BUT Sciences des Donnes 3A – R5.01

base de données non relationnelles - 2023/2024

MongoDB en bref

1. MongoDB

MongoBD stocke des données sous forme de **documents au format JSON binaire** : le **BSON**. Le BSON comporte quelques types supplémentaires, comme le **type date.**

Le serveur MongoDB est organisé en plusieurs base (databases) :
Chaque database contient des collections.
Chaque collection contient des documents.
Chaque document est au format JSON et contient donc des propriétés (autres noms : champs, attributs)

Comparaison SQL / MongoDB

SQL	MongoDB
base de données + schéma base de données	Base de données
(Quelques) Table(s)	Collection
Enregistrement, raw	document
Attribut (atomique)	Propriété, atomique ou complexe

Schema-less
C'est une base schema-less, aussi une collection peut contenir des documents de structures différentes. Il n'est pas possible de définir la structure a priori d'une collection. La structure d'une collection est définie par les documents qui la composent, et elle peut évoluer dynamiquement au fur et à mesure des insertions et suppressions.

Identification clé / valeur
Chaque document est identifié de manière unique au sein d'une collection par un champ nommé "**_id**". Ce champ est géré automatiquement par mongo par défaut, mais on peut aussi lui donner une valeur explicite.

Architecture
MongoDB fonctionne a minima sous la forme d'un serveur auquel il est possible de se connecter avec un client textuel (mongo shell), ou d'autres outils (langage, UI ...)
Dans les dernières versions, dont celle du TP, l'accès via le shell est un outils séparé. (**mongosh**)

MongoDB peut être distribuée sur plusieurs serveurs (partitionnement horizontal ou sharding) et accédée à travers de multiples couches applicatives (langages, API…), comme par exemple l'UI compass

Documentation : **https://www.mongodb.com/docs/manual/**

2. Le shell mongodb - mongosh

Le shell mongo, ***mongosh***, est une console interactive pour interagir avec le serveur mongodb (REPL environnement – Read Eval Print Loop). C'est l'équivalent de *psql* pour un serveur postgresql, ou *mysql* pou un serveur MySQL.

Il permet de gérer et requêter les données, et d'effectuer des tâches d'administration sur le serveur.

Il est basé sur un interpréteur JavaScript, on peut donc utiliser des fonctions et la syntaxe javascript dans la console.

Le shell mongo se connecte automatiquement sur le serveur local (localhost), sur le port par défaut 27017

Il peut être utilisé en mode interactif, ou pour exécuter des fichiers javascript (.js). Il y a quelques différences d'utilisation entre les deux (détaillées un peu plus bas).

3. Créer des bases de données et des collections

MongoDB est une base schema-less : la création des bases et des collections est dynamique lors d'une première insertion de données

Pour créer une base de données, il faut exécuter l'instruction `use new_db` puis donner un ordre d'insertion dans une nouvelle base : `new_db.collection.insert({"x":1})`

L'insertion associe un automatiquement un identifiant (_id) à chaque document de la collection.

Shell - Quelques commandes utiles

`use new_db`	La base par défaut utilisée sera db1
`db`	Affiche la base par défaut utilisée
`db.new_coll.insert(...)`	Insère un document json en créant si besoin la base '`new_db`' et la collection '`new_coll`'
`show dbs`	Liste les bases existantes
`show collections`	Liste les collections de la base par défaut
`db.my_collection.find(...)`	Cherche dans le contenu d'une collection
`db.my_collection.drop()`	Efface une collection et ses index
`db.dropDatabase()`	Efface la base courante

Mongo shell vs javascript

Pour exécuter un script .js (javascript) contenant des commandes à une base mongodb :

1) Depuis le shell mongo : `load("script.js")`

2) Depuis une invite de commande : `mongosh script.js`

Dans un script .js, il faut tout d'abord se connecter à une base de donnée mongo :

`connection=new Mongo();` // connection par default au localhost

`mydb=connection.getDB('my_db');` // variable pointant vers la base de données 'my_db'

Ensuite, là où dans le shell mongo on utilise 'db', on utilisera dans le javascript la variable 'mydb'

Dans un script js, les résultats d'une requête ne s'affichent pas automatiquement, il faut les parcourir avec un curseur, par ex :

```
recordset = mydb.test.find(….)
while ( recordset.hasNext() ) {
    printjson( recordset.next() );
}
```

Voir la doc javascript pour plus d'info, https://developer.mozilla.org/fr/docs/Web/JavaScript

Ou effectuer des recherches sur la toile, il y a beaucoup d'exemples.

4. Recherche dans une collection – méthode find()

La **méthode 'find'** permet d'effectuer des recherches dans les documents des collections.

Utilisation : `db.collection.find(Document JSON, document JSON)`, avec :

- le premier document JSON définit une restriction, c'est à dire un filtre sur les documents retournés
- le second document JSON définit une projection. C'est à dire quel sont les champs qui seront retournés.

