilisation d'opérateurs:

Opérateur	Syntaxe	Exemple
\$gt, \$gt=, \$lt, \$lt=	\$gt : Greater than > \$lt : Lower than <	\$gt : Creation a > equal \$lt : Creation a <= equal
\$exists	test sur l'existence d'un champ	profession : { \$exists : true }
\$type	test sur le type d'un champ	name : { \$type : "string" }
\$regex	Recherche de pattern dans une chaîne (contient une chaîne, commence / termine par une chaîne)	name : { \$regex : ".*" }
\$in	Sur logique	age : [{ name : { \$regex : ".*" } }, age : { \$exists : true }]
\$and	Et logique	\$and : [{ name : { \$regex : ".*" } }, age : { \$exists : true }]

NB : - db.people.find({age: { \$lt : 32, \$gt : 35 }}) → va retourner rien car age < 32 et age >= 35 → Pas d'intersection → ce qui veut dire que il n'y a pas de résultat.

- db.people.find({age : { \$lt : 32 }, age.\$gt : 35 }) → affiche les documents dont l'âge est inférieur à 32 et supérieur à 35 (Intersection) → il y a des résultats.

On donne plusieurs conditions pour le filtre champ → shell va construire une phrase objet correspondant à la 1^{re} condition age <= 32, puis évaluer et ajouter quand à elle la 2^{re} condition (age >= 35)

Exemples : • contient le champ age : { age : { \$contains : 3 } }

• logique du champ nom est Smith : { nom : { \$logique : 2 } }

• Nom qui commence par h : { nom : { \$regex : "h" } }

• nom qui termine par h : from : { name : "hs" }

• nom qui commence par s : { nom : { \$regex : "s" } }

• mots saisis dont le nom est Smith Bob : db.people.find({ name : { \$in : ["Smith", "Bob"] } })

BD Relational : db.users.find({ name : { \$in : ["smith", "bob"] } })
Select + filter users WHERE name in ["smith", "bob"]

• mots saisis dont leurs pronoms sont naming : db.people.

db.people.find({ favorite : { \$all : ["naming", "public"] } })

• limit() : pour récupérer des n premiers résultats uniquement.

Ex : db.people.find().limit(1) → retourne le 1^{er} document.

• sort() : trier les résultats

autre en sens : .sort({ name : 1 })

autre direction : .sort({ { name : -1 } })

.count() : retourne le nombre de documents satisfaisant la requête (Ex db.people.find().count())

Lecture d'un sous-document :

db.article.insert({ title : "article", author : { person : "Hamed", name : "Salah" } })

un sous-document.

La partie dans le nom de Hamed est Salah : db.article.find({ "author.name" : "Salah" })

Récupération des documents :

Récupération globale :

