TP4 : Agrégations MongoDB II

Plan de l'Atelier

1. Introduction aux agrégations MongoDB et au concept de pipelines.
2. Exercices guidés pour se familiariser avec les opérateurs.
3. Cas pratiques interactifs en petits groupes.
4. Discussion et synthèse pour consolider les apprentissages.

Introduction

Qu'est-ce qu'une Agrégation dans MongoDB ?

Une agrégation est un processus permettant de traiter des données dans MongoDB en appliquant une série d'étapes (appelées *pipelines*) pour regrouper, trier ou transformer les documents d'une collection.

Les Principaux Opérateurs d'Agrégation

Voici une liste des opérateurs d'agrégation, classés par leur fonction :

1. Opérateurs de Pipeline

Ces opérateurs définissent les étapes principales du pipeline :
— **$match** : Filtre les documents selon des critères spécifiques (similaire à un `find`).
— **$group** : Regroupe les documents selon une clé et applique des fonctions d'agrégation (`$sum`, `$avg`, etc.).
— **$project** : Sélectionne ou transforme les champs à inclure dans le résultat.
— **$sort** : Trie les documents selon un ordre spécifié.
— **$limit** : Limite le nombre de documents transmis à l'étape suivante.
— **$skip** : Ignore un certain nombre de documents avant de transmettre les suivants.
— **$lookup** : Effectue une jointure avec une autre collection.
— **$addFields** : Ajoute de nouveaux champs aux documents.
— **$out** : Écrit le résultat du pipeline dans une nouvelle collection.

2. Opérateurs d'Agrégation dans $group

Ces opérateurs effectuent des calculs sur les groupes de documents :
— **$sum** : Calcule la somme des valeurs.
— **$avg** : Calcule la moyenne des valeurs.
— **$min** : Trouve la valeur minimale.
— **$max** : Trouve la valeur maximale.
— **$first** : Récupère le premier document du groupe.
— **$last** : Récupère le dernier document du groupe.
— **$push** : Ajoute des valeurs dans un tableau pour chaque groupe.
— **$addToSet** : Ajoute des valeurs uniques dans un tableau pour chaque groupe.

3. Opérateurs de Transformation

Ces opérateurs permettent de modifier ou de générer des champs :
— **$concat** : Concatène des chaînes de caractères.
— **$substr** : Découpe une sous-chaîne.
— **$toLower** : Convertit une chaîne en minuscules.
— **$toUpper** : Convertit une chaîne en majuscules.
— **$arrayElemAt** : Récupère un élément d'un tableau par son index.
— **$size** : Donne la taille d'un tableau.
— **$mergeObjects** : Fusionne plusieurs objets.

4. Opérateurs Logiques et Conditionnels

Ces opérateurs sont utilisés pour des expressions conditionnelles :
— **$cond** : Implémente une condition `if/else`.
— **$eq**, **$ne** : Compare deux valeurs pour tester l'égalité ou l'inégalité.
— **$gt**, **$gte** : Vérifie si une valeur est supérieure (ou égale).
— **$lt**, **$lte** : Vérifie si une valeur est inférieure (ou égale).
— **$and**, **$or**, **$not** : Combine plusieurs conditions logiques.

5. Opérateurs Géospatiaux

Ces opérateurs permettent de travailler avec des données géographiques :
— **$geoNear** : Effectue une recherche basée sur la proximité géographique.
— **$geoWithin** : Trouve des documents à l'intérieur d'une zone définie.
— **$geoIntersects** : Trouve des documents qui intersectent une zone.

Exercices

```
db.orders.insertMany([
```

```
  { "order_id": 1, "name": "Margarita", "size": "small", "quantity": 2, "
      price": 8.5 },
  { "order_id": 2, "name": "Pepperoni", "size": "medium", "quantity": 1, "
      price": 10.0 },
  { "order_id": 3, "name": "Vegan", "size": "large", "quantity": 3, "price":
       12.0 },
  { "order_id": 4, "name": "Vegan", "size": "small", "quantity": 1, "price":
       12.0 },
  { "order_id": 5, "name": "Margarita", "size": "medium", "quantity": 5, "
      price": 8.5 },
  { "order_id": 6, "name": "Pepperoni", "size": "large", "quantity": 2, "
      price": 10.0 },
  { "order_id": 7, "name": "Margarita", "size": "large", "quantity": 3, "
      price": 8.5 },
  { "order_id": 8, "name": "Vegan", "size": "medium", "quantity": 4, "price
      ": 12.0 }
]);
```

Exercice 1 : Requêtes Simples

1. **Calculez le nombre total de commandes :**

```
db.orders.countDocuments()
```

2. **Affichez les noms uniques de pizzas :**

```
db.orders.distinct("name")
```

3. **Affichez toutes les commandes :**

```
db.orders.aggregate([{ $match: {} }])
```

Exercice 2 : Agrégations Basées sur les Données

1. **Commandes pour les pizzas Vegan :**

```
db.orders.aggregate([{ $match: { name: "Vegan" } }])
```

2. **Total des pizzas vendues :**

```
db.orders.aggregate([
  { $group: { _id: null, totalPizzas: { $sum: "$quantity" } } }
])
```

3. **Total des pizzas par nom :**

```
db.orders.aggregate([
  { $group: { _id: "$name", totalPizzas: { $sum: "$quantity" } } }
])
```

Exercice 3 : Combinaison d'Opérateurs

1. **Prix total des pizzas small trié par ordre décroissant :**

```
db.orders.aggregate([
  { $match: { size: "small" } },
  { $group: { _id: "$name", totalPrice: { $sum: "$price" } } },
  { $sort: { totalPrice: -1 } }
])
```