Les principales manières de filtrer (restrictions)

{"attribut1":"valeur1", "attribut2" :"valeur2"} *(Attention : {"Field1":"valeur1", "Field1" :"valeur2"} n'est pas un document json valide)*	Utilisation basique. La virgule a valeur de ET. Attention, pas le même champ à un même niveau.
{"attribut1.subfield":"valeur1"}	Filtre sur le champ d'un sous objet
{$or : [{"a": 1},{"b": 5}]}	Opérateurs logiques : `$and`, `$or`, `$not`, `$nor`
{"notes": {$gte: 13}}	Opérateurs de comparaison d'un champ et d'une valeur : $eq, $ne, $gt, $gte, $lt, $lte, $in
{ $expr: { $gt: ["$spent" , "$budget"] } })	Comparer 2 champs avec $expr
{ "tags": { $all: ["red", "blank"] }} inclusion { "tags": { $in: ["red", "blank"] } } intersection	Opérateurs sur les tableaux `$in $nin $all $size`
{"name":{"$regex" : "string", "$options"": "i"}} {"name": /string/i}	Recherche dans des textes – expressions régulières – ILIKE '%name%'
`{ field : null }` : champ non existant ou null { field: {$ne: null } } : champ existant et non null `{ field: { $exists: true } }` champs existe `{ field: { $exists: false } }` champ n'existe pas	Tester l'existence d'un champ avec ou sans les valeurs nulles
{ quantite: { $exists: true, $nin: [5, 15] } }	Combiner des conditions sur un même attribut

Voir le manuel : https://www.mongodb.com/docs/manual/reference/operator/query/

Projection

db.users.find({}, { name: 1, roles: 1 })	Affiche les champs name et roles + le champs _id par défaut
db.users.find({}, { name: 1, roles: 1, _id : 0 })	Affiche les champs name et roles mais pas le champ _id
db.users.find({ }, { name.first: 0, birth: 0)	Affiche tous les champs sauf name et birth
db.bios.find({ },{ _id: 0, 'name.last': 1, roles: { $slice: 2 } })	Affiche name.last, et les 2 premiers éléments de roles (roles est un tableau)

Méthodes utiles associées aux requêtes

db.coll.distinct("key", {<conditions>})	Valeurs distinctes d'un champ
db.coll.find().size()	Compter le nombre d'éléments
db.coll.find().sort({"key1" : 1, "key2" : -1})	Trier les résultats
db.coll.find().limit(10)	Limite le nombre de documents retournés

On peut chainer certaines actions :

```
db.test.find().limit( 5 ).sort( { name: 1 } )
```

Utiliser les sous objets et les éléments des tableaux :
- Référencer un sous objet: "`field.subfield`". **Attention : Les " ne sont plus optionnels**
- Accéder au 1er element d'un tableau : `"array.0"`
- Accéder au champ 'field1' dans un tableau d'objets : `"array.field1"`
- Accéder au champ 'field' du premier object d'un tableau d'objets : `"array.0.field1"`

Comparaison d'objet :
Quand on filtre sur un objet : {object:{….}}
Les objets sont considérés les même si ils contiennent les même valeurs pour les **mêmes champs dans le même ordre**

Utile :
Le shell retourne les 20 premières lignes des résultats par défaut. Il faut taper "it" pour accéder aux résultats suivants.
Pour changer cette valeur, et afficher par exemple 100 résultats d'un coup :
`config.set("displayBatchSize",100)`

5. Le Pipeline MongoDB et la méthode aggregate()

La fonction `aggregate()` permet de chainer plusieurs opérations de suite sur les données, dont des opérations de regroupement (agrégation). C'est le **pipeline d'agrégation.** Les possibilités sont très riches, nous n'en verront qu'une petite partie.

Doc :
https://www.mongodb.com/docs/manual/reference/operator/aggregation-pipeline/

Chaque opération (vocabulaire mongo : **stage ou étape**) est appliquée sur les résultats de l'opération précédente : l'ordre compte.

Utilisation : on passe en paramètre à la fonction aggregate() un tableau json contenant les étapes (ou opérations) à appliquer successivement sur les données.

Exemple du pipeline sur la base cinema :
Liste des film parus après 1972, on affiche uniquement le titre et l'année, et trié par année décroissante (note : on peut faire une requête équivalent avec le find)

```
db.cinema.aggregate([
    {$match:{annee:{$gt:1972}}},
    {$project:{titre:1, annee:1, _id:0}},
    {$sort:{annee:-1}}
])
```

Les étapes (stages) clefs sont : $match ; $project et $group

$match : Filtre les documents qui respectent certaines conditions.
`{ $match: { <query> } }`
`Query :` Même utilisation que pour le premier argument de la méthode find.