- db.people.update({nom: 'smith'}, {nom: 'Alain', salaire: 5000})
 Selectionner le document dont le nom est smith. ↗ Modification à effectuer sur le document dont nom est smith.
- Cette mise à jour va Supprimer le document sélectionné et va insérer un nouveau document avec les valeurs passées en 2^{ème} paramètre.
- La mise à jour nous permet d'ajouter des nouveaux champs.
- Mise à jour sélective:
- db.people.update({nom: 'jean'}, {\$set: {salaire: 5000}}) document
 - db.people.update({nom: 'jean'}, {\$unset: {salaire: 1}}) on va pas supprimer le type, on va ajouter un à 3 documents ont le nom est jean à ce document champ salaire.
- ↳ Il va changer que le 1^{er} document
 ⇒ \$set n'effectue pas des mises à jour multiples → mise à jour sur 1 seul document.
- Indiquer valeur d'un champ → ajouter \$inc sur le champ (doit être un entier).
- Ex: db.people.update({nom: 'Jean'}, {\$inc: {age: 1}}) → champ age va s'ajouter [1] (21 → 22)
- Supprimer un champ d'un document → ajouter \$unset sur le champ à supprimer.
 - Ex: db.people.update({nom: 'Jean'}, {\$unset: {salaire: 1}}) ↗ true : champ salaire à supprimer.
- Mise à jour de plusieurs documents:
- db.people.update({nom: 'jean'}, {\$set: {salaire: 5000}}, {multi: true}).
 → Pour mettre à jour plusieurs documents : ajouter à \$set, un 3^{ème} argument {multi: true}.
- Mise à jour d'un champ de type tableau:
- Définition d'un champ de type tableau (le 1^{er} indice d'un tableau est 0):
 db.people.insert({nom: 'smith', age: 34, activité: ['sport', 'musique']})
 - On va modifier 'musique' par 'camping' → sélectionner un champ dans le tableau par son indice : db.people.update({nom: 'smith'}, {\$set: {activité.1: "camping"}})
 ↗ accéder à un élément d'un tableau
- \$push : ajouter un élément à un tableau:
 Ex: db.people.update({nom: 'smith'}, {\$push: {activité: 'musique'}}).
- \$pop : retirer un élément d'un tableau selon position de l'élément (premier, dernier).
 Ex: db.people.update({nom: 'smith'}, {\$pop: {activité: 1}}).
- \$pull : retirer un élément d'un tableau selon son nom:
 Ex: db.people.update({nom: 'smith'}, {\$pull: {activité: 'camping'}}).
- Suppression des documents et des collections:
- Suppression des documents : db....remove() → remove s'applique sur Tous les documents.
- Ex: supprimer le document dont nom : 'jean' : db.people.remove({nom: 'jean'})
 ⇒ Il va supprimer tous les documents dont le nom est 'jean'.
- Suppression de collections : db....drop()
- Transfer de données: ↗ nom de la collection à supprimer. Ex: db.article.drop()
- Import: Importer des données à partir de fichier JSON / CSV.
 mongoimport --db nombase --collection nomcollection --type json --file "chemin_fichier.json"
- Export: Exporter les données vers un fichier JSON.
 mongoexport --db nombase --collection nomcollection --out nom_fichier_sortie
- NB: mongoimport et mongoexport → exécutés dans cmd et non mongo shell.
- Sauvegarde et récupération:

Sauvegarde:

- Sauvegarder une partie / la totalité de la base dans un dossier.
- mongo dump --db base à sauvegarder --out nom_dossier_de_sauvegarde
 - mongo dump --help : options de mongo dump.

Récupération: récupérer les données à partir fichier BSON.

mongorestore --db test --collection people backup/test/people.bson

NB: JSON → format d'entrée de données.

BSON → format de sortie de mongo db.

--db : base dans laquelle on va mettre la collection.

--collection : collection à récupérer.

backup/test/people.bson : fichier à partir de lequel on va importer la collection ;

Chapitre 4: MongoDB Operations & aggregations

Aggregation

Agrégation → grouping (groupement) → combinaison de plusieurs méthodes et des résultats de plusieurs documents pour retourner un seul résultat.

Netnode aggregate()

- forme générale: db. collection-aggregate ([{...}])

- forme générale :
 - Le framework d'agrégation de MongoDB → basé sur le concept de pipelines de traitement de données : documents entrent dans une pipeline à plusieurs étapes/stages

Documents en entrée (Ex: collection) → Stage 1 → Stage 2 → ... → Stage n → Documents en sortie (BSON)

L'ensemble 1 est nommé transforme les documents

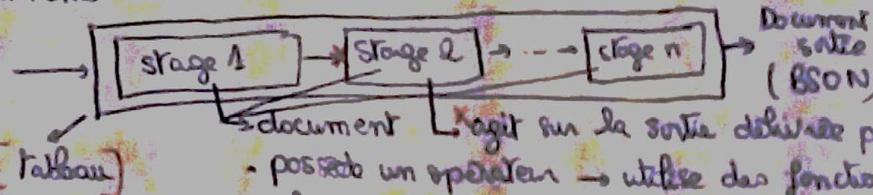
Document → Stage 1 → Stage 2 → ... → Stage n → Document

L'agir sur la sortie délivrée par stage précédent.

 - possède un opérateur → utilise des fonctions
 - plupart peuvent être utilisés plusieurs fois

(Mais ne peut pas être utilisé qu'à fois)

manipulation d'une comparaison



- possède un opérateur → utilise des fonctions → manipulation d'une
- plupart peuvent être utilisés plusieurs fois → composition

Exemples de stages

\$project: Fait transmet les documents selon les champs demandés (sélectionner des champs) à l'étape suivante de pipeline.

\$match : filtrer les documents selon des conditions (équivalent à WHERE en SQL)

\$ set : Trouver les documents d'entrée et les renvoyer au pipeline.

limit: limite le nombre de documents passés au stage suivant.

skip : Ignorer un nombre de documents et parser les documents restants.

group: équivalent du GROUP BY en SQL ↗ créer des groupements
appliquer des fonctions de groupe.