2. **Quantité totale et prix total des pizzas small :**

```
db.orders.aggregate([
  { $match: { size: "small" } },
  { $group: {
      _id: "$name",
      totalQuantity: { $sum: "$quantity" },
      totalPrice: { $sum: "$price" }
  } }
])
```

Gestion d'un Réseau Social

Contexte

Vous développez une application de réseau social similaire à Twitter, où :
— Les utilisateurs peuvent publier des **posts** contenant du texte, des images ou des vidéos.
— Chaque post peut être associé à des **tags** et recevoir des **réactions** (like, commentaire).
— Les utilisateurs suivent d'autres utilisateurs pour voir leurs publications.
La base de données NoSQL doit permettre de :
— Stocker efficacement les publications et leurs métadonnées.
— Effectuer des recherches rapides (par utilisateur, tag ou type de contenu).
— Supporter la flexibilité des données (contenu des posts non structuré).

Modélisation NoSQL

Collection Structurée : `users`

Structure définie pour gérer les informations des utilisateurs.

```
{
  "user_id": 1,
  "name": "Alice",
  "email": "alice@example.com",
  "following": [2, 3], // Liste des IDs des utilisateurs suivis
  "followers": [3] // Liste des IDs des utilisateurs qui suivent Alice
}
```

Collection Non Structurée : `posts`

Structure flexible pour gérer des publications variées.

```
{
  "post_id": 101,
  "user_id": 1,
  "content": {
    "text": "J'adore les bases de données NoSQL!",
    "image": "photo_url",
    "video": null
  },
  "tags": ["NoSQL", "MongoDB"],
  "reactions": {
    "likes": 10,
    "comments": [
      {"user_id": 2, "text": "Moi aussi!"},
      {"user_id": 3, "text": "Super intéressant."}
    ]
  },
  "created_at": "2024-12-03T10:00:00Z"
}
```

—

Requêtes MongoDB

1. Afficher tous les posts d'un utilisateur

Récupérer les publications de l'utilisateur ayant l'ID `1` :

```
db.posts.find({ "user_id": 1 })
```

2. Rechercher des posts par tag

Afficher tous les posts associés au tag `"NoSQL"` :

```
db.posts.find({ "tags": "NoSQL" })
```

3. Compter les likes sur tous les posts d'un utilisateur

Calculer le total des `likes` sur les posts d'un utilisateur :

```
db.posts.aggregate([
  { $match: { "user_id": 1 } },
  { $group: { _id: null, totalLikes: { $sum: "$reactions.likes" } } }
])
```

4. Ajouter une réaction (like) à un post

Mettre à jour le post ayant l'ID `101` pour ajouter un `like` :

```
db.posts.updateOne(
  { "post_id": 101 },
  { $inc: { "reactions.likes": 1 } }
)
```

5. Afficher les posts contenant un mot-clé spécifique dans le texte

Récupérer les posts dont le champ `text` contient le mot-clé `"NoSQL"` :

```
db.posts.find({ "content.text": { $regex: "NoSQL", $options: "i" } })
```

6. Rechercher les utilisateurs suivis par un utilisateur spécifique

Récupérer la liste des utilisateurs suivis par l'utilisateur ayant l'ID `1` :

```
db.users.find(
  { "user_id": 1 },
  { "following": 1, "_id": 0 }
)
```

—

Étapes Suivantes

— Ajouter des fonctionnalités comme les partages de posts.
— Implémenter une recherche avancée combinant plusieurs critères (par utilisateur, tag, date, etc.).
— Optimiser les performances avec des index sur les champs utilisés fréquemment (`user_id`, `tags`, etc.).

Gestion et Analyse des Données GPS

Une application de navigation comme Waze collecte en temps réel des données GPS provenant de millions d'utilisateurs pour fournir des informations sur le trafic, les itinéraires optimaux et les incidents sur la route. Vous devez modéliser une base de données NoSQL et écrire des requêtes pour répondre aux besoins suivants.

Contexte

L'application collecte les données suivantes :
— **Localisation GPS** : Latitude et longitude de chaque utilisateur.
— **Trajets** : Historique des trajets avec l'heure de début, l'heure de fin, et la distance parcourue.

— **Incidents** : Signalisations des utilisateurs sur des événements comme les embouteillages, les accidents ou les travaux.
— **Utilisateurs** : Profils des utilisateurs, comprenant l'identifiant, le nom et la liste des trajets réalisés.

Objectifs de l'Exercice

1. Modéliser deux collections NoSQL :
— Une collection structurée pour gérer les informations des utilisateurs.
— Une collection non structurée pour enregistrer les trajets et les incidents signalés.
2. Écrire des requêtes MongoDB pour :
— Récupérer tous les incidents signalés dans une zone géographique spécifique.
— Calculer la distance totale parcourue par un utilisateur donné.
— Identifier les trajets où la vitesse moyenne dépasse une limite donnée (par exemple, 100 km/h).
— Compter le nombre total d'incidents signalés par type (embouteillage, accident, travaux, etc.).
— Trouver les itinéraires alternatifs pour contourner une zone impactée par un incident.

Hypothèses et Contraintes

— La base de données doit pouvoir gérer des millions de points GPS et incidents en temps réel.
— Les données GPS doivent être organisées pour optimiser les recherches par zone géographique.
— Les signalisations d'incidents doivent inclure la nature de l'incident, son emplacement exact, et l'heure de signalement.