$group : groupe les documents suivant une clef donnée. En sortie, on a un document pour chaque clef. La clef peut être un champ ou plusieurs champs.
Il faut utiliser le champ "_id" pour définir la clef du groupe. Il faut aussi fournir un opérateur d'agrégation.
Exemple : compte le nombre de film par année :

```
db.cinema.aggregate([
  { $group: {
      _id: {"annee": "$annee"},
      "nb_film": {$sum: 1}
  } }
])
```

$project : modifie la forme des documents : ajoute, enlève, modifie des champs, … (raccourcis : $set, $unset, $addField)
Exemple – on veut un champs "realisateur" qui concatene les nom-prenom des réalisateurs, et qui les place qu même niveau que titre et annee :

```
db.cinema.aggregate([
    { $project:{
            titre:1,
            annee: 1,
            "_id": 0,
            realisateur:{
                $concat:["$realisateur.prenom"," ","$realisateur.nom"]
            }
    } }
])
```

$sort : tri les résultats. Même utilisation que pour la méthode find

$limit – limite le nombre de résultats de sortie à n. Usage : `{ $limit: n } )`

$count – compte le nombre de documents.

Exemple d'utilisation : nombre de document qui ont pour pays 'FR' :

```
db.films.aggregate([
   { $match:{ country:"FR" } ,
   { $count: "nb_fr" }
])
```

$unwind : transforme un tableau en document pour chaque élément du tableau (permet par ex d'appliquer des opération de regroupement sur le contenu d'un tableau)

usage de base : { $unwind: "$nom_tableau"}

$skip – skippe les n premiers documents (`{ $skip: n }`)

Enfin on peut noter sur les versions récentes de mongo la possibilité de réaliser des opérations ensemblistes (`$unionwith`) et de jointure (`$lookup`), proches des fonctionnalités du relationnel.

Les opérateurs

Pour effectuer des calculs, des modifications sur les valeurs des champs, on peut utiliser des opérateurs. Ils s'appliquent dans le cadre d'un `$group` (une agrégation) ou non.

https://www.mongodb.com/docs/manual/reference/operator/aggregation/

La liste est longue, voici un petit échantillon
numériques : `$add, $substract , $sum, $sqrt, $multiply, $avg, $max, $min`
string : `$concat, $toUpper, $toLower, $trim, $regexMatch`
fonction de date, booléennes, array, etc

6. Recherches textuelles

Expressions régulières – regex (regular expressions)

Pour faire des recherches de type 'string contains' , équivalentes au like ' %%' en SQL, on utilise avec mongodb des expressions régulières ou regex.

Rappel (ou pas) : les expressions régulières permettent de faire des recherches dans des chaines de caractère en fonction de différents patterns (motifs).Ce sont des outils très puissant et régulièrement utilisés qu'on retrouve dans différents langages.

ex :

films dont le titre se termine par le pattern Métro

```
db.films.find( { title: { $regex: /Métro$/ } } )
```

films dont le titre contient pattern métro, et indifférent à la casse

```
db.films.find( { title: { $regex: /métro/i } } )
```

films dont le titre contient le pattern métro ou metro, et indifférent à la casse

```
db.films.find( { title: { $regex: /m[ée]tro/i } } )
```

films dont le titre commence par le pattern Pierrot, et indifférent à la casse

```
db.films.find( { title: { $regex: /^pierrot/i } } )
```

films dont le titre contient des caractères numériques

```
db.films.find( { title: { $regex: /[0-9]/ } } )
```

Collation

On appelle ‘collation’ un ensemble de règles qui permettent de comparer et trier des jeux de caractères.

On peut se servir de la collation pour faire des recherches exactes sur les strings sans tenir compte des caractères accentués (caractères diacritiques) ou sans tenir compte de la casse.

Exemple avec `find()` : liste des films dont le prénom du directeur est Francois ou François avec ou sans majuscules:

```
db.films.find( { "director.first_name": "francois"}).collation({ locale: "fr", strength:1 } )
```

Exemple avec `aggregate()` : l’ajout de la collation fr permet de trier en tenant compte des prénoms commençant par des lettres accentuées.

```
db.films.aggregate( […,{ $sort: {prenom:1} }], {collation:{ locale: "fr" }} )
```

- ‘**locale**’ défini le langage utilisé ; obligatoire

- ‘**strength**’ défini le type de comparaison . **1** : ne tiens pas compte des caractères diacritiques ni de la casse, **2** : ne tiens pas compte des caractères diacritiques mais prend en compte la casse, **3** : prend en compte les caractères diacritiques et la casse

La valeur par défaut pour *strength* est 3

Dommage, on ne peut pas utiliser la collation avec les regex.

Si on veut faire des requêtes textuelles plus poussée, on peut utiliser un **index de type text,** mais on peut en définir **un seul par collection** (hors utilisation via le cloud Atlas).

Si c’est bloquant, une solution peut être de basculer sur une base elasticsearch

Appendice : Licence, credit