Unwind: entraîne un élément d'un champ de type tableau.

`$lookup`: récupérer un document référencé par le document courant (une sorte de jointure).
`$out`: (la dernière étape) écrit les documents résultants de lagrégation dans une collection.

\$ project

Manipulation de chaînes de caractères

Ces manipulations n'affectent pas les valeurs des champs et un affichage peut être valide que lors de requête.

Exemple de fonction	Explication	Exemples.
<code>\$t lower / \$t upper</code>	Afficher un champ en minuscule ou en minuscule	db. Items aggregate ([{"\$project": {"-id:0, author: "\$author", "Title": "Basket"}, "title": 1}]).
<code>\$concat</code>	Concaténation des chaînes (même des champs : <code>\$title</code> →) dans une variable utilisée lors de l'affichage	db. Items aggregate ([{"\$project": {"author:1, title:1, description: ["concat": ["\$title", "-", "description"]]}]}])
<code>\$split</code>	<u>Separateur</u> : séparer le contenu du champ avec un opérateur et retourne un tableau (chaque él. → ms)	db. Items aggregate ([{"\$project": {"-id:0, description: ["\$split": ["\$description", " "]}]}])
<code>\$substr</code>	Substitution en une sous chaîne : <u>Welcome</u> → Wel <code>substr(0,2)</code>	db. Items aggregate ([{"\$project": {"-id:0, description: ["\$substr": ["\$description", 0, 4]}]}])
<code>\$strLenBytes</code>	longueur d'une chaîne (champ: <code>\$title</code>) → nb caractères (<code>length</code>)	db. Items aggregate ([{"\$project": {"-id:0, title:1, length: ["\$strLenBytes": "\$title"]}}])

Manipulation des valeurs monétaires

Exemples de fonctions	Explication	Exemples
<code>\$add</code>	Incrementer la valeur du champ à un nombre donné en affichage seulement.	db.bikes.aggregate([{\$project:{bikes:[{\$add:[{"\$literal":1}, {"\$ref": "bikes", "\$field": "nb"}, {"\$const": 3}]}]}}, {bikes:1}])
<code>\$ceil</code>	plus petit entier supérieur égale nombre donné. Ex: <code>ceil(2.001) → 3 / ceil(1) → 1 / ceil(.35) → 1</code>	db.bikes.aggregate([{\$project:{bikes:[{\$ceil:{\$ref:"bikes", "\$field": "nb"}]}]}}, {bikes:1}])

- **\$slice**: lorsque un nombre & un entier ou une chaîne
Ex: db.livres.aggregate([{"\$project": {"_id:0, likes: {\$slice: ["\$likes", 3]}}})

Manipulation des tableaux:

Ex de fonctions | Explication

\$arrayElemAt
Selectionner un élément du tableau et partir de son index:
tags[0] → 1er élément.

\$reverseArray
Inverser l'ordre du tableau:
Ex: [abc|d|e|f] → [f|d|e|abc]

\$size
Taille du tableau: nombre d'éléments d'un tableau.

Exemples

db.livres.aggregate([{"\$project": {"_id:0, first: {\$arrayElemAt: ["\$tags", 0]} }}])

db.livres.aggregate([{"\$project": {"_id:0, first: {\$reverseArray: ["\$tags"]}} }])

db.livres.aggregate([{"\$project": {"_id:0, size: {\$size: ["\$tags"]}} }])

Manipulation des dates:

Exemples | Fonctions | Explication

\$year
Extraire l'année du champ date et l'afficher

\$month
extraire mois du champ date et l'afficher → 1er mois

\$isoDayOfWeek
retourne le jour de la semaine en chiffre (lundi:1) → position du jour dans la semaine.

Exemples

db.livres.aggregate([{"\$project": {"year: "\$date-pub"} }])

db.livres.aggregate([{"\$project": {"month: "\$date-pub"} }])

db.livres.aggregate([{"\$project": {"isoDayOfWeek: "\$date-pub"} }])

\$match ... \$sort ... \$limit ... \$skip:

\$match: filtreage avec conditions

des titres des livres dont l'année est 2010 :

Ex: db.livres.aggregate([{"\$match: {"\$and: [{"year: "2010"}]}}, {"\$sort: {"year: 1"}}, {"\$skip: 1}, {"\$limit: 1}])
db.livres.find({date-pub: 2010}, {_id:0, title:1, date-pub:1})

les titres triés par date de publication :

db.livres.aggregate([{"\$sort: {"date-pub: -1"}}, {"\$project: {"_id:0, title:1}}])

skipper le 1^{er} titre et retourne les 2 derniers triés par date de publication.

db.livres.aggregate([{"\$skip: 1}, {"\$sort: {"date-pub: -1"}}, {"\$project: {"_id:0, title:1, date-pub:1}}])

\$group:

• **\$group** joue le même rôle que **group by** au niveau d'une requête SQL.

• **\$group** → 2 paramètres ↗ **_id**: identifiant du groupe → le champ sur lequel on va appliquer group by (Ex: group by nom).
fonction de groupe: fonction à appliquer sur le groupe (sum, count, avg).

• Nombres de livres par auteur: (Groupement simple)

db.livres.aggregate([{"\$group: {"_id: "\$author", nbr_livres: {"\$sum: 1}} }])

SQL ↗ SELECT count(*) nbr_livres GROUP BY author

Nombres de livres par nom-auteur et année (Groupement composé):

db.livres.aggregate([{"\$group: {"_id: {"nom_auteur: "\$author", année: "\$year", "date-pub": "\$date-pub"}, nbr_livres: {"\$sum: 1}} }])

SQL ↗ SELECT count(*) nbr_livres FROM nom_auteur, année GROUP BY nom_auteur, année

• Nombres de j'aime par auteur: **\$sum: nom champs**

db.livres.aggregate([{"\$group: {"_id: "\$author", sum_likes: {"\$sum: "\$likes"} } }])

SQL ↗ SELECT sum.likes FROM nom_auteur GROUP BY nom_auteur

Moyenne de likes : \$avg

db.livres.aggregate([{\$group:{_id:{\$avg:"\$likes"}, avg_likes:\$avg}}]), SQL SELECT avg(likes) avg_likes FROM livres

→ Ici moyenne totale = nb likes d'un auteur / total likes.

NB: {\$group:{_id:{}...}} → utiliser l'opérateur \$group même vide avec les fonctions de groupe (\$avg, \$max, \$min...) → on met _id et on laisse les {} vides.

→ Si on n'ajoute pas \$group → il va retourner un erreur.

- Si on ajoute \$author dans _id → moyenne par auteur.

- Livre ayant maximum de likes (plus grand nombre): \$max.

db.livres.aggregate([{\$group:{_id:\$max:"\$likes"}}, {"\$max": "\$likes"}]), SQL SELECT max(likes) max_likes FROM livres

- Livre ayant minimum de likes: \$min

db.livres.aggregate([{\$group:{_id:\$min:"\$likes"}}, {"\$min": "\$likes"}]), SQL SELECT min(likes) min_likes FROM livres

- Livre ayant minimum de likes par année:

db.livres.aggregate([{\$group:{_id:{year:"\$date-pub"}, min_likes:\$min:"\$likes"}}, {"\$min": "\$likes"}]), SQL SELECT min(likes) min_likes FROM livres GROUP BY year

- first → 1^{er} valeur de chaque groupe.
- last → Dernière valeur de chaque groupe } utilisée avec \$sort.

Ex: Premier auteur pour chaque année de publication → Si on a deux auteurs ont la même année, il va comparer toute la date (mois et jour).

db.livres.aggregate([{\$group:{_id:{year:"\$date-pub"}, auteur:\$first:\$author}}])

- Afficher le résultat sous forme d'un tableau.

• \$addToSet: [liste de livres] par auteur. Pas de duplication de valeur (on va obtenir un ensemble set)

↳ va s'afficher sous forme d'un tableau.

db.livres.aggregate([{\$group:{_id:"\$author", livres:\$addToSet:"\$title"}}, {"\$min": "\$likes"}]).

- \$push: similaire à \$addToSet → avec duplication de valeurs.

• \$lookup: Recuperer un document référence (Foreign key):

- \$lookup → permet de faire des références (comme une jointure) sur un autre document

db.conference.insert({title:"Hello", description:"ceci est une conference", speaker_id:"techno-maniac"})

de l'étranger.

db.speaker.insert({_id:"techno-maniac", name:"Ahmed", email:"m"})

db.conference.aggregate([{\$lookup:{from:"speakers", localField:"speaker_id", foreignField:"_id", as:"speaker-infos"}}, {"\$unwind": "\$speaker-infos"}])

• \$unwind: extraire des éléments d'un tableau hors de ce tableau → séparer les éléments d'un tableau.

Ex: Séparer les éléments du tableau tag (noms des) et sortir le nom des tags par identifiant.

db.livres.aggregate([{\$unwind:"\$tags"}, {"\$group:{_id:"\$tags", nbre_tags:\$sum:1}]), {"\$sort": {"count": -1}}])

- \$out: Recuperer le résultat d'aggrégation dans une autre collection.

db.livres.aggregate([{\$unwind:"\$tags"}, {"\$group:{_id:"\$tags", nbre_tags:\$sum:1}]), {"\$out": "resultat"}, {"\$sort": {"count": -1}}])

Visualisation du résultat du \$out dans resultat; db.resultat.find()

Cours BD NoSQL

Chapitre 5: MongoDB Indexation

Introduction

- Les index
 - améliorent performance en lecture: utilise avec find retourne un résultat rapide: au lieu de parcourir la collection, utilise l'index dans le recherche.
 - sont des objets de la BD.
 - peuvent avoir un impact sur les performances en écriture: Insertion / Mise à jour d'un document → mise à jour de tous les index.

Indexer dans MongoDB

→ très similaires aux SGBD relationnels

se fait sur 1 ou plusieurs champs

clé primaire composite

Utilisation de explain

- explain → retourne les informations sur l'utilisation des index.

db.students.find({student_id: 50}).explain("executionStats")

suivi de la requête: trace

d'exécution de la requête

- à travers explain, on peut avoir des informations sur:

→ temps de réponse de mongo: "executionTimeMillis" (Temps d'exécution).

→ comment il est en train de parcourir la collection: "execution Stages": { "stage"

- =COLLSCAN": en train de scanner toute la collection → parcourir la collection

Création et affichage d'un index

Création d'un index:

Créer un index → utilisant méthode createIndex. → création de l'index par index?

Ex: db.students.createIndex({ "student_id": 1 }) → 1: création par index

L'indexation de l'index sur ce champ

Affichage des index d'une collection:

- Afficher les index de la collection "students": db.students.getIndexes()

- Pour notre collection "students" → on a deux index :

→ Clé primaire " _id ": index par défaut (présent même si on ne créé pas d'index).

→ Clé primaire "student_id": créer un champ qui s'est transformé en index par la méthode createIndex. (mais recherche fait en direct).

Information sur l'index créé:

db.students.find({ student_id: 50 }).explain("executionStats")

une recherche par student_id → comparaison

résultat de recherche

- on constate suite à la création d'un index que:

→ = executionTimeMillis " a largement diminué (très proche de 0) ⇒ un temps de réponse / exécution plus rapide.

* = executionStages: { "stage": "FETCH", ... , inputStage: { "stage": "INDEX", ... } }

NB: on peut lors de la création d'index, attribuer un nom

à cet index

• Si on ne nomme pas l'index → MongoDB attribue un nom par défaut

Suppression d'un index:

• Suppression d'un index: db.students.dropIndex("student_id": 1)

• Suppression de tous les index: db.students.dropIndexes()

Types d'index:

Index Unique:

- Index Unique → Appliquer l'unicité à l'index défini → unique
 - un index créé → pour des champs et valeurs uniques (ses valeurs ne se répètent pas)
- Pour indexer que les champs et valeurs uniques:
 - db.students.createIndex({ "student_id": 1 }, { unique: true })
- NB:
 - lorsque l'index a des valeurs qui se répètent et on essaie de créer un index unique → MongoDB renvoie une exception.
 - Si un document n'a pas de valeur pour le champ indexé, l'index aura la valeur Null.
 - MongoDB permet l'insertion : 1 valeur nulle (seule) → 1 index unique.

Index Composé:

- Créer un index sur plusieurs champs (un index composé):
 - db.students.createIndex({ "student_id": 1, "type": 1, "score": 1 }, { name: "val2" })
 - un de primaire (index) composé de 3 champs : student_id, type, score
 - optionnelle : nommer un index.
- Index composé
 - Requête de recherche possible (find)
 - student_id
 - student_id, type
 - student_id, type, score
 - Requête de recherche non possible
 - type
 - score
 - type, score
- Ex: db.students.find({ "student_id": 50 }) .explain("executionStats") ⇒ Recherche par index
 - db.students.find({ "type": "Essay" }) .explain("executionStats") ⇒ Recherche classique : va parcourir toute la collection.

⇒ Il faut respecter l'ordre de création des index (l'ordre de définition des champs dans la méthode createIndex).

Sparsité:

- Sparsité permet d'indexer uniquement les documents contenant des valeurs non nulles d'un champ → des index spécifiques / partiels
 - on n'indexe pas tous les documents
 - indexe les documents dont les champs sont non nuls.

Ex: On va insérer 3 documents : db.series.insert({ "word": "newbie" }) ⇒ n'est pas indexé
 db.series.insert({ "word": "ahby", "score": 82 }) ⇒ indexé
 db.series.insert({ "word": "nina", "score": 90 }) ⇒ indexé

2) Créer un index sparsé sur le champ score : db.series.createIndex({ "score": 1 }, { sparse: true })

⇒ le 1^{er} document { "word": "newbie" } n'est pas indexé car le champ score est null.

Index Partiel:

- Index Partiel
 - représentent un sous-ensemble des fonctionnalités offertes par les index
 - déterminent les entrées d'index en fonction du filtre spécifique sur un champ choisi
- Crédit de l'index : db.students.createIndex({ "score": 1 }, { partialFilterExpression: { "score": { \$gt: 50 } } })

- On va indexer uniquement que les documents dont le score est > 50.
- Utilisation de l'index
 - Index global : db.scores.find({score: 62}).explain("executionStats")
 - on cherche les scores = 62 > 50 → Recherche par index global.
 - Index non utilisé : db.scores.find({score: 15}).explain("executionStats")
 - on cherche les scores = 15 < 50 ⇒ Recherche classique.
- ⇒ Index partial : pas besoin d'indexer toute la collection, on indexe une partie de la collection → à utiliser lorsque l'on sait ou préalable qu'on va faire des requêtes selon un critère (Ex: les requêtes sur des scores sup à 50).

hint: force l'utilisation d'un index spécifique pour répondre à la requête.

Ex: Forcer l'utilisation de l'index student_id :

db.students.find(), hint({student_id: 1}).explain("executionStats").

Utilisation de sparse avec hint:

- On a un index sparse sur le champ "score" de la collection score :
 - db.scores.find() → Recherche classique sans index (Pas de critère) → même les documents sans score vont être affichés.
 - db.scores.find().hint({score: 1}) → ajoute hint ; on va utiliser l'index sparse.
- ⇒ on va avoir que les documents avec un score non nul.

Index textuel:

NB: On peut enchaîner et imbriquer plusieurs index (sparse + hint, Index composite + Index unique →)

• Index textuel → index du texte pour prendre en charge les requêtes de recherche de texte sur le contenu de chaîne.

• Index de texte → peut inclure des champs de valeur → tableau de chaîne

Ex: 1) Créer un index textuel : db.livres.createIndex({description: "text"})) → pour indiquer que l'index est textuel.

2) Chercher le mot "sexe" dans la description : db.livres.find({\$text: {\$search: "sexe"}}) → on va indexer sur le champ description.

→ Il va chercher sexe ou Sexe puisqu'on a pas indiqué qu'il doit être sensible à la casse. ({\$caseSensitive: true}).

Ex: db.livres.find({\$text: {\$search: "sexe"}, \$caseSensitive: true}).

3) Chercher la sous chaîne "Java Script côté" (la rétention de sous chaîne) :

db.livres.find({\$text: {\$search: "\\"Javascript côté\\\""}, \$description: 1}).

4) Chercher toutes les combinaisons possibles (sans ajouter "\")

Java script côté
Java script côté :

db.livres.find({\$text: {\$search: "Java script côté"}, \$score: {\$meta: "TextScore"}}, {\$sort: {\$score: {\$meta: "TextScore"}}}).

\$\$meta → fonction prédefined qui donne un score de ressemblance selon la chaîne entrée.

Les deux sortes de "transcripte" → livres en couleur. Les premières ("transcripte") sont en couleur. Les deux sortes de transcripte sont le transcripte et le transcripte.

Indice giro apertidi:
 Indice ≥ 0 - aperto \rightarrow bolo \rightarrow tipo "2d" \rightarrow dentro sistema en el que piden un
 a condensación \swarrow integrado \searrow bolo \rightarrow tipo "2d" ($-2, 0, 2, 3, 4$)

Classmate Practiced the class. With India (1 hour)

Exemple : les deux points suivants appartiennent à la portion $[50,100]$: