# Présentation

## Qu'est-ce que mongoDB ?

- Base de données moderne et non relationnelle
- Fondée en 2007 (DoubleClick/10gen)
- Open-sourced en 2009
- Changement de licence en 2018

## Un logiciel largement utilisé

BrightRoll Castlight Health Citrix Craiglist Ebay ElectronicArts Forbes Foursquare GILT Github HTC IBM InVision MetLife SAP Shutterfly Sony T-Mobile Toyota Twitter Verizon ViaCom Zendesk ...

## Pourquoi utiliser MongoDB ?

- Combler le fossé entre les systèmes de stockage clé-valeur et les SGBDR
- Stockage orienté document
    - Les données sont stockées sous la forme de documents de style JSON.
- Indexation sur n'importe quel attribut
- Réplication et haute disponibilité
- Auto-sharding
- Requêtes riches
- Mises à jour rapides sur place
- Nombreux drivers pour des langages de programmation
- Support professionnel de MongoDB

## Avantages de MongoDB par rapport aux SGBDR

- Moins de schéma
- La structure d'un objet unique est claire.
- Pas de jointures complexes.
- Possibilité d'interrogation approfondie.

- Tuning.
- Facilité de mise à l'échelle : MongoDB est facile à faire évoluer.
- La conversion/mise en correspondance des objets de l'application en objets de la base de données n'est pas nécessaire.

## Limitations de MongoDB

- Pas de transactions
- Pas de jointures
- Entiers 32 bits
  - MongoDB corrigé
  - Certains drivers limitent encore à 32 bits (même dans PHP 7.x 64bits)
- Enorme consommation de RAM
- Taille du document : 16 Mo
- Authentification

## Où utiliser MongoDB ?

- Big Data
- Gestion et diffusion de contenu
- Infrastructure mobile et sociale
- Gestion des données utilisateur
- Hub de données

# Le mouvement NoSQL

## Définition

- Terme "flou"
- Types de bases de données non (seulement) relationnelles
    - stocker les données dans un format différent des "tables" relationnelles
- Interroger à l’aide d'un langage spécifique
    - non algébrique
    - non-relationnel
    - mais "riche" quand même (selon la BDD)
- Permettre une scalabilité horizontale facile
- Manipuler de volumes de données importants

## Théorie

- Orienté-agrégats
    - groupe de données (documents) modifiée en meme temps
    - encapsulation de données dans d'autres données (au lieu d'une relation)
- Orienté-graphes
    - documents = node
    - requetes multi-document sans jointures
    - ne pas confondre avec triple-stores (orienté relation)
- Sans-schéma
    - capacité d'intégration de données hétérogènes
    - facilité d'évolution
    - déporter les vérifications de cohérence & consistance coté applicatif

## Typologies

- Clé-valeur (ex: Redis, Riak)
- Graphe (Neo4j, HyperGraphDB)
- Document (MongoDB, CouchDB)

- Big Table (Cassandra, HBase)

## Propriétés ACID

Les capacités ACID garantissent que si plusieurs utilisateurs font de manière simultanée des modifications des données,

- toutes les modifications vont être prises en compte,
- dans un ordre précis et maîtrisé de manière à avoir un résultat cohérent (intégrité des données)
- avec l'historique des modifications faites par chacun.

La mise en œuvre stricte des capacités ACID entraîne des coûts logiciels importants et un niveau de performance moindre à infrastructure matérielle équivalente.

## Références


- https://fr.wikipedia.org/wiki/NoSQL#NoSQL_orient%C3%A9-agr%C3%A9gats

# Rappels sur les bases de données

## Théoreme CAP

### Trois contraintes

**Cohérence** (Consistency)

- Tous les noeuds du système voient les mêmes données au même moment

**Disponibilité** (Availability)

- Garantie que toutes les requêtes (en lecture ou en écriture) reçoivent une réponse

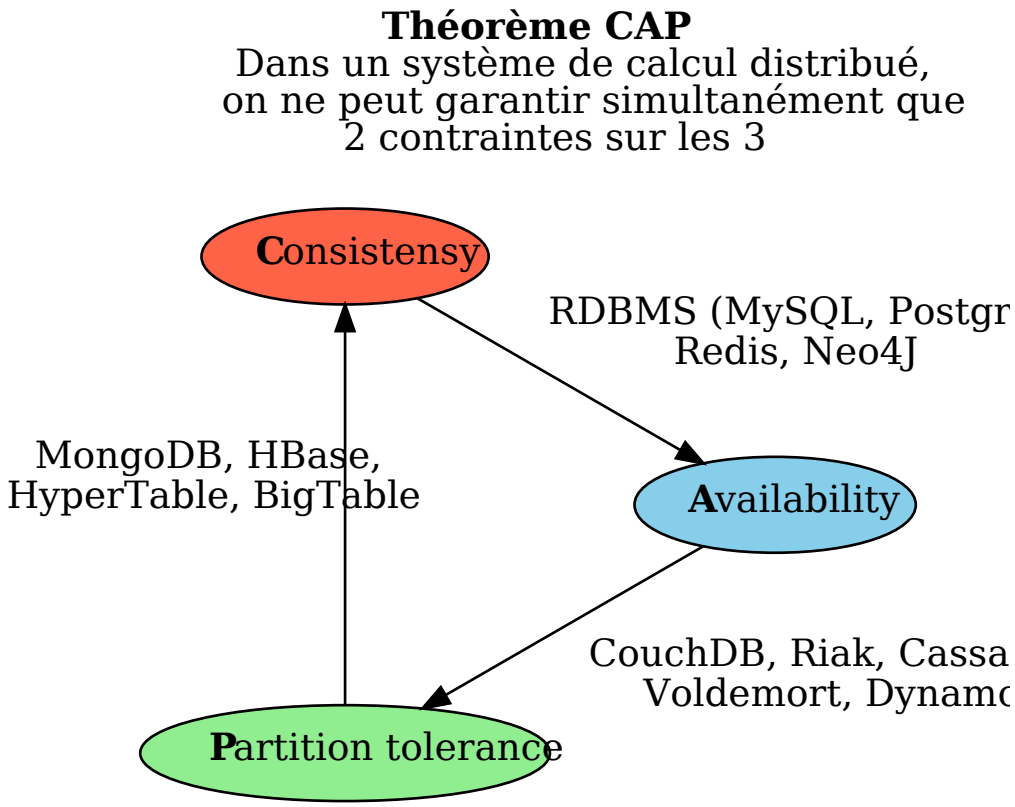
**Tolérence au partitionnement** (Partition tolerance)

- Aucune panne moins importante qu'une coupure totale du réseau ne doit empecher le systeme de répondre correctement
- En cas de morcellement en sous-réseaux, chacun doit pouvoir fonctionner de maniere autonome comme un seul système

### Énoncé du théorème - et propriétés des BDD

D'après ce théorème :

- Il est impossible sur un systeme informatique de calcul distribué de garantir en même temps (de manière synchrone) les trois contraintes C-A-P.
- Un système de calcul distribué ne peut garantir que deux de ces contraintes.




> **Note**
>
> En général, les SGBD essayent de compenser la partie qu'elles ne peuvent pas garantir, au moins pour certains cas d'usage.

## Propriétés ACID

**Atomicité (Atomic)**

- Une transaction est entière : ^S tout ou rien ^T.

**Cohérence (Consistent)**

- Une transaction amène la base d’un état stable à un autre.

**Isolation (Isolated)**

- Les transactions n’agissent pas les unes sur les autres.

**Durabilité (Durable)**

- Une transaction validée provoque des changements permanents.

Les propriétés ACID sont quatre propriétés essentielles d’un sous-système de traitement de transactions d’un système de gestion de base de données

> **Important**
>
> On considère souvent que seuls les SGBD qui respectent ces propriétés sont dignes d’être considérées comme des bases de données

# Rappels sur JSON

- `JavaScript Object Notation`
- Créé en 2005
- Format d'échange de données structurées léger
- Schéma des données non connu
  - contenu dans les données
- Basé sur deux notions :
  - collection de couples clé/valeur
  - liste de valeurs ordonnées

## Syntaxe et types en JSON

- Similaire à du JavaScript uniquement pour les données
- On parle de **littéral**
- Deux types atomiques ( `string` et `number` )
- Trois constantes ( `true` , `false` , `null` )
- Structures possibles :
  - objet (couples clé/valeur) :
    - `{}`
    - `{ "nom": "doe", "prenom": "jon" }`
  - tableau (collection de valeurs) :
    - `[]`
    - `[ 1, 5, 10]`
  - une valeur dans un objet ou dans un tableau peut être elle-même un littéral

## Validation du JSON

Validation possible du JSON à l'aide de jsonlint.com/

```
{
  "name": "John Doe",
  "age": 42,
```

```
  "food": ["pizza", "cake"],
  "wife": {
    "name": "Jane Doe",
    "age": 42
  }
}
```

> **Note**
>
> Il existe aussi des moyens de définir des structures de données spécifiques en JSON (notion de schéma de donneés) et de valider un document d'après ce schéma
> Voir https://json-schema.org/

## Compléments

`BSON` : extension de `JSON`

- Quelques types supplémentaires (identifiant spécifique, binaire, date, ...)
- Distinction entier et réel

# Modèle de données

## Equivalences avec les SGBDR

Le tableau suivant montre la relation entre la terminologie des SGBDR et MongoDB

| RDBMS | MongoDB |
|---|---|
| Database | Database |
| Table | Collection |
| Tuple/Row | Document |
| column | Field |
| Table Join | Embedded Documents |
| Primary Key | Primary Key (*) |

> **Note**
>
> (*): Clé primaire `_id` fournie par mongodb lui-même

## Les données sont des documents

- stocké en Binary JSON (BSON)
- documents similaires rassemblés dans des collections
- pas de schéma des documents définis en amont
    - contrairement à un BD relationnel ou NoSQL de type Column Store
- les documents peuvent n'avoir aucun point commun entre eux
- un document contient (généralement) l'ensemble des informations
    - pas (ou très peu) de jointure à faire

- BD respectant CP (dans le théorème CAP)
    - propriétés ACID au niveau d’un document

### Exemple de documents

```
{
 _id: ObjectId(7df78ad8902c)
 title: 'MongoDB Overview',
 description: 'MongoDB is no sql database',
 by: 'tutorials point',
 url: 'http://www.tutorialspoint.com',
 tags: ['mongodb', 'database', 'NoSQL'],
 likes: 100,
 comments: [
  {
   user:'user1',
   message: 'My first comment',
   dateCreated: new Date(2011,1,20,2,15),
   like: 0
  },
  {
   user:'user2',
   message: 'My second comments',
   dateCreated: new Date(2011,1,25,7,45),
   like: 5
  }
 ]
}
```

## Schéma dynamique

- Documents variant très fortement entre eux, même dans une même collection
- On parle de **self-describing documents**
- Ajout très facile d'un nouvel élément pour un document, même si cet élément est inexistant pour les autres
- Pas de `ALTER TABLE` ou de redesign de la base

## Langage d'interrogation

- Pas de SQL (bien évidemment), ni de langage proche
- Définition d'un langage propre (basé sur `JS`)
- Langage permettant plus que les accès aux données
    - définition de variables
    - boucles

    - …
- **Pas de jointures entre les collections**

# Installation sous Linux

Pour les systèmes Linux antérieurs 2018, la version opensource de MongoDB est fournie dans les packages officiels de la distribution :

```
$ apt-get update
$ apt-get install mongodb mongodb-clients
```

Pour les nouveaux systemes postérieurs à 2018 (licence non-opensource), il faut la Community Edition depuis les serveurs de MongoDB :

```
$ apt-get install gnupg
$ wget -qO - https://www.mongodb.org/static/pgp/server-5.0.asc \
   | apt-key add -
$ echo "deb http://repo.mongodb.org/apt/debian buster/mongodb-org/5.0 main" \
   | tee /etc/apt/sources.list.d/mongodb-org-5.0.list
$ apt-get update
$ apt-get install -y mongodb-org
$ systemctl start mongod
```

## Gestion du service Linux

```
sudo systemctl start mongodb
sudo systemctl stop mongodb
sudo systemctl restart mongodb
```

```
mongo
```

## Références

https://www.mongodb.com/docs/manual/tutorial/install-mongodb-on-debian/

# Installation sous MacOS

## Avec Homebrew

```
$ brew tap mongodb/brew
$ brew update
$ brew install mongodb-community@5.0
$ brew services start mongodb-community@5.0
```

# Installation sous Windows

## Avec chocolatey

```
C:\> choco update
C:\> choco install mongodb
```

## Installation manuelle

1. Télécharger mongodb sur le site officiel http://www.mongodb.org/downloads
2. Lancer l'exécutable
3. Cliquer sur suivant tant que nécessaire jusqu'à la fin de l'installation

**Important**

Les version 32 bits supportent uniquement des BDD < 2GB et ne sont pas utilisable en production

```
C:\> wmic os get osarchitecture
OSArchitecture
64-bit
```

# Installation avec Docker

## Utilisation de l'image officielle mongo

Pour le serveur

```
$ docker run -it --rm --name mongo -p 27017:27017 mongo
```

Pour le client (une fois le serveur lancé)

```
docker exec -it mongo bash
```

**Warning**

L'image docker utilise l'authentification par défaut. Il sera donc nécessaire d'y créer des utilisateurs pour l'utiliser.

# Après l'installation

## Créer le dossier des données

Sous Windows, il peut arriver que le dossier des données ne soit pas correctement créé et empêche le lancemente de mongo.

Pour corriger ce bug, ouvrir un terminal ( `powershell.exe` )

```
> mkdir C:\data\db
```

## Mongo dans le PATH

Assurez vous que

1. les commandes mongo(sh) et mongod sont bien installées
2. qu'elle sont correctement executables depuis tout chemin

Au cas échéant, ajoutez le chemin des binaires de mongo dans le PATH systeme.

# Authentification

## A savoir

- MongoDB ne force pas l'authentification, ni les autorisations par défaut
- Les modeles de roles de MongoDB sont relativement limités
- ... mais ça s'améliore

## Création d'un user pour le mode sécurisé

Lancer MongoDB sans controle d'accès

```
$ mongod --port 27017 --dbpath /var/lib/mongodb
```

Lancer la commande `mongosh`, puis taper :

```
> use admin
> db.createUser({
  user: 'admin',
  pwd: 'admin',
  roles: [{
    role: 'root',
    db: 'admin'
  }]
})
> exit
```

## Lancement de MongoDB en mode sécurisé

Eteindre MongoDB apres la création du user

```
> db.adminCommand( { shutdown: 1 } )
```

Relancer mongo avec l'option `--auth`

```
$ mongod --auth --port 27017 --dbpath /var/lib/mongodb
```

ou bien utiliser l'option `security.authorization` dans le fichier de configuration

```
security:
  authorization: enabled
```

## Authentification

Ensuite relancer le client mongo en s'authentifiant :

```
$ mongosh --authenticationDatabase XXX -u admin -p
```

(où XXX vaut `admin` ou `$external`)

## Références

- https://stackoverflow.com/questions/20117104/mongodb-root-user
- https://www.mongodb.com/docs/manual/reference/built-in-roles/
- https://www.mongodb.com/docs/manual/tutorial/configure-scram-client-authentication/

# Prise en main

## Concepts et outils

Database Server and Client

| RDBMS | MongoDB |
| --- | --- |
| Mysqld/Oracle | mongod |
| mysql/sqlplus | mongosh (anciennement mongo) |

## Obtenir de l'aide

Lister les sujet d'aide d'une base de données

```
> db.help()
```

Obtenir de l'aide sur une commande spécifique

```
> db.COMMANDE.help()
```

## Voir l'implémentation d'une commande

Taper le nom de la commande, sans les parentheses ()

```
> db.COMMANDE
```

A minima, cela retourne les propriétés de la commande souhaitée.

## Obtenir des statistiques

```
db.help()
```

# Bases de données

## Concept

- La base de données est un conteneur physique pour les collections
- Chaque base de données obtient son propre ensemble de fichiers sur système de fichiers
- Un serveur MongoDB possède généralement plusieurs bases de données

## Lister les bases de données

```
show databases
```

## Création d'une base de données

```
> use DATABASE_NAME
```

Exemple :

```
> use mydb
switched to db mydb
```

## Retrouver la base de donnée actuelle

```
> db
mydb
```

Pour lister toutes les BDD :

```
>show dbs
local   0.78125GB
test    0.23012GB
```

## Persistance d'une base de données

Pour qu'une bdd existe/persiste, il faut insérer au moins un document dedans.

```
> db.movie.insert({"name":"demo"})
> show dbs
local    0.78125GB
mydb     0.23012GB
test     0.23012GB
```

# MongoDB - Drop Database

```
> db.dropDatabase()
```

# Collections

## Concept

- `collection` représente la collection dans laquelle nous allons effectuer l'opération, et doit donc correspondre à une des collections présentes dans la base

## Notion de namespace

L'espace de nom est une combinaison des collections et du nom de la base de données

## Formalisme

Dans MongoDB, nous utilisons un formalisme de type `db.collection.fonction()` :

- `db` représente la base de données choisie grâce à la commande use (ce mot clé est non modifiable)
- `collection` représente la colletion dans laquelle nous allons effectuer l'opération, et doit donc correspondre à une des collections présentes dans la base
- `fonction()` détermine l'opération à effectuer sur la collection.

## Lister les collections

Soit

```
> show collections
```

Soit

```
> db.getCollectionsInfos()
```

## Créer une collection

```
> db.createCollection("movies")
```

## Lire une collection

Nous aborderons ce sujet dans la partie Recherche de documents.

## Mettre à jour une collection

```
> db.collection.update()
```

## Supprimer une collection

```
> db.movies.remove()
> db.movies.drop()
```

## Compter le nombre d'éléments

En premier lieu, on peut dénombrer le nombre de documents de chaque collection, grâce à la fonction count().

```
> db.restaurants.count()
```

## Reférences


- https://github.com/ozlerhakan/mongodb-json-files/tree/master/datasets
- https://www.mongodb.com/docs/manual/reference/method/db.collection.update/
- https://www.mongodb.com/docs/manual/reference/glossary/

# Documents

- Un document est un ensemble de paires clé-valeur.
- Les documents ont un schéma dynamique.

Le schéma dynamique signifie que :

- les documents d'une même collection ne doivent pas nécessairement avoir le même ensemble de champs ou la même structure,
- les champs communs des documents d'une collection peuvent contenir différents types de données.

## Types de données

### Primitifs

- **String**: This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
- **Integer**: This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean**: This type is used to store a boolean (true/ false) value.
- **Double**: This type is used to store floating point values.
- **Null**: This type is used to store a Null value.

## Types de données

### Objets

- **Min/Max Keys**: This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays**: This type is used to store arrays or list or multiple values into one key.
- **Timestamp**: ctimestamp. This can be handy for recording when a document has been modified or added.
- **Object**: This datatype is used for embedded documents.
- **Date**: This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.

# Types de données

## Complexes

- **Object ID**: This datatype is used to store the document’s ID.
- **Binary data**: This datatype is used to store binary data.
- Code: This datatype is used to store JavaScript code into the document.
- **Regular expression**: This datatype is used to store regular expression.
- **Symbol**: This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.

# Structure des documents

- Les documents présents dans une collection n’ont pas de schémas prédéfinis.
- Si nous souhaitons avoir une idée de ce que contient la collection, il est possible d’afficher un document (le premier trouvé), avec findOne(). Cette opération permet de comprendre la structure global d’un document, même s’il peut y avoir des différences entre documents.

  ```
  db.restaurants.findOne()
  ```

# Valeurs prises par les itemps

Une autre fonction très utile pour mieux appréhender les données est de lister les valeurs prises par les différents items de la collection, grâce à distinct(). Pour spécifier un sous-item d’un item, il est nécessaire d’utiliser le formalisme item.sousitem.

```
db.restaurants.distinct("borough")
db.restaurants.distinct("cuisine")
db.restaurants.distinct("address.zipcode")
db.restaurants.distinct("grades.grade")
```

# Insertion de documents

## Document unique

```
> db.COLLECTION_NAME.insert(document)
```

```
> db.mycol.insert({
    _id: ObjectId(7df78ad8902c),
    title: 'MongoDB Overview',
    description: 'MongoDB is no sql database',
    tags: ['mongodb', 'database', 'NoSQL'],
    likes: 100
})
```

Dans le shell mongosh, on utilisera plutot `insertOne(...)`

## Documents multiples

Il est possible d'insérer plusieurs documents en une fois en passant un array à la fonction `insertMany()`

```
> db.post.insertMany([{ ... }, { ... }])
```

Il est également possible d'utiliser la méthode `save()`. Si aucun `_id` n'est spécifié, elle se comportera de la même façon que `insert()`

## Import de données en masse

Depuis le shell (unix)

```
$ mongoimport \
    --authenticationDatabase admin \
    --username=XXX \
    --password=XXX \
    --db dbName \
    --collection collectionName \
    --file fileName.json \
    --jsonArray
```

## Références

- https://www.mongodb.com/docs/manual/reference/method/db.collection.insertMany/
- https://www.mongodb.com/docs/manual/reference/method/db.collection.insert/

# Recherche d’informations

## Requêtes de documents

Pour faire des recherches, on utilise la fonction `find()` et `findOne()`

- Sans paramètre, elle renvoie l’ensemble des documents.
  - Il faut donc l’utiliser avec précautions.
- Elle peut aussi prendre deux paramètres :
  - les critères de sélection des documents
  - les choix d’items des documents à afficher

Ces deux paramètres doivent être écrits sous la forme d’objets JSON.

```
> db.coll.find(..)
```

Exemple : recherche d'après deux critères

```
> db.coll.find({"cuisine": "Bakery", "borough": "Bronx"})
```

> **Note**
>
> Pour rendre le resultat plus plaisant à l'oeil on peut ajouter la fonction `pretty()` sur le résultat du `find()`
>
> ```
> > db.coll.find(..).pretty()
> ```

## Recherche sur une clé de sous-document

```
> db.coll.find({"address.zipcode": "10462"})
```

## Projection

Pour afficher seulement certains éléments, on peut ajouter un deuxième argument spécifiant les items que l’on veut (valeur 1) ou qu’on ne veut pas (avec valeur 0).

```
> db.restaurants.find({ cuisine: "French" }, { name: 1 })
```

Par défaut, l'identifiant du document, toujours nommé `_id`, est renvoyé. Pour ne pas l'avoir, il faut ainsi le préciser avec `"_id": 0`.

```
> db.restaurants.find({ cuisine: "French" }, { _id: 0, name: 1 })
```

## Sous-item

Les documents peuvent être complexes (c'est même le but), et les critères portent donc souvent sur des sous-items. Il faut utiliser le même formalisme déjà vu ("item.sousitem"). Il faut noter deux choses :

- on peut aller aussi loin que nécessaire dans l'utilisation du . ;
- il est obligatoire d'utiliser des '"'' pour les items.

```
db.restaurants.find({ "grades.grade" : "A" }, { _id: 0, name: 1 })
```

Ce formalisme est le même pour indiquer un sous-item à afficher. Nous ajoutons à la requête précédent l'affichage des scores. Nous voyons que le résultat inclu tous les scores.

```
> db.restaurants.find({ "grades.grade" : "A" }, { _id: 0, name: 1, "grades.grade" : 1 })
```

## Comparaisons avec d'autres opérateurs

| **Opération** | **Syntaxe** |
|---|---|
| Egalité | `{<key>: <value>}` |
| Inférieur à | `{<key>: {$lt: <value>}}` |
| Inférieur ou égal à | `{<key>: {$lte: <value>}}` |
| Supérieur à | `{<key>: {$gt: <value>}}` |
| Supérieur ou égal à | `{<key>: {$gte: <value>}}` |
| Différent de | `{<key>: {$ne: <value>}}` |

Exemples:

```
> db.coll.find({"by": "point"}).pretty()

// where age >= 18
> db.coll.find({"age": {$gte: 18}}).pretty()
```

# Opérateurs logiques

## ET logique

Il suffit de passer plusieurs clés séparées par ","

```
> db.coll.find({
   key1:value1,
   key2:value2
 }).pretty()
```

## OU logique

```
>db.coll.find({
   $or: [
     {key1: value1}, {key2:value2}
   ]
 }).pretty()
```

Ex:

```
>db.coll.find({
   $or: [
     {"by":"tutorials"},
     {"title": "MongoDB Overview"}
   ]
 }).pretty()
```

## NON OU logique

```
> db.coll.find({
   $nor: [
     {"by": "tutorials"},
     {"title": "MongoDB Overview"}
   ]
 }).pretty()
```

## NON logique

```
> db.coll.find({
   $not: {"by": "tutorials"}
  }).pretty()
```

### ET + OU logique

```
> db.coll.find({
   "likes": {$gt:10},
   $or: [
     {"by": "tutorials"},
     {"title": "MongoDB Overview"}
   ]
  }).pretty()
```

## Présence d’un item

Pour tester la présence ou non d’un item, on utilise l’opérateur `$exists` avec

- `true` si on teste la présence
- et `false` l’absence

```
> db.restaurants.find({ french : { $exists: true } }, { _id: 0, name: 1 })
```

## Dénombrement

La fonction `count()` peut s'ajouter à la suite d’une fonction `find()` pour connaître la taille du résultat.

```
> db.restaurants.find({ cuisine : "French" }).count()
```

# Filtrage des résultats

## Limitation du résultat

On peut aussi limiter le nombre de documents renvoyés par la fonction find() en lui ajoutant la fonction limit(), comme ici où nous nous restreignons aux 5 premiers résultats.

```
db.restaurants.find({ cuisine : "French" }, { _id: 0, name: 1 }).limit(5)
```

## Tri

- Réalisable avec la fonction `sort()` .
- On doit indiquer les items de tri
  - valeur de 1 pour un tri ascendant
  - valeur de -1 pour un tri descendant.

Exemple

```
> db.restaurants.find(
   { cuisine : "French" },
   { _id: 0, name: 1 }
 ).sort({ name: 1 })
```

Idem que précédemment, mais dans l'ordre décroissant.

```
> db.restaurants.find(
   { cuisine : "French" },
   { _id: 0, name: 1 }
 ).sort({ name: -1 })
```

Il est possible de mettre plusieurs critères de tri, en indiquant croissant ou décroissant pour chaque item de tri.

```
db.restaurants.find(
   { cuisine : "French" },
   { _id: 0, name: 1, borough: 1 }
).sort(
   { borough: -1, name: 1 }
)
```

# Agrégats

En plus des recherches classiques d'informations, le calcul d'agrégat est très utilisé, pour l'analyse, la modélisation ou la visualisation de données. Ce calcul s'effectue avec la fonction aggregate(). Celle-ci prend en paramètre un tableau d'opérations (appelé aussi pipeline), pouvant contenir les éléments suivants :

- $project : redéfinition des documents (si nécessaire)
- $match : restriction sur les documents à utiliser
- $group : regroupements et calculs à effectuer
- $sort : tri sur les agrégats
- $unwind : découpage de tableaux
- ...

Syntaxe

```
> db.coll.aggregate([
   {$group: {_id: <expression>, // Group By Expression
         <field1>: { <accumulator1> : <expression1> },
         ...}
   }
])
```

## Agrégat simple

Voici un premier exemple permettant un calcul pour toute la base. Ici, nous réalisons un dénombrement (il fait la somme de la valeur 1 pour chaque document).

```
db.restaurants.aggregate([
   { $group: { _id: null, nb: { $sum: 1 }}}
])
```

Les calculs peuvent être plus complexes, comme nous le verrons plus tard. Bien évidemment, ces calculs d'agrégats peuvent se faire aussi en ajoutant des critères de regroupement. Attention au $ avant le nom de l'item.

```
db.restaurants.aggregate([
   { $group: { _id: "$borough", nb: { $sum: 1 }}}
])
```

## Tri

Ce résultat peut être trié en ajoutant l'action $sort dans le tableau, avec le même mécanismes que précédemment (1 : ascendant, -1 : descendant).

```
db.restaurants.aggregate([
    { $group: { _id: "$borough", nb: { $sum: 1 }}},
    { $sort: { "nb": -1 }}
])
```

## Redéfinition des documents (et limite)

Il est parfois intéressant (voire nécessaire) de redéfinir les documents, pour ne garder que les items qui nous intéressent. Dans cette projection, il existe différentes fonctions de traitement comme la concaténation (cf ci-dessous).

L'action $limit permet de limiter le nombre de sorties renvoyées par le moteur.

```
db.restaurants.aggregate([
    { $project: { _id: 0, name: 1, info: { $concat: [ "$cuisine", " - ", "$borough" ]} }},
    { $limit: 5 }
]).pretty()
```

## Restriction

On peut aussi faire une restriction avant le calcul, avec l'opération $match.

```
db.restaurants.aggregate([
    { $match: { cuisine: "French" }},
    { $group: { _id: "$borough", nb: { $sum: 1 }}}
])
```

## Découpage des tableaux

Imaginons maintenant que nous souhaitons calculer le nombre de restaurant par grade. La première idée serait de réaliser l'opération suivante.

```
db.restaurants.aggregate([
    { $group: { _id: "$grades.grade", nb: { $sum: 1 }}}
])
```

Malheureusement, nous voyons que le regroupement se fait par ensemble de grades existant dans la base. Il faut donc faire un découpage des tableaux grades dans chaque document. Pour cela, il existe l'opération $unwind.

Pour montrer comment fonctionne cette opération, voici les 5 premières lignes renvoyées lorsqu'on l'applique directement sur les données. On s'apercoit que chaque document ne contient plus qu'une seule évaluation.

```
db.restaurants.aggregate([
    { $unwind: "$grades" },
    { $limit: 5 }
]).pretty()
```

Par ce biais, nous pouvons donc maintenant faire l'opération de regroupement par grade.

```
db.restaurants.aggregate([
    { $unwind: "$grades" },
    { $group: { _id: "$grades.grade", nb: { $sum: 1 }}}
])
```

## Calcul statistique

Comme indiqué précédemment, on peut faire tous les calculs d'agrégats classiques, comme ici avec la somme ($sum), la moyenne ($avg), le minimum ($min) et le maximum ($max).

```
db.restaurants.aggregate([
    { $unwind: "$grades" },
    { $group: {
        _id: null,
        nb: { $sum: 1 },
        scoreTot: { $sum: "$grades.score" },
        scoreMoy: { $avg: "$grades.score" },
        scoreMin: { $min: "$grades.score" },
        scoreMax: { $max: "$grades.score" }
    }}
]).pretty()
```

## Regroupement de valeurs

Une fois que les documents sont scindés en plusieurs suite à l'action $unwind, il peut être intéressant de regrouper les valeurs dans un tableau. Ici, nous cherchons les différents grades obtenus par chaque restaurant.

```
db.restaurants.aggregate([
    { $limit: 5 },
    { $unwind: "$grades" },
    { $group: { _id: "$name", nb: { $sum: 1 }, eval: { $addToSet: "$grades.grade" }}}
])
```

# Jointures

Il est possible de réaliser une jointure entre deux collections, dans un aggrégat, avec l'opérateur $lookup.

```
db.transactions_small.aggregate([
    { $match: { "horodateur": 57080603}},
    { $lookup: {
        from: "mobiliers",
        localField: "horodateur",
        foreignField: "fields.numhoro",
        as: "mobilier"
    }},
    { $limit: 5}
])
```

## Références

- https://www.mongodb.com/docs/manual/reference/operator/aggregation/lookup/#mongodb-pipeline-pipe.-lookup
- https://www.mongodb.com/docs/manual/reference/operator/aggregation/graphLookup/#mongodb-pipeline-pipe.-graphLookup

# Atomicité

- pas de transactions
- utiliser db.coll.findAndModify à la place

## Références

- https://www.mongodb.com/docs/manual/reference/method/db.collection.findAndModify/
- https://www.journaldev.com/6221/mongodb-findandmodify-example

# NodeJS

- MongoDB s'est rapidement développé pour devenir une base de données populaire pour les applications Web et convient parfaitement à Node.JS applications
- Nous avons beaucoup de modules tiers pour se connecter à MongoDB: Mongoose, MongoDB, MongoClient
- Nous devons d'abord établir une connexion entre l'application de noeud et MongoDB.
- Une fois la connexion établie, lancez la requête pour effectuer une opération CURD dans la base de données.

## Avec MongoClient

## Avec Mongoose (support des modèles)

### Vue d'ensemble

- Object modelling for MongoDB
- Define schemas in Node.js for MongoDB
- Query building (ODM)
- Mongoose: http://mongoosejs.com/

### Installation

```
$ npm install -g mongoose
```

```
var mongoose = require('mongoose')
```

### Utilisation

```
var express = require('express');
var mongoose = require('mongoose');
var app = express();

// Middleware
mongoose.connect("mongodb://localhost:27017/userManagement");
```

## Utilisation de schémas

### Définir le schéma

```
var movieSchema = new mongoose.Schema({
  title: String,
  rating: String,
  releaseYear: Number,
  likeIt: Boolean
});
```

### Utiliser la base de données:

```
var Movie = mongoose.model("Movie", movieSchema, "movies");

function showAllMovies() {
  Movie.find(function(err, movies)) {
    if (err) { return console.log(err); }

    movies.forEach(function(movie) {
      console.log(movie);
    });
  }
}
```

# Python

## Avec PyMongo

```
import pymongo
URI = 'mongodb+srv://user:user@cluster0.ougec.mongodb.net/test'
client = pymongo.MongoClient(URI) # enlever le paramètre URI si connexion locale
db = client.test
```

📚 https://pymongo.readthedocs.io/en/stable/

## Références

- https://fxjollois.github.io/cours-2021-2022/du-ads/python-mongodb.slides.html

# Modélisation de données

- Concevez votre schéma en fonction des besoins de l'utilisateur.
- Combinez des objets en un seul document si vous les utilisez ensemble. Autrement séparez-les (mais assurez-vous de ne pas avoir besoin de joindre).
- Dupliquez les données (mais limitées) car l’espace disque est bon marché par rapport au temps de calcul.
- Faire des jointures en écriture, pas en lecture.
- Optimisez votre schéma pour la plupart des cas d'utilisation fréquents.
- Faire une agrégation complexe dans le schéma.

## Exemple d'un site/blog

Le site Web a les exigences suivantes.

- Chaque message a le titre, la description et l'URL uniques.
- Chaque message peut avoir un ou plusieurs tags.
- Chaque message porte le nom de son éditeur et le nombre total de goûts.
- Chaque message a des commentaires donnés par les utilisateurs avec leur nom, message, données- temps et aime.
- Sur chaque post, il peut y avoir zéro ou plus de commentaires. Dans le schéma SGBDR, la conception pour les exigences ci-dessus aura au minimum trois tables.

## Exemple - avec une BDD relationnelle

## Exemple - avec une BDD NoSQL

```
{
  _id: POST_ID
  title: TITLE_OF_POST,
  description: POST_DESCRIPTION,
  by: POST_BY,
  url: URL_OF_POST,
  tags: [TAG1, TAG2, TAG3],
  likes: TOTAL_LIKES,
  // ...
}
```

# Exemple - avec une BDD NoSQL (suite)

```
{
 // ...,
 comments: [
   {
    user:'COMMENT_BY',
    message: TEXT,
    dateCreated: DATE_TIME,
    like: LIKES
   },
   {
    user:'COMMENT_BY',
    message: TEXT,
    dateCreated: DATE_TIME,
    like: LIKES
   }
 ]
}
```

## Création d'une base de données

`use DATABASE_NAME`

Exemple :

```
>use mydb
switched to db mydb
```

Pour retrouver la bdd actuelle :

```
>db
mydb
```

Pour lister toutes les BDD :

```
>show dbs
local   0.78125GB
test    0.23012GB
```

Pour qu'une bdd existe, il faut insérer au moins un document dedans.

```
>db.movie.insert({"name":"tutorials point"})
>show dbs
local    0.78125GB
mydb     0.23012GB
test     0.23012GB
```

# MongoDB - Drop Database

```
db.dropDatabase()
```

## Créer une collection

```
>use test
switched to db test
>db.createCollection("mycollection")
{ "ok" : 1 }
>
```

```
>show collections
mycollection
system.indexes
```

```
>db.createCollection("mycol", { capped : true, autoIndexID : true, size :
6142800, max : 10000 } )
{ "ok" : 1 }
>
```

N.B: mongodb crée une collection automatiuement lorsqu'on insere un document (voir plus haut)

```
> db.movie.insert({"name":"tutorials point"})
> show collections
```

## Suppression de collection

```
db.COLLECTION_NAME.drop()
```

## Example

First, check the available collections into your database mydb.

```
>use mydb
switched to db mydb
>show collections
mycol
mycollection
system.indexes
tutorialspoint
>
```

```
> db.mycollection.drop()
```

# Types de données

- **String**: This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
- **Integer**: This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean**: This type is used to store a boolean (true/ false) value.
- **Double**: This type is used to store floating point values.
- **Min/Max Keys**: This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays**: This type is used to store arrays or list or multiple values into one key.
- **Timestamp**: ctimestamp. This can be handy for recording when a document has been modified or added.
- **Object**: This datatype is used for embedded documents.
- **Null**: This type is used to store a Null value.
- **Symbol**: This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
- **Date**: This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID**: This datatype is used to store the document’s ID.
- **Binary data**: This datatype is used to store binary data.
- Code: This datatype is used to store JavaScript code into the document.
- **Regular expression**: This datatype is used to store regular expression.

## Insertion de document

```
>db.COLLECTION_NAME.insert(document)
```

```
>db.mycol.insert({
_id: ObjectId(7df78ad8902c),
title: 'MongoDB Overview',
description: 'MongoDB is no sql database',
by: 'tutorials point',
url: 'http://www.tutorialspoint.com',
tags: ['mongodb', 'database', 'NoSQL'],
```

```
likes: 100
})
```

N.B: il est possible d'insérer plusieurs documents en une fois en passant un array à la fonction `insert()`

```
>db.post.insert([{ ... }, { ... }])
```

N.B: il est également possible d'utiliser la méthode `save`. Si aucun `_id` n'est spécifié, elle se comportera de la même façon que `insert()`

## - Requêtes de documents

```
>db.COLLECTION_NAME.find()
```

Pour rendre le resultat sexy

```
>db.mycol.find().pretty()
```

## Équivalences avec les RDBMS : égalité

| **Opération** | **Syntaxe** |
|---|---|
| Egalité | `{<key>:<value>}` |
| Inférieur à | `{<key>:{$lt:<value>}}` |
| Inférieur ou égal à | `{<key>:{$lte:<value>}}` |
| Supérieur à | `{<key>:{$gt:<value>}}` |
| Supérieur ou égal à | `{<key>:{$gte:<value>}}` |
| Différent de | `{<key>:{$ne:<value>}}` |

```
// where by = 'tutorials point'
> db.mycol.find({"by":"tutorials point"}).pretty()
> db.mycol.find({"age":{$gte: 18}}).pretty()
```

## Composition : ET

Il suffit de passer plusieurs clés séparées par ","

```
>db.mycol.find({key1:value1, key2:value2}).pretty()
```

Il suffit de passer plusieurs clés séparées par ","

## Composition : OU

```
>db.mycol.find(
  {
    $or: [
      {key1: value1}, {key2:value2}
    ]
  }
).pretty()
```

Ex:

```
>db.mycol.find(
  {
    $or: [
      {"by":"tutorials point"},
      {"title": "MongoDB Overview"}
    ]
  }
).pretty()
```

## Composition : ET + OU

```
>db.mycol.find({"likes": {$gt:10}, $or: [{"by": "tutorials point"}, {"title": "MongoDB
Overview"}]}).pretty()
```

# Indexes et performances

## Création d'un index

Version 5.0

```
> db.coll.createIndex( { score: 1 } )
```

Versions antérieures

```
> db.coll.ensureIndex("tg" : 1)
```

### Sur un champ embarqué (sous-item)

```
> db.coll.createIndex( { "location.state": 1 } )
```

### Index composés

```
> db.coll.createIndex( { <field1>: <type>, <field2>: <type2>, ... } )
```

## Lister les index

```
> db.coll.getIndexes()
```

## Supprimer les index

```
> db.coll.dropIndex()
> db.coll.dropIndexes()
```

## Mesurer l'utilisation des index

```
> db.coll.aggregate( [ { $indexStats: {} } ] )
```

- Voir aussi le query plan avec `explain()`, `explain("queryPlanner")` ou `explain("executionStats")`
  - Noter le ratio : (executionStats.totalDocsExamined / executionStats.nReturned)
- Voir aussi `hint()` pour forcer l'utilisation d'un index

## Références

- https://www.mongodb.com/docs/manual/tutorial/measure-index-use/

# Validation des données

## Références

- https://www.mongodb.com/docs/manual/core/schema-validation/

# Introduction

## Deux types de backup

- Physiques
- Logiques

## Outils

MongoDB fournit deux utilitaires pour gérer les sauvegardes logiques : Mongodump et Mongorestore.

- La commande Mongodump décharge une sauvegarde de la base de données au format ".bson", et celle-ci peut être restaurée en fournissant les instructions logiques trouvées dans le fichier de décharge aux bases de données.
- La commande Mongorestore est utilisée pour restaurer les fichiers de vidage créés par Mongodump. La création de l'index se fait après la restauration des données.

## Références

- https://www.mongodb.com/basics/backup-and-restore
- https://www.mongodb.com/docs/manual/tutorial/backup-and-restore-tools/

# Sauvegarde

```
$ mongodump \
      -h sample.mongodbhost.com:27017 \
      -d DATABASE_NAME \
      -u USER_NAME \
      -p SAMPLE_PASSWORD \
      -o ~/Desktop
```

# Restauration

```
$ mongorestore \
    --host sample.mongohost.com \
    --port 27017 \
    --username USER_NAME \
    --password SAMPLE_PASSWORD \
    --db DATABASE_NAME
```

# Bonnes pratiques

- MongoDB utilise les formats de fichiers JSON et BSON (Binary JSON).
  - Il est préférable d'utiliser BSON lors des sauvegardes et des restaurations.
  - JSON ne prend pas en charge tous les types de données que BSON prend en charge, cela peut entraîner une non-conformité des données.
- Vous n'avez pas besoin de créer explicitement une base de données MongoDB,
  - Elle sera automatiquement créée lorsque vous spécifierez une base de données à importer.
  - De même, la structure d'une collection sera créée lorsque le premier document sera inséré dans la base de données.
- Utilisez des serveurs secondaires pour les sauvegardes
  - Cela permet d'éviter de dégrader les performances du nœud primaire
- Planifiez la sauvegarde des ensembles de données autour des périodes de faible bande passante/trafic.
  - Les sauvegardes peuvent prendre beaucoup de temps, surtout si les ensembles de données sont assez volumineux.
- Utilisez une connexion en replica set lorsque vous utilisez des scripts non supervisés
  - Une connexion unique échouera si l'hôte MongoDB visé s'avère indisponible

# Replication

## Guide

1. Start a mongod instance
2. Start another mongod instance
3. Start Replication (rs.initiate())
4. Add a MongoDB Instance to the Replica Set (rs.add())
5. Check Replication Status (rs.status())
6. Check Replication (insert documents)

## Authentication

Creation du keyfile

```
cd
openssl rand -base64 741 > mongodb.key
chmod 600 mongodb.key
```

mongodb.conf

```
replication:
  replSetName: rs0

security:
  authorization: enabled
  keyFile: /home/USERNAME/mongodb.key
```

## Références


- https://www.mongodb.com/docs/manual/tutorial/deploy-replica-set/
- https://www.mongodb.com/docs/manual/tutorial/deploy-replica-set-for-testing/
- https://www.mongodb.com/docs/manual/core/replica-set-high-availability/
- https://www.mongodb.com/docs/manual/reference/method/rs.initiate/
- https://www.mongodb.com/docs/manual/reference/method/rs.add/
- https://www.digitalocean.com/community/tutorials/how-to-configure-a-mongodb-replica-set-on-ubuntu-20-04
- https://www.tutorialspoint.com/mongodb/mongodb_replication.htm

- https://www.sohamkamani.com/docker/mongo-replica-set/
- StackOverflow: MongoDB replica set with simple password authentication

# Sharding

## Overview

A MongoDB sharded cluster consists of the following components:

> shard: Each shard contains a subset of the sharded data. As of MongoDB 3.6, shards must be deployed as a replica set.
> mongos: The mongos acts as a query router, providing an interface between client applications and the sharded cluster. Starting in MongoDB 4.4, mongos can support hedged reads to minimize latencies.
> config servers: Config servers store metadata and configuration settings for the cluster. As of MongoDB 3.4, config servers must be deployed as a replica set (CSRS).

## Références

- https://www.tutorialspoint.com/mongodb/mongodb_sharding.htm
- https://www.mongodb.com/docs/manual/core/sharded-cluster-components/

# index

## Mongo pour administrateurs

- Installation de mongoDB
- Configure MongoDB as a single instance deployment.
- Learn to configure a file based configuration file for Mongod.
- Activer l'authentification & les autorisations
- Mise en place d'un cluster de réplication w/ un arbitre
- Faire de commandes `rs.*`
- Différence entre `mongodump` et `mongoexport`
- Restauration d'une instance sur une autre

# Ajustement du Shell Mongo

## Personnalisation du shell Mongo

### Configuration du fichier .mongorc.js

- Créer un fichier .mongorc.js dans le répertoire utilisateur
  - `~/.mongorc.js` sur Linux et macOS,
  - `%USERPROFILE%\.mongorc.js` sur Windows
- Editer et ajouter des variables et fonctions personnalisées

```
// activer le profilage par défaut
db.setProfilingLevel(1)
```

### Modification de l'affichage et des couleurs

Vous pouvez personnaliser l'affichage du shell Mongo en utilisant des fonctions d'affichage et des codes de couleurs.

- Utiliser la fonction `print()` pour afficher du texte
- Utiliser `printjson()` pour un affichage formaté des objets JSON
- Personnaliser les couleurs avec la bibliothèque colors.js ou des codes d'échappement ANSI
- Exemple : `print("\x1b[31mHello World\x1b[0m")` pour afficher du texte en rouge

### Ajout de fonctions personnalisées

- Définir des fonctions dans .mongorc.js pour automatiser les tâches récurrentes

```
// Exemple: une fonction pour rechercher les documents par nom
function findByName(collection, name) {
```

```
    return db.getCollection(collection).find({ "name": name });
}
```

- Utiliser les fonctions personnalisées dans le shell Mongo

```
findByName("users", "John");
```

# Utilisation de scripts pour automatiser les tâches

## Exécution de scripts JavaScript

- Utiliser `load()` pour exécuter un script JS depuis le shell Mongo
  - Permet d'exécuter des fichiers JavaScript directement dans le shell Mongo
  - Facilite l'automatisation des tâches répétitives et l'exécution de scripts complexes

```
// File: script.js

var users = [
  { name: "Alice", age: 30 },
  { name: "Bob", age: 25 },
  { name: "Charlie", age: 35 }
];

users.forEach(function(user) {
  db.users.insert(user);
});

print("Utilisateurs insérés :");
db.users.find().forEach(printjson);
```

```
load("script.js")
```

## Chargement de scripts à partir de fichiers externes

- Inclure des scripts JS dans le fichier `.mongorc.js` avec `load()`
- Exemple : `load("/chemin/vers/script.js")` pour charger un script externe

# Trucs et astuces pour une utilisation efficace du shell Mongo

## Raccourcis clavier

- Utiliser les flèches haut/bas pour naviguer dans l'historique des commandes
- Utiliser Tab pour compléter automatiquement les noms de collections et de commandes

## Commandes couramment utilisées

- `show dbs` pour lister les bases de données
- `use <nom_db>` pour sélectionner une base de données
- `show collections` pour afficher les collections de la base de données sélectionnée
- `db.<collection>.find()` pour afficher les documents d'une collection
- `db.<collection>.find().pretty()` pour un affichage formaté des documents

# Manipulation efficace des opérations CRUD

## Insertions

### Insertion de documents individuels et multiples

- Utiliser `insertOne()` pour insérer un document individuel
- Utiliser `insertMany()` pour insérer plusieurs documents à la fois
- Exemple : `db.collection.insertOne({ field1: "value1", field2: "value2" })`

### Meilleures pratiques pour l'insertion de données

- Valider les données avant l'insertion
- Utiliser des identifiants uniques et significatifs pour le champ `_id`
- Insérer des documents de taille similaire pour éviter la fragmentation

## Requêtes

### Utilisation d'opérateurs avancés de requête

- `$gt`, `$gte`, `$lt`, `$lte` pour les comparaisons numériques
- `$in`, `$nin` pour vérifier si un champ est dans une liste de valeurs
- `$regex` pour les expressions régulières
- Exemple : `db.collection.find({ age: { $gte: 18, $lt: 65 } })`

### Filtrage, tri et pagination des résultats

- Utiliser `find()` pour filtrer les documents
- Utiliser `sort()` pour trier les résultats (1 pour croissant, -1 pour décroissant)
- Utiliser `skip()` et limit() pour la pagination
- Exemple : `db.collection.find({}).sort({ age: 1 }).skip(10).limit(10)`

### Projection de champs

- Utiliser la projection pour limiter les champs renvoyés

- Valeur `1` pour inclure un champ, `0` pour l'exclure
- Exemple : `db.collection.find({}, { field1: 1, field2: 0 })`

# Mises à jour

## Mise à jour de documents individuels et multiples

- Utiliser `updateOne()` pour mettre à jour un document individuel
- Utiliser `updateMany()` pour mettre à jour plusieurs documents à la fois
- Exemple : `db.collection.updateOne({ _id: id }, { $set: { field1: "newValue" } })`

## Utilisation d'opérateurs de mise à jour avancés

- `$set`, `$unset` pour ajouter ou supprimer des champs
- `$inc`, `$mul`, `$min`, `$max` pour modifier les valeurs numériques
- `$push`, `$pop`, `$addToSet`, `$pull` pour manipuler des tableaux
- Exemple : `db.collection.updateOne({ _id: id }, { $inc: { counter: 1 } })`

## Mises à jour atomiques

- Utiliser `findAndModify()` pour effectuer une mise à jour atomique
- Les opérations sont effectuées en une seule étape, sans risque de conflits
- Exemple : `db.collection.findAndModify({ query: { _id: id }, update: { $set: { field1: "newValue" } } })`

# Suppressions

## Suppression de documents individuels et multiples

- Utiliser `deleteOne()` pour supprimer un document individuel
- Utiliser `deleteMany()` pour supprimer plusieurs documents à la fois
- Exemple : `db.collection.deleteOne({ _id: id })`

## Meilleures pratiques pour la suppression de données

- Supprimer les données en fonction des besoins métier (date d'expiration, archivage, etc.)
- Utiliser des index pour accélérer les suppressions

- Planifier les suppressions pendant les périodes de faible charge pour minimiser l'impact sur les performances
- Utiliser les index TTL (Time-To-Live) pour supprimer automatiquement les documents après un certain temps

# Commandes d'administration utiles

## Création, modification et suppression d'utilisateurs

### Création d'utilisateurs

- Utiliser la commande `db.createUser()`
- Exemple : `db.createUser({ user: "techy", pwd: "password", roles: ["readWrite", "dbAdmin"] })`

### Modification d'utilisateurs

- Utiliser la commande `db.updateUser()`
- Exemple : `db.updateUser("techy", { roles: ["readWrite", "dbAdmin", "clusterAdmin"] })`

### Suppression d'utilisateurs

- Utiliser la commande `db.dropUser()`
- Exemple : `db.dropUser("techy")`

## Attribution et retrait de rôles

### Attribution de rôles

- Les rôles sont attribués lors de la création d'un utilisateur, comme montré précédemment.
- Pour mettre à jour les rôles d'un utilisateur existant, utilisez la commande `db.updateUser()` avec les nouveaux rôles, comme expliqué dans la section "Modification d'utilisateurs".

### Création de rôles

- Commande `db.createRole()`

```
db.createRole({
  role: "customRole",
  privileges: [
    {
      resource: { db: "exampleDB", collection: "exampleCollection" },
      actions: ["find", "update", "remove"]
    }
```

```
  ],

  // roles hérités
  roles: ["read"]
})
```

### Retrait de rôles

- Mettre à jour les rôles d'un utilisateur avec `db.updateUser()`
- Spécifier uniquement les rôles à conserver.

## Opérations de maintenance

### Compactage de collections

- Utiliser la commande `db.runCommand({ compact: "collectionName" })`
- Exemple : `db.runCommand({ compact: "products" })`
- À noter : opération bloquante pour la collection concernée

### Vérification de l'intégrité de la base de données

- Utiliser la commande `db.runCommand({ validate: "collectionName" })`
- Exemple : `db.runCommand({ validate: "products" })`
- Vérifier les erreurs et les avertissements dans le résultat

## Surveillance et diagnostic

### Utilisation de la commande db.stats()

- Obtenir des statistiques sur l'utilisation de la base de données
- Exemple : `db.stats()`
- Analyser les résultats pour déterminer l'utilisation de la mémoire, de l'espace disque, etc.

### Vérification de l'état des index et des performances

- Utiliser la commande `db.collection.getIndexes()`
- Exemple : `db.products.getIndexes()`
- Examiner les index pour déterminer leur utilisation et leur impact sur les performances

- Utiliser la commande `db.collection.explain("executionStats").find(query)`
- Exemple : `db.products.explain("executionStats").find({ category: "electronics" })`
- Analyser les statistiques d'exécution pour optimiser les requêtes et les index

# Outils de supervision intégrés

## Introduction à mongotop

- Outil de supervision en ligne de commande
  - mongotop est un outil intégré à MongoDB, accessible via la ligne de commande
  - Il s'exécute directement sur l'ordinateur hébergeant le serveur de base de données
  - Aucune installation supplémentaire n'est requise, car mongotop est inclus dans le package MongoDB
- Surveille les opérations de lecture et d'écriture par collection
  - mongotop surveille en temps réel les opérations effectuées sur chaque collection de la base de données
  - Il donne un aperçu du temps passé en lecture et en écriture pour chaque collection
  - Les informations sont affichées sous forme d'un tableau avec des colonnes pour les lectures, les écritures et le temps total
- Affiche le temps passé sur chaque opération
  - mongotop affiche le temps moyen passé en millisecondes pour chaque opération de lecture et d'écriture par collection
  - Les résultats sont mis à jour en temps réel à intervalles réguliers (par défaut, toutes les secondes)
- Utile pour identifier les collections les plus sollicitées
  - mongotop aide à détecter les collections qui sont fréquemment sollicitées
  - Les collections ayant des temps de lecture et d'écriture élevés peuvent nécessiter une optimisation (par exemple, l'ajout d'index)

Exemple:

```
$ mongotop --host localhost:27017 --username myUser --password myPassword
        ns    total    read    write
test.coll1      0ms     0ms      0ms
test.coll2      8ms     8ms      0ms
test.coll3     15ms     0ms     15ms
```

# Utilisation de mongotop pour surveiller les opérations

- Paramètres optionnels : intervalle de rafraîchissement (en ms) et authentification
  - Intervalle de rafraîchissement : mongotop
  - Authentification : mongotop -u -p --authenticationDatabase
  - Exemple : mongotop 1000 -u admin -p mypassword --authenticationDatabase admin

- Interprétation des résultats : temps de lecture (read), écriture (write) et total par collection :
  - Les résultats de mongotop sont présentés sous forme de tableau avec les colonnes suivantes : ns, total, read, write.
  - ns : le nom de la collection (par exemple, mydb.mycollection)
  - total : le temps total (en ms) passé sur les opérations de lecture et d'écriture pour cette collection
  - read : le temps total (en ms) passé sur les opérations de lecture pour cette collection
  - write : le temps total (en ms) passé sur les opérations d'écriture pour cette collection

- Identifier les collections à optimiser
  - Analyser les temps de lecture et d'écriture
  - Repérer les collections avec des temps élevés
  - Exemple : dans la sortie ci-dessus, la collection "orders" présente des temps de lecture et d'écriture plus élevés que les autres collections

# Introduction à mongostat

- Outil de supervision en ligne de commande
  - Utilise la CLI pour interagir avec MongoDB
  - Commande : mongostat
- Surveille les statistiques globales de la base de données
  - Opérations de lecture, écriture, et mise à jour par seconde
  - Utilisation de la mémoire

    - Taux de lock
    - Nombre de connexions
- Affiche les informations sur les opérations, l'utilisation de la mémoire, la réplication, etc.
    - "insert" : nombre d'opérations d'insertion par seconde
    - "query" : nombre de requêtes par seconde
    - "update" : nombre d'opérations de mise à jour par seconde
    - "delete" : nombre d'opérations de suppression par seconde
    - "resident" (res) : mémoire résidente utilisée (en Mo)
    - "virtual" (virt) : mémoire virtuelle utilisée (en Mo)
    - "netIn" et "netOut" : taux d'entrée/sortie réseau (en Ko/s)
    - "conn" : nombre de connexions actives
    - "repl" : état de la réplication (PRIMARY, SECONDARY, etc.)
- Aide à détecter les problèmes de performance et de configuration
    - Identifier les goulots d'étranglement (bottlenecks) ou les ressources saturées
    - Surveiller la réplication et l'état du cluster

Exemple:

```
$ mongostat
insert  query update delete getmore command dirty  used flushes vsize   res qrw arw net_in
net_out conn  time
  *0     *0     *0     *0       0     1|0  0.0%  0.1%       0 1.44G 13.0M 0|0 1|0  62b    125b   10
12:34:56
```

## Utilisation de mongostat pour surveiller les statistiques de la base de données

- Lancer mongostat depuis la CLI avec la commande mongostat
    - Exemple : mongostat
    - Affiche les statistiques en temps réel avec un intervalle de rafraîchissement par défaut de 1 seconde
- Paramètres optionnels : intervalle de rafraîchissement (en s) et authentification
    - Exemple : mongostat --authenticationDatabase admin -u yourUsername -p yourPassword --host yourHost 5

    - Ici, on spécifie un intervalle de 5 secondes et on fournit les informations d'authentification
    - Exemple : mongostat 1 pour un intervalle d'1 seconde
- Interprétation des résultats : opérations par seconde, utilisation de la mémoire, nombre de connexions, etc.
    - insert, query, update, delete : opérations de la base de données par seconde
    - command : nombre de commandes exécutées par seconde
    - dirty, used : pourcentage de la mémoire utilisée par les données modifiées et par les données en général
    - vsize, res : taille de la mémoire virtuelle et de la mémoire résidente utilisées par le processus mongod
    - netIn, netOut : débit réseau entrant et sortant en octets par seconde
    - conn : nombre de connexions actives à la base de données
- Analyser les indicateurs pour détecter les problèmes et optimiser la base de données
    - Surveiller les opérations avec un taux élevé pour identifier les collections très sollicitées
    - Vérifier l'utilisation de la mémoire pour détecter les problèmes de gestion de la mémoire
    - Surveiller le débit réseau pour identifier les goulots d'étranglement liés au réseau
    - Examiner le nombre de connexions pour s'assurer que la base de données n'est pas surchargée

# Mémoire et performances des E/S

## Mémoire et caches dans MongoDB

### Architecture de la mémoire MongoDB

- Cache de requêtes
  - Stocke les plans d'exécution des requêtes récentes
  - Améliore les performances des requêtes répétées
  - Exemple : Utiliser db.collection.find(query).explain("executionStats") pour voir le plan d'exécution
- Journal
  - Stocke les opérations de lecture et d'écriture
  - Assure la durabilité des données
  - Exemple : Configurer la fréquence de journalisation avec `--journalCommitInterval`
- Cache des données
  - Stocke les données fréquemment utilisées en mémoire
  - Permet un accès rapide aux données
  - Exemple : Utiliser `db.serverStatus().wiredTiger.cache` pour vérifier les statistiques du cache

### Importance de la gestion de la mémoire

- Amélioration des performances
  - Les données en cache réduisent les accès disque
  - Diminution des temps de réponse
  - Exemple : Configurer la taille du cache avec `storage.wiredTiger.engineConfig.cacheSizeGB`
- Éviter les problèmes de contention
  - Contention de la mémoire : concurrence pour l'accès à la mémoire
  - Peut causer des problèmes de performance
  - Exemple : Surveiller la contention avec `db.serverStatus().globalLock`

# Gestion de la mémoire virtuelle

## Mémoire virtuelle : définition

- Espace disque pour les données en mémoire
- Extension de la mémoire physique
- Comprend la mémoire RAM et l'espace disque

## Rôle de la mémoire virtuelle dans MongoDB

- Stockage des données
  - Utilisation de la mémoire RAM pour accélérer l'accès aux données
- Gestion du cache
  - Conserver les données fréquemment utilisées en mémoire pour optimiser les performances

## Suivi de la mémoire virtuelle

- Commande `db.serverStatus()`
  - Exemple de commande shell : mongo --eval "db.serverStatus()"
- Champ `mem.virtual`
  - Indique la quantité de mémoire virtuelle utilisée

```
{
  "mem": {
    "virtual": 2048
  }
}
```

## Optimisation de la mémoire virtuelle

- Réglage du système d'exploitation
  - Ajuster les paramètres de mémoire virtuelle du système d'exploitation (ex. Linux sysctl)
  - Exemple de configuration Linux :

```
# /etc/sysctl.conf
vm.swappiness = 1
```

- Limitation de la mémoire allouée
  - Restreindre la quantité de mémoire virtuelle utilisée par MongoDB
  - Exemple de configuration MongoDB :

```
# /etc/mongod.conf
processManagement:
  fork: true
systemLog:
  destination: file
  path: "/var/log/mongodb/mongod.log"
storage:
  dbPath: "/var/lib/mongo"
net:
  bindIp: "127.0.0.1"
  port: 27017
setParameter:
  wiredTigerCacheSizeGB: 4
```

# Mesure des performances d'E/S

## Importance des performances d'E/S

- Lecture/écriture des données
  - Crucial pour les opérations courantes
  - Impact direct sur l'expérience utilisateur
- Impacts sur la vitesse et la stabilité
  - Temps de réponse réduit
  - Moins de risques de blocage ou de crash

## Outils pour mesurer les performances d'E/S

- Commandes MongoDB :
  - `db.stats()` : statistiques globales de la base de données
  - `db.<collection>.stats()` : statistiques d'une collection spécifique
- Outils système : `iostat`, `vmstat`
  - `iostat` : statistiques d'utilisation des disques Exemple : `iostat -x 1`

- `vmstat` : statistiques de la mémoire virtuelle Exemple : `vmstat 1`

### Analyse des indicateurs de performance

- Taux d'E/S
  - Nombre d'opérations d'entrée/sortie par seconde
  - Indicateur clé pour évaluer les performances
- Latence
  - Temps écoulé entre l'émission d'une requête et la réception de la réponse
  - Plus la latence est faible, meilleures sont les performances
- Taux d'erreurs
  - Proportion d'opérations d'E/S ayant échoué
  - Un taux d'erreurs élevé peut indiquer des problèmes matériels ou de configuration

## Optimisation des performances d'E/S

### Techniques d'optimisation

- Répartition des données (sharding)
  - Distribue les données sur plusieurs serveurs
  - Équilibre la charge et réduit la contention
- Indexation appropriée
  - Crée des index pour accélérer les requêtes fréquentes
  - Évite les scans de collection complets
- Compression des données
  - Réduit l'espace disque et améliore les performances d'E/S
  - Configurable avec WiredTiger

### Configuration du matériel

- Disques SSD
  - Performances d'E/S supérieures aux disques durs traditionnels
  - Faible latence et temps d'accès rapides

- RAID
  - Combine plusieurs disques pour améliorer les performances et la tolérance aux pannes
  - RAID 10 recommandé pour MongoDB
- Mémoire suffisante
  - Permet de stocker davantage de données en cache
  - Réduit les accès disque

## Réglages MongoDB

- Moteur de stockage WiredTiger
  - Utilisé par défaut depuis la version 3.2
  - Performances supérieures à MMAPv1
  - Gère la compression des données et l'isolation des transactions
  - Activer WiredTiger : `--storageEngine wiredTiger`
- `storage.wiredTiger.engineConfig.cacheSizeGB`
  - Configure la taille du cache WiredTiger
  - Ajuster en fonction de la taille de la mémoire
  - Exemple de configuration : `--wiredTigerCacheSizeGB 8`

📚 Percona Blog: MMAPv1 VS WiredTiger

# Requêtes sous-optimales et profiling

## Comprendre les requêtes sous-optimales

### Requêtes sous-optimales

- Définition : requêtes lentes ou consommant trop de ressources
- Requêtes avec des scans de collection entière (table scans)
- Requêtes avec des jointures mal optimisées
- Requêtes avec des filtres, des tris ou des agrégations inefficaces
- Exemple de requête lente sans index : db.collection.find({age: {$gte: 18}})

### Impact sur les performances globales de la base de données

- Ralentissement de l'exécution des autres requêtes
- Augmentation de la charge CPU et de l'utilisation de la mémoire
- Diminution de la capacité à répondre aux demandes en temps réel

### Nécessité d'identifier et d'optimiser ces requêtes

- Amélioration des performances globales de la base de données
- Réduction de la consommation de ressources (CPU, mémoire, disque)
- Meilleure expérience pour les utilisateurs finaux

## Activer et configurer le profileur de requêtes

### Activation du profileur :

- Utilisez la méthode `db.setProfilingLevel(level, options)` pour activer le profileur
- `level` : niveau de profilage souhaité
- `options` : paramètres supplémentaires, tels que slowms

### Niveaux de profilage

- `0` : désactivé, aucune requête n'est enregistrée
- `1` : activé pour les requêtes lentes, seulement les requêtes dépassant le seuil slowms sont enregistrées

- `2` : activé pour toutes les requêtes, toutes les requêtes sont enregistrées

### Options : seuil de durée en ms pour le niveau 1

- slowms : seuil de durée en millisecondes pour déterminer si une requête est considérée comme lente
- Exemple : db.setProfilingLevel(1, {slowms: 200})

### Exemple

```
> db.setProfilingLevel(1, {slowms: 100})
{ "was" : 0, "slowms" : 100, "sampleRate" : 1, "ok" : 1 }
```

📚 https://www.mongodb.com/docs/manual/tutorial/manage-the-database-profiler/

📚 https://www.mongodb.com/docs/manual/reference/method/db.setProfilingLevel/

📚 https://studio3t.com/knowledge-base/articles/mongodb-query-performance/

## Analyser les résultats du profileur

### Récupération des données

- Commande : `db.system.profile.find().pretty()`
- Affiche les entrées du profileur dans un format lisible

### Informations clés : opération, durée, index utilisés, nombre de documents examinés

- Opération : type d'opération (ex : query, update, insert, remove)
- Durée : temps d'exécution de l'opération en millisecondes
- Index utilisés : index exploités lors de l'exécution de la requête
- Nombre de documents examinés : quantité de documents traités pour répondre à la requête

Exemple

```
{
   "op" : "query",
   "ns" : "testDB.products",
   "query" : { "price" : { "$gte" : 100 } },
   "cursorid" : NumberLong("4512952734067705982"),
   "keysExamined" : 0,
```

```
    "docsExamined" : 2000,
    "nreturned" : 50,
    "responseLength" : 10234,
    "millis" : 120,
    "planSummary" : "COLLSCAN",
    "execStats" : { ... },
    "ts" : ISODate("2023-04-17T12:00:00.000Z"),
    "client" : "192.168.0.100",
    "allUsers" : [ ],
    "user" : ""
}
```

### Analyse des données pour déterminer les requêtes sous-optimales

- Rechercher les entrées avec une longue durée (ex : > 100 ms) `db.system.profile.find({ "millis": { "$gt": 100 } }).pretty()`
- Identifier les requêtes sans index (ex : `"planSummary" : "COLLSCAN"` au lieu de `"IDHACK"` ) `db.system.profile.find({ "planSummary": "COLLSCAN" }).pretty()`
- Requêtes avec un grand nombre de documents examinés `db.system.profile.find({ "docsExamined": { "$gt": 1000 } }).pretty()`
- Requete combinée `db.system.profile.find({ "millis": { "$gt": 100 }, "planSummary": "COLLSCAN" }).pretty()`

Exemple (pour la requête ci-dessus)

- elle n'utilise pas d'index
- elle examine 2000 documents
- elle prend un temps d'exécution de 120 ms,
- ... tout cela indique une requête sous-optimale!

## Optimisation des requêtes

### Utilisation d'index appropriés pour accélérer les requêtes

- Création d'index : `db.collection.createIndex(keys, options)`
- Exemple d'index simple : `db.users.createIndex({age: 1})`
- Exemple d'index composé : `db.users.createIndex({age: 1, city: 1})`
- Importance de choisir les bons index pour les requêtes fréquentes
- Surveiller les performances des index avec `db.collection.aggregate([{$indexStats: {}}])`

## Réécriture des requêtes pour améliorer les performances

- Utiliser des opérateurs efficaces, tels que `$in` et `$elemMatch`
- Exemple de requête optimisée avec `$in` : `db.users.find({city: {$in: ["Paris", "London"]}})`
- Préférer l'usage de `$and` au lieu de `$or` lorsque cela est possible
- Exemple de requête optimisée avec `$and` : `db.users.find({$and: [{age: {$gt: 25}}, {city: "Paris"}]})`
- Éviter les opérations de tri coûteuses en utilisant des index

## Limitation des champs renvoyés pour réduire la charge de la bande passante

- Utiliser le second argument de la méthode find pour spécifier les champs à renvoyer
- Exemple : `db.users.find({age: {$gt: 25}}, {name: 1, city: 1, _id: 0})`
- Utiliser `1` pour inclure un champ, et `0` pour l'exclure (sauf pour `_id` qui est inclus par défaut)
- Renvoyer uniquement les données nécessaires pour réduire la charge sur le réseau et le serveur

# Moteurs de stockage

## Présentation des moteurs de stockage

### Deux moteurs de stockage principaux : MMAPv1 et WiredTiger

- MMAPv1 : moteur de stockage par défaut avant MongoDB 3.0
- WiredTiger : moteur de stockage par défaut depuis MongoDB 3.2

### Importance du choix du moteur de stockage pour les performances et les fonctionnalités

- Impacts sur les temps de réponse des requêtes
- Impacts sur les fonctionnalités supportées (transactions, compression des données)
- Considérations pour les déploiements existants et les migrations

### Vérifier le moteur de stockage utilisé

Pour vérifier le moteur de stockage actuel :

```
use admin;
db.runCommand({ serverStatus: 1 });
// ou bien
db.serverStatus().storageEngine;
```

Exemple de sortie pour WiredTiger :

```
"storageEngine" : {
    "name" : "wiredTiger",
    "supportsCommittedReads" : true,
    "persistent" : true
}
```

Exemple de sortie pour MMAPv1 :

```
"storageEngine" : {
    "name" : "mmapv1",
    "supportsCommittedReads" : false,
    "persistent" : true
}
```

# MMAPv1 : caractéristiques et cas d'utilisation

## MMAPv1 : premier moteur de stockage développé pour MongoDB

- Moteur de stockage par défaut avant MongoDB 3.0
- Remplacé par WiredTiger à partir de MongoDB 3.2

## Architecture basée sur les fichiers mmap

- Utilise la mémoire mappée sur des fichiers pour lire et écrire les données
- Le système d'exploitation gère le cache et les entrées/sorties disque
- Les fichiers de données ont l'extension .mmap

## Verrouillage au niveau de la collection

- MMAPv1 gère les verrous de lecture et d'écriture au niveau de la collection
- Accepte moins de concurrence que le verrouillage au niveau du document (WiredTiger)
- Exemple : Si une opération d'écriture est en cours sur une collection, les autres opérations d'écriture doivent attendre la fin de la première opération

## Compression des données : non disponible

- MMAPv1 ne prend pas en charge la compression des données stockées sur le disque
- Les données sont stockées telles quelles, sans compression

## Transactions : non supportées

- MMAPv1 ne prend pas en charge les transactions multi-documents
- Les opérations atomiques sont supportées seulement au niveau du document

## Cas d'utilisation : applications nécessitant une compatibilité avec les anciennes versions de MongoDB

- Exemple : Migration d'une base de données MongoDB 2.6 vers une version plus récente sans changer de moteur de stockage
- Exemple : Applications ayant des contraintes de compatibilité avec des bibliothèques ou des pilotes spécifiques à MMAPv1

Exemple de commande pour démarrer MongoDB avec le moteur de stockage MMAPv1 :

```
mongod --dbpath /data/db --storageEngine mmapv1
```

# WiredTiger : caractéristiques et cas d'utilisation

## Moteur de stockage par défaut depuis MongoDB 3.2

- Pour vérifier le moteur de stockage utilisé : db.serverStatus().storageEngine
- Pour configurer WiredTiger au démarrage de MongoDB : mongod --storageEngine wiredTiger

## Utilise des fichiers de données avec des extensions wt

- Les fichiers sont stockés dans le répertoire de données (par défaut : /data/db)
- Exemple de fichier : collection-0--123456789.wt

## Verrouillage au niveau du document

- Permet des opérations de lecture et d'écriture concurrentes sur différents documents
- Améliore les performances en réduisant la contention entre les opérations

## Compression des données : disponible (Snappy, zlib, zstd)

- Compression par défaut : Snappy (équilibré entre vitesse et taux de compression)
- Configurer la compression pour une collection :

```
db.createCollection("ma_collection", {
   storageEngine: {
      wiredTiger: {
         configString: "block_compressor=zlib"
      }
   }
});
```

## Transactions : supportées depuis MongoDB 4.0

```
session = db.getMongo().startSession();
session.startTransaction();
```

```
try {
    session.getDatabase("test").getCollection("ma_collection").insertOne({x: 1});
    session.getDatabase("test").getCollection("ma_collection").updateOne({x: 1}, {$set: {y:
2}});
    session.commitTransaction();
} catch (error) {
    session.abortTransaction();
    print("Transaction échouée :", error);
} finally {
    session.endSession();
}
```

### Cas d'utilisation

- Applications nécessitant de meilleures performances, une compression des données et des transactions
- Applications à fort trafic avec des opérations de lecture et d'écriture concurrentes
- Applications nécessitant de réduire l'espace disque utilisé grâce à la compression
- Applications nécessitant d'assurer la cohérence des données à travers plusieurs opérations avec des transactions

## Choisir le bon moteur de stockage

### Évaluer les besoins de l'application

- Identifier les besoins en termes de performances : lecture, écriture, mises à jour
- Déterminer si l'application nécessite des transactions multi-documents
- Évaluer le besoin de compression des données pour économiser de l'espace disque

### Prendre en compte la compatibilité avec les versions précédentes de MongoDB

- MMAPv1 pour les applications utilisant des versions antérieures à MongoDB 3.0
- WiredTiger pour les applications utilisant MongoDB 3.2 ou ultérieure

## Comparer les performances des deux moteurs avec des charges de travail réalistes

- Créer des scénarios de test représentatifs des opérations de l'application
- Exemple de commande pour comparer les performances en lecture : `db.collection.find(query).explain("executionStats")`
- Comparer les temps d'exécution, les ressources utilisées et l'utilisation de la mémoire

## Changer de moteur de stockage : utiliser mongodump et mongorestore avec l'option --storageEngine

- Sauvegarder les données avec mongodump : `mongodump --out /path/to/backup`
- Arrêter le serveur MongoDB : `sudo systemctl stop mongod`
- Modifier le fichier de configuration `/etc/mongod.conf` pour changer le moteur de stockage (par exemple, passer de MMAPv1 à WiredTiger)

```
storage:
  engine: wiredTiger
```

- Redémarrer le serveur MongoDB : `sudo systemctl start mongod`
- Restaurer les données avec mongorestore et l'option `--storageEngine` :

```
mongorestore --storageEngine wiredTiger /path/to/backup
```

# Les Explainable objects

## Introduction aux Explainable objects

### Présentation du concept d'Explainable objects

- Outils pour analyser et comprendre le fonctionnement des requêtes
- Aide à l'optimisation des performances de la base de données

### Objectif : analyser et optimiser les performances des requêtes

- Détection des goulots d'étranglement et des requêtes coûteuses
- Mise en place de solutions pour améliorer les performances

### Fonctionnement : obtenir des informations détaillées sur le plan d'exécution de requêtes

- Analyse du plan d'exécution généré par MongoDB pour chaque requête
- Informations sur les index utilisés, les documents scannés, etc.

### Méthode : utilisation de la méthode .explain() dans la CLI MongoDB

- Utilisation directe sur les requêtes, par exemple : `db.collection.find({...}).explain()`
- Résultats sous forme d'un document JSON détaillé contenant
  - le plan d'exécution de la requête,
  - les index utilisés,
  - le nombre de documents scannés,
  - le temps d'exécution, etc.

Exemple

```
// Exemple de requête
db.users.find({age: {$gte: 18}})

// Utilisation de la méthode .explain()
db.users.find({age: {$gte: 18}}).explain()
```

# Utilisation des Explainable objects pour analyser les requêtes

## Utilisation de la méthode .explain() sur les requêtes

- Permet d'obtenir des informations détaillées sur le plan d'exécution
- Aide à identifier les problèmes de performance et les index manquants ou mal utilisés

## Exemple de code : db.collection.find({...}).explain()

```
db.products.find({category: "Electronics", price: {$gte: 100}}).explain()
```

## Résultats : informations sur les index utilisés, nombre de documents scannés, temps d'exécution, etc.

- `"queryPlanner"` : informations sur le plan d'exécution et les index utilisés
- `"executionStats"` : statistiques détaillées sur le temps d'exécution, les documents scannés, etc.
- `"serverInfo"` : informations sur le serveur et la version de MongoDB

## Analyse des résultats pour détecter les problèmes de performances

- Identifier les index non utilisés ou inadaptés
- Repérer les étapes coûteuses du plan d'exécution
- Exemple : si `"totalDocsExamined"` est élevé, cela peut indiquer un problème d'indexation

Exemple

```
var explainResult = db.products.find({category: "Electronics", price: {$gte:
100}}).explain();
print("Index utilisé : ", explainResult.queryPlanner.winningPlan.inputStage.indexName);
print("Documents examinés : ", explainResult.executionStats.totalDocsExamined);
print("Temps d'exécution (ms) : ", explainResult.executionStats.executionTimeMillis);
```

# Comprendre les plans d'exécution

## Importance de comprendre les plans d'exécution pour optimiser les requêtes

- Augmenter les performances des requêtes
- Réduire le temps de réponse
- Diminuer l'utilisation des ressources système

## Structure du plan d'exécution : étapes, coût, index utilisés, etc.

- `winningPlan:` plan d'exécution choisi par MongoDB
- `rejectedPlans:` autres plans d'exécution envisagés mais rejetés
- `inputStage:` étape initiale de la requête
- `stage:` type d'opération (e.g. IXSCAN, FETCH, SORT)
- `indexName:` nom de l'index utilisé
- `nReturned:` nombre de documents retournés
- `executionTimeMillis:` temps d'exécution en millisecondes

## Identification des parties coûteuses du plan d'exécution

- Analyser executionTimeMillis pour détecter les étapes lentes
- Examiner nReturned pour identifier les requêtes inefficaces
- Vérifier inputStage et les opérations stage pour repérer les optimisations potentielles

## Utilisation des informations pour améliorer les performances de la requête

- Adapter les index en fonction des étapes coûteuses
- Modifier la requête pour utiliser les index optimisés
- Comparer les plans d'exécution avant et après optimisation

## Exemple

Exécuter une requête avec .explain() pour afficher le plan d'exécution :

```
> db.collection.find({ field1: value1, field2: value2 }).explain()
{
```

```
  "queryPlanner": {
    "winningPlan": {
      "stage": "FETCH",
      "inputStage": {
        "stage": "IXSCAN",
        "keyPattern": { "field1": 1 },
        "indexName": "field1_1",
        ...
      },
      ...
    },
    "rejectedPlans": [...],
    ...
  },
  "executionStats": {
    "nReturned": 10,
    "executionTimeMillis": 120,
    ...
  },
  ...
}
```

Créer un nouvel index pour améliorer les performances :

```
db.collection.createIndex({ field1: 1, field2: 1 })
```

Réexécuter la requête avec .explain() pour vérifier l'amélioration des performances :

```
db.collection.find({ field1: value1, field2: value2 }).explain()
```

# MongoDB Cloud Manager et Munin

## Présentation de MongoDB Cloud Manager

### Service en ligne de gestion et de monitoring de MongoDB

### Surveillance en temps réel des performances et des ressources

### Alertes personnalisables pour les événements importants

### Sauvegardes automatisées et restaurations point-in-time

### Gestion des déploiements de réplication et de sharding

## Configuration et utilisation de MongoDB Cloud Manager

### Création d'un compte et connexion à MongoDB Cloud Manager

### Ajout d'une organisation et d'un projet

### Installation de l'agent Cloud Manager sur les serveurs MongoDB

- Téléchargement de l'agent et obtention de l'API key
- Configuration de l'agent avec l'API key et l'URL du projet
- Démarrage de l'agent

### Visualisation des données de monitoring dans l'interface Cloud Manager

- Métriques clés : opérations, latence, utilisation du disque, etc.
- Graphiques personnalisables et tableaux de bord

Configuration des alertes et des notifications

## Introduction à Munin

Outil open-source de surveillance des ressources système

Fonctionne avec une architecture maître/nœud

Collecte et affiche des données sous forme de graphiques

Large éventail de plugins, y compris pour MongoDB

Facilité d'extension avec des plugins personnalisés

## Installation et configuration de Munin pour MongoDB

Installation de Munin et Munin-node sur les serveurs concernés

Configuration du fichier munin.conf pour définir les nœuds

Installation du plugin MongoDB pour Munin

- Téléchargement du plugin depuis le référentiel GitHub
- Copie du plugin dans le répertoire des plugins Munin
- Configuration du plugin dans le fichier plugin-conf.d/mongodb

Redémarrage de Munin-node pour appliquer les modifications

Visualisation des graphiques de monitoring MongoDB dans Munin

## Travaux pratiques : Mise en place et utilisation de MongoDB Cloud Manager et Munin

### Configurer et utiliser MongoDB Cloud Manager pour surveiller une base de données MongoDB

- Installer l'agent Cloud Manager
- Configurer les alertes et les notifications

### Installer et configurer Munin pour surveiller MongoDB

- Installer Munin et Munin-node
- Configurer le plugin MongoDB pour Munin
- Analyser les graphiques de monitoring MongoDB dans Munin

# Gestion et fonctionnement des index

## Création d'index

### Utiliser la commande createIndex pour créer un index

- Créer un index simple
  - Exemple d'index simple : `db.collection.createIndex({ champ: 1 })`
  - champ représente le nom du champ à indexer
  - 1 signifie un ordre de tri croissant

### Préciser l'ordre de tri

- Définir l'ordre de tri avec 1 (croissant) ou -1 (décroissant)
- Exemple d'index composé : `db.collection.createIndex({ champ1: 1, champ2: -1 })`
  - L'index sera trié d'abord par `champ1` en ordre croissant, puis par `champ2` en ordre décroissant

### Créer un index en arrière-plan

- Permet d'éviter de bloquer les opérations sur la collection pendant la construction de l'index
- Utiliser l'option `{ background: true }`
- Exemple : `db.collection.createIndex({ champ: 1 }, { background: true })`

## Suppression d'index

### Supprimer un index

- Utiliser la commande `db.collection.dropIndex(...)`
  - Exemple de commande : `db.collection.dropIndex("champ_1")`

### Supprimer tous les index d'une collection

- Utiliser la commande `db.collection.dropIndexes()`
  - Exemple de commande : `db.maCollection.dropIndexes()`

### Exemple

```
// Créer un index sur le champ "age" de la collection "users"
db.users.createIndex({ age: 1 })

// Vérifier que l'index a été créé
db.users.getIndexes()

// Supprimer l'index sur le champ "age"
db.users.dropIndex("age_1")

// Vérifier que l'index a été supprimé
db.users.getIndexes()

// Supprimer tous les index de la collection "users" (sauf l'index par défaut sur le champ "_id")
db.users.dropIndexes()
```

## Analyse des index existants

### Lister les index d'une collection

- Utiliser la commande : `db.maCollection.getIndexes()`
- Permet d'afficher tous les index de la collection et leurs propriétés

### Identifier les index les plus utilisés

- Utiliser la commande : `db.maCollection.aggregate([ { $indexStats: {} } ])`
- Permet d'obtenir des statistiques sur l'utilisation des index pour optimiser les performances

### Exemple d'analyse des index existants

```
// Créer une collection avec des index :
db.maCollection.insertMany([
  { champ1: "valeur1", champ2: "valeur2" },
  { champ1: "valeur3", champ2: "valeur4" }
])
db.maCollection.createIndex({ champ1: 1 })
db.maCollection.createIndex({ champ2: 1 })

//Lister les index de la collection :
db.maCollection.getIndexes()
```

Résultat :

```
[
  {
    "v": 2,
    "key": {
      "_id": 1
    },
    "name": "_id_"
  },
  {
    "v": 2,
    "key": {
      "champ1": 1
    },
    "name": "champ1_1"
  },
  {
    "v": 2,
    "key": {
      "champ2": 1
    },
    "name": "champ2_1"
  }
]
```

Obtenir les statistiques d'utilisation des index :

```
db.maCollection.aggregate([ { $indexStats: {} } ])
```

Résultat :

```
[
  {
    "name" : "_id_",
    "key" : {
      "_id" : 1
    },
    "host" : "localhost:27017",
    "accesses" : {
      "ops" : 4,
      "since" : ISODate("..."),
      "note" : "..."
    }
  },
  ...
]
```

# Gestion des index en arrière-plan

## Concept des index en arrière-plan

- Permet aux autres opérations de continuer pendant la création de l'index

- Avantages : évite de bloquer les opérations de la base de données lors de la création d'index
- Incovénients:
  - Peuvent prendre plus de temps à créer que les index créés en mode normal (foreground)
  - Utilisent plus de ressources système (CPU, mémoire) pendant leur création

### Fonctionnement des index en arrière-plan

- Ajouter `{ background: true }` lors de la création d'index
- Exemple de commande shell pour créer un index en arrière-plan : `db.collection.createIndex({ champ: 1 }, { background: true })`

### Exemple de création d'un index en arrière-plan

```
db.utilisateurs.createIndex({ email: 1 }, { background: true })
```

## Utilisation de l'option "sparse"

### Concept des index "sparse"

- N'incluent que les documents contenant le champ indexé
- Ignorent les documents sans le champ spécifié, réduisant ainsi la taille de l'index
- Peuvent améliorer les performances pour les collections avec des structures variables
- Commande: `db.collection.createIndex({ champ: 1 }, { sparse: true })`

### Exemple

Création d'un index "sparse" sur le champ "age" :

```
db.users.createIndex({ age: 1 }, { sparse: true })
```

Insertion de documents avec et sans le champ "age" :

```
db.users.insertMany([{ name: "Alice", age: 30 }, { name: "Bob" }])
```

Requête utilisant l'index "sparse" :

```
db.users.find({ age: { $gt: 25 } })
```

... retourne uniquement le document avec "Alice" et "age" 30, car l'index "sparse" n'inclut pas le document "Bob" sans le champ "age"

# Index des champs uniques et composés

## Index uniques

### Objectif des index uniques

- Assurer l'intégrité des données en empêchant les doublons pour un champ spécifique
- Les index uniques empêchent l'insertion de documents avec des valeurs de champ en double

### Création d'un index unique

- Utiliser l'option { unique: true } lors de la création de l'index
- Exemple de commande : `db.collection.createIndex({ champ: 1 }, { unique: true })`

### Exemple d'index unique

```
db.collection.createIndex({ champ: 1 }, { unique: true })
```

## Index composés

### Définition des index composés

- Les index composés combinent plusieurs champs pour créer un index
- Ils permettent d'optimiser les requêtes impliquant plusieurs critères de recherche

### Avantages des index composés

- Réduisent le besoin de parcourir l'ensemble des documents pour les requêtes impliquant plusieurs champs
- Accélèrent les requêtes de recherche et de tri sur plusieurs champs

### Création d'un index composé

- Utiliser la commande `db.collection.createIndex({ champ1: 1, champ2: -1 })`

- Exemple: création d'un index composé sur les champs champ1 (ascendant) et champ2 (descendant)

### Exemple

Pour créer un index composé

```
db.users.createIndex({ age: 1, lastName: -1 })
```

Pour effectuer une requête utilisant un index composé

```
db.users.find({ age: { $gte: 25 }, lastName: "Doe" }).sort({ age: 1, lastName: -1 })
```

## Ordre des champs dans les index composés

### Importance de l'ordre des champs

- L'ordre des champs dans les index composés affecte la manière dont MongoDB utilise l'index
- Les champs les plus restrictifs doivent être placés en premier

### Exemple d'ordre des champs

- Les champs les plus restrictifs doivent être placés en premier
  - Les champs avec une grande diversité de valeurs
  - Les champs les moins fréquemment utilisés en tant que critères de recherche
- Exemple : `db.collection.createIndex({ age: 1, nom: 1 })` (l'âge étant un champ plus restrictif que le nom)

### Utilisation des préfixes d'index

- Les préfixes d'index sont des sous-ensembles d'index composés
- Les requêtes utilisant un préfixe d'index peuvent tirer parti de l'index composé
- Exemple : `db.collection.find({ age: 30 }).sort({ nom: 1 }` (utilise l'index composé `{ age: 1, nom: 1 }` )

# Utilisation des index composés pour trier les résultats

## Les index composés peuvent être utilisés pour trier les résultats des requêtes

- Améliore les performances de tri
- Évite les tris en mémoire

## Respecter l'ordre des champs dans l'index pour le tri

- La direction des champs (ascendante/descendante) doit être respectée
- Exemple :
  - Index : `{ champ1: 1, champ2: -1 }`
  - Tri valide : `{ champ1: 1, champ2: -1 }`
  - Tri invalide : `{ champ1: -1, champ2: 1 }`

# Index des tableaux et des sous-documents

## Index des tableaux (multi-clés)

### Objectif des index multi-clés

- Accélèrent les requêtes sur les éléments de tableaux
- Créés automatiquement pour les champs de tableaux lors de l'indexation

### Commande pour créer un index multi-clés

- Utiliser la commande : `db.collection.createIndex({field: 1})`
- Exemple : `db.products.createIndex({tags: 1})`

### Recherche dans les tableaux avec des index multi-clés

- Opérateurs spécifiques : `$all`, `$elemMatch`, `$size`
- Exemples:
  - `$all` : `db.products.find({tags: {$all: ["electronics", "computers"]}})`
  - `$elemMatch` : `db.students.find({grades: {$elemMatch: {score: {$gt: 90}}}})`
  - `$size` : `db.products.find({tags: {$size: 3}})`

### Limitations des index multi-clés

- Ne peuvent pas couvrir plusieurs champs de tableaux dans une seule requête
- Les index multi-clés à composés sont limités à 31 champs maximum

## Index des sous-documents

### Objectif

- Améliorer les requêtes sur les sous-documents avec des index
- Indexer les sous-champs pour optimiser les recherches

### Création d'index sur un sous-champ

- Commande : `db.collection.createIndex({"field.subfield": 1})`

- Exemple : `db.users.createIndex({"address.city": 1})`

### Recherche dans les sous-documents

- Notation pointée pour accéder aux sous-champs
- Exemple

```
// Recherche d'utilisateurs par ville :
db.users.find({"address.city": "New York"})

// Recherche d'utilisateurs par code postal et pays :
db.users.createIndex({"address.zip": 1, "address.country": 1})
db.users.find({"address.zip": "12345", "address.country": "USA"})
```

### Tri des résultats

- Utiliser `.sort()` avec la notation pointée
- Exemple : `db.users.find().sort({"address.city": 1})`

## Recherche et requêtes avancées

### Index multi-clés et sous-documents

- Index multi-clés pour les requêtes sur les tableaux
- Notation pointée pour les requêtes sur les sous-documents

### Exemples de requêtes avancées

- Combinaison de conditions sur un tableau et un sous-document :

```
db.users.find({tags: "developer", "address.city": "Paris"})
```

- Utilisation de $elemMatch pour filtrer sur plusieurs conditions dans un tableau :

```
db.users.find({hobbies: {$elemMatch: {name: "tennis", skillLevel: {$gte: 3}}}})
```

# Index géo-spatiaux

## Introduction aux index géo-spatiaux

### Les index géo-spatiaux facilitent les requêtes sur des données géographiques

Exemple d'une structure de document avec des données géographiques :

```
{
  name: "Tour Eiffel",
  location: {
    type: "Point",
    coordinates: [2.2945, 48.8583]
  }
}
```

### Types d'index géo-spatiaux : 2D et 2DSphere

- Index 2D : pour des données planaires (cartésiennes)
  - Convient pour des données dans un espace cartésien, comme des coordonnées [x, y] ou {x, y}
  - Exemple de données plannaires:

    ```
    {
      name: "Point A",
      location: [50, 100]
    }
    ```

- Index 2DSphere : utilisés pour des données sphériques (géographiques) tenant compte la courbure de la Terre
  - Exemple de données géographiques au format GeoJSON

    ```
    {
      name: "Tour Eiffel",
      location: {
        type: "Point",
        coordinates: [2.2945, 48.8583]
      }
    }
    ```

# Création d'index 2D

## Créer un index 2D

- Exemple : `db.places.createIndex({location: "2d"})`

## Format des données géographiques pour index 2D

- Exemple 1 (format tableau) : `{location: [2, 3]}`
- Exemple 2 (format objet) : `{location: {x: 2, y: 3}}`

## Insérer des documents avec des coordonnées géographiques

- Exemple : `db.places.insert([{name: "A", location: [2, 3]}, {name: "B", location: [4, 6]}, {name: "C", location: [8, 2]}])`

## Requête pour trouver des documents à proximité d'un point donné avec un index 2D

- Requête : `db.places.find({location: {$near: [3, 4], $maxDistance: 5}})`
- Résultat : documents dont la distance de "location" au point `[3, 4]` est inférieure ou égale à 5 unités

# Création d'index 2DSphere

## Créer un index 2DSphere

- Exemple : `db.places.createIndex({location: "2dsphere"})`

## Format des données géographiques pour index 2DSphere (GeoJSON)

- Point : `{type: "Point", coordinates: [<longitude>, <latitude>]}`
- Polygon : `{type: "Polygon", coordinates: [[[<lon1>, <lat1>], [<lon2>, <lat2>], ..., [<lon1>, <lat1>]]]}`

## Exemple d'insertion et de requête avec index 2DSphere

Insertion d'un document avec des données géographiques au format GeoJSON :

```
db.places.insert({
  name: "Tour Eiffel",
  location: {
    type: "Point",
    coordinates: [2.294481, 48.858158]
  }
})
```

Création de l'index 2DSphere sur le champ "location" :

```
db.places.createIndex({location: "2dsphere"})
```

Recherche des lieux à proximité de la Tour Eiffel dans un rayon de 1 km :

```
db.places.find({
  location: {
    $near: {
      $geometry: {
        type: "Point",
        coordinates: [2.294481, 48.858158]
      },
      $maxDistance: 1000
    }
  }
})
```

# Requêtes géo-spatiales avec MongoDB

## Opérateur $near

- Recherche des documents à proximité d'un point spécifique
- Exemple de commande shell :

```
db.places.find({
 location: {
  $near: {
    $geometry: {
      type: "Point",
      coordinates: [-73.9667, 40.78] // longitude, latitude
    },
    $maxDistance: 1000 // distance en mètres
  }
 }
})
```

## Opérateur $geoWithin

- Recherche des documents dans une région géographique délimitée

- Exemple de commande shell :

```
db.places.find({
  location: {
    $geoWithin: {
      $geometry: {
        type: "Polygon",
        coordinates: [
          [
            [-74.0, 40.7], // coordonnées des sommets
            [-73.9, 40.7],
            [-73.9, 40.8],
            [-74.0, 40.8],
            [-74.0, 40.7]
          ]
        ]
      }
    }
  }
})
```

## Opérateur $geoIntersects

- Recherche des documents qui intersectent une région géographique spécifique (ex. ligne, polygone)
- Exemple de commande shell :

```
db.places.find({
  location: {
    $geoIntersects: {
      $geometry: {
        type: "LineString",
        coordinates: [
          [-74.0, 40.7], // coordonnées des points de la ligne
          [-73.9, 40.7]
        ]
      }
    }
  }
})
```

# Collections plafonnées, index TTL et curseurs

## Collections plafonnées

### Collections à taille fixe

- Taille déterminée à la création
- Suppression des documents les plus anciens lorsque la taille maximale est atteinte

### Rotation automatique des documents

- Pas besoin de gérer manuellement la suppression des documents
- Idéal pour les données temporaires et les flux de données en temps réel

### Création avec db.createCollection()

- Commande shell : db.createCollection("logs", {capped: true, size: 1000000, max: 1000})
- Configuration : {capped: true, size: , max: }
- size : taille maximale de la collection en octets
- max : nombre maximal de documents (optionnel)

### Utilisations courantes : logs, cache, flux de données en temps réel

Exemple de code pour insérer un log :

```
db.logs.insertOne({ timestamp: new Date(), message: "Nouveau log", level: "info" });
```

Exemple de code pour récupérer les logs récents :

```
var logs = db.logs.find().sort({ timestamp: -1 }).limit(10);
logs.forEach(log => printjson(log));
```

# Index TTL (Time-To-Live)

## Suppression automatique des documents après un délai spécifié

## Adapté aux données temporaires

- Exemples : sessions utilisateur, logs, données expirables
- Applications : surveillance, analyse des logs, traitement de flux de données

## Création d'un index TTL

- Utiliser la commande db.collection.createIndex()
- Spécifier le champ de date ou Timestamp et le délai d'expiration en secondes

Exemple :

```
db.sessions.createIndex({ "createdAt": 1 }, { expireAfterSeconds: 3600 })
```

## Définir le champ de date ou Timestamp lors de l'insertion du document

Exemple d'insertion de document avec un champ "createdAt" :

```
db.sessions.insert({ "user": "jdoe", "createdAt": new Date() })
```

## Fonctionnement des index TTL

- Le processus MongoDB supprime périodiquement les documents expirés
- Les documents sont supprimés en fonction de leur champ de date ou Timestamp
- Le délai d'expiration peut être modifié en utilisant collMod

Exemple pour modifier le délai d'expiration à 7200 secondes :

```
db.runCommand({ collMod: "sessions", index: { keyPattern: { createdAt: 1 },
expireAfterSeconds: 7200 }})
```

## Limitations des index TTL

- Ne fonctionne pas sur les collections plafonnées

- Ne supprime pas les documents si le champ de date ou Timestamp est manquant ou non valide
- Les suppressions ne sont pas instantanées, mais effectuées lors des vérifications périodiques par MongoDB

# Utilisation des curseurs Tailable

## Objectifs

- Curseurs persistants pour les collections plafonnées
- Suivre les insertions en temps réel
- Similaire à la fonctionnalité tail -f sous Unix

## Création d'une collection plafonnée

- Commande : db.createCollection("logs", { capped: true, size: 1000000, max: 1000 })

## Utilisation avec cursor.addOption(DBQuery.Option.tailable)

- Exemple :

```
var cursor = db.logs.find().sort({ $natural: -1 }).limit(1);
cursor.addOption(DBQuery.Option.tailable);
cursor.addOption(DBQuery.Option.awaitData);
while (cursor.hasNext()) {
  printjson(cursor.next());
}
```

Commande alternative :

```
db.logs.find().addOption(DBQuery.Option.tailable).addOption(DBQuery.Option.awaitData);
```

# Agrégation à finalité unique

## Introduction à l'agrégation à finalité unique

- L'agrégation à finalité unique permet d'effectuer des opérations d'agrégation simples directement sur un champ spécifique d'une collection.
- Les méthodes d'agrégation à finalité unique sont généralement plus rapides et plus simples que les pipelines d'agrégation et Map-reduce pour des opérations simples.

## Utilisation de méthodes d'agrégation simples : count, sum, avg, min, max

- `$count` : Compter le nombre de documents dans une collection ou correspondant à un filtre.
  - Exemple : `db.collection.count()`
- `$sum` : Calculer la somme des valeurs d'un champ spécifique pour tous les documents correspondant à un filtre.
  - Exemple : `db.collection.aggregate([{ $group: { _id: null, total: { $sum: "$champ" }}}])`
- `$avg` : Calculer la moyenne des valeurs d'un champ spécifique pour tous les documents correspondant à un filtre.
  - Exemple : `db.collection.aggregate([{ $group: { _id: null, moyenne: { $avg: "$champ" }}}])`
- `$min` : Trouver la valeur minimale d'un champ spécifique parmi les documents correspondant à un filtre.
  - Exemple : `db.collection.aggregate([{ $group: { _id: null, minimum: { $min: "$champ" }}}])`
- `$max` : Trouver la valeur maximale d'un champ spécifique parmi les documents correspondant à un filtre.
  - Exemple : `db.collection.aggregate([{ $group: { _id: null, maximum: { $max: "$champ" }}}])`

# Agrégation avec des opérateurs de groupe

## $group

- regrouper des documents en fonction d'un ou plusieurs champs

## $sum

- additionner les valeurs d'un champ spécifié
- Exemple: `{ $sum: "$field" }`

## $avg

- calculer la moyenne des valeurs d'un champ spécifié
- Exemple: `{ $avg: "$field" }`

## $min

- déterminer la valeur minimale d'un champ spécifié
- Exemple: `{ $min: "$field" }`

## $max

- déterminer la valeur maximale d'un champ spécifié
- Exemple: `{ $max: "$field" }`

## $push

- ajouter les valeurs d'un champ spécifié à un tableau
- Exemple: `{ $push: "$field" }`

## $addToSet

- ajouter les valeurs d'un champ spécifié à un tableau de valeurs uniques
- Exemple: `{ $addToSet: "$field" }`

# Agrégation conditionnelle

## $cond

- appliquer une condition ternaire (si... alors... sinon...) sur les documents
- Exemple: `{ $cond: { if: { $gte: ["$field", 0] }, then: "$field", else: "N/A" } }`

## $ifNull

- remplacer les valeurs nulles ou inexistantes par une valeur par défaut
- Exemple: `{ $ifNull: ["$field", "N/A"] }`

# Exemples d'agrégation à finalité unique

## Exemple 1: Calculer le montant total des ventes par catégorie

```
db.collection.aggregate([
  {
    $group: {
      _id: "$category", totalSales: { $sum: "$sales" }
    }
  }
])
```

## Exemple 2: Obtenir la moyenne des notes par étudiant

```
db.collection.aggregate([
  {
    $group: {
      _id: "$student",
      avgGrade: { $avg: "$grade" } }
    }
])
```

## Exemple 3: Trouver le prix minimum et maximum par produit

```
db.collection.aggregate([
  {
    $group: {
      _id: "$product",
      minPrice: {
        $min: "$price"
      },
      maxPrice: {
        $max: "$price"
      }
```

```
    }
  }
])
```

# Pipelines d'agrégation

## Introduction aux pipelines d'agrégation

- Les pipelines d'agrégation permettent de traiter et d'analyser des données dans MongoDB.
- Les avantages incluent la flexibilité, l'expressivité et la performance.
- Les pipelines sont composés d'étapes successives qui modifient et filtrent les données.
- Syntaxe de base de l'agrégation pipeline : db.collection.aggregate([])

## Les étapes du pipeline

### $match

- Utilisation de $match pour filtrer les documents en fonction de critères spécifiques
- Équivalent d'une clause WHERE dans SQL
- Commande : {$match: {}}
- Exemple : db.sales.aggregate([{$match: {status: "A"}}])

### $project

- Sélectionner, renommer ou ajouter des champs aux documents.
- Équivalent d'une clause SELECT dans SQL
- Commande : {$project: {: , : , ...}}
- Exemple : db.sales.aggregate([{$project: {_id: 0, item: 1, amount: {$multiply: ["$price", "$quantity"]}}}])

### $unwind

- Transformer un champ de type tableau en plusieurs documents.
- Commande : {$unwind: {path: , includeArrayIndex: , preserveNullAndEmptyArrays: }}
- Exemple : db.books.aggregate([{$unwind: "$authors"}])

## $group

- Regrouper des documents par un ou plusieurs champs.
- Équivalent d'une clause GROUP BY dans SQL
- Commande : {$group: {_id: , : {: }, ...}}
- Exemple : db.sales.aggregate([{$group: {_id: "$status", total: {$sum: "$price"}}}])

## $sort

- Trier les documents en fonction de critères spécifiés.
- Équivalent d'une clause ORDER BY dans SQL
- Commande : {$sort: {: , : , ...}}
- Exemple : db.sales.aggregate([{$sort: {item: 1, price: -1}}])

## $limit

- Limiter le nombre de documents retournés.
- Équivalent d'une clause LIMIT dans SQL
- Commande : {$limit: }
- Exemple : db.sales.aggregate([{$limit: 5}])

## $skip

- Ignorer un certain nombre de documents en début de pipeline.
- Équivalent d'une clause OFFSET dans SQL
- Commande : {$skip: }
- Exemple : db.sales.aggregate([{$skip: 10}])

## $lookup

- Joindre des documents d'une autre collection.
- Exemple: {$lookup: {from: 'orders', localField: '_id', foreignField: 'customer_id', as: 'customer_orders'}}

## $out

- Écrire le résultat du pipeline dans une nouvelle collection.

- Commande : {$out: }
- Exemple : db.sales.aggregate([{$match: {status: "A"}}, {$out: "active_sales"}])

## Exemples de pipelines d'agrégation

Exemple 1 : Calculer le total des ventes par statut

```
db.sales.aggregate([{$group: {_id: "$status", total: {$sum: "$price"}}}])
```

Exemple 2 : Trouver le nombre moyen d'étoiles par catégorie pour les films ayant plus de 100 votes

```
db.movies.aggregate([
  {$match: {numVotes: {$gt: 100}}},
  {$group: {_id: "$genre", avgStars: {$avg: "$stars"}}},
  {$sort: {avgStars: -1}}
])
```

Exemple 3 : Calculer le total des ventes par jour et par produit

```
db.sales.aggregate([
  {$project: {day: {$dayOfYear: "$date"}, item: 1, price: 1}},
  {$group: {_id: {day: "$day", item: "$item"}, total: {$sum: "$price"}}}
])
```

Exemple 4 : Trouver le top 3 des clients ayant effectué le plus d'achats

```
db.orders.aggregate([
  {$group: {_id: "$customerId", totalSpent: {$sum: "$amount"}}},
  {$sort: {totalSpent: -1}},
  {$limit: 3}
])
```

Exemple 5 : Calculer la répartition des ventes par quartier et par catégorie de produit

```
db.sales.aggregate([
  {$unwind: "$items"},
  {$group: {_id: {neighborhood: "$neighborhood", category: "$items.category"}, totalSales: {$sum: "$items.price"}}},
  {$sort: {"_id.neighborhood": 1, totalSales: -1}}
])
```

# Map-Reduce

## Introduction à Map-reduce

### Concept

- Map-reduce est un modèle de programmation pour traiter et générer de grands ensembles de données
- Popularisé par Google, utilisé dans divers systèmes de traitement de données massives, tels qu'Hadoop
- Divise les tâches en deux étapes principales : Map et Reduce
- Les données sont traitées en parallèle pour améliorer les performances

### Avantages

- Capacité à traiter de grandes quantités de données
- Scalabilité et distribution sur plusieurs machines
- Parallélisation des tâches pour améliorer les performances

## Fonctions Map et Reduce

### Fonction Map

- La fonction Map prend en entrée des paires clé-valeur (documents dans MongoDB)
- Applique une fonction de transformation à chaque paire clé-valeur pour produire de nouvelles paires clé-valeur intermédiaires
- Les paires clé-valeur intermédiaires sont regroupées par clé

Exemple de fonction Map dans MongoDB :

```
function() {
    emit(this.categorie, this.valeur);
}
```

### Fonction Reduce

- La fonction Reduce prend en entrée les paires clé-valeur intermédiaires regroupées par clé (produites par la fonction Map)

- Applique une fonction de réduction pour fusionner les valeurs associées à chaque clé
- Génère des paires clé-valeur finales comme résultat
- Exemple de fonction Reduce dans MongoDB :

```
function(cle, valeurs) {
   return Array.sum(valeurs);
}
```

### Utiliser Map-reduce dans MongoDB

- Map-reduce est implémenté dans MongoDB via la méthode `mapReduce()`
- La méthode `mapReduce()` prend en arguments les fonctions Map et Reduce, ainsi que d'autres options (par exemple, le nom de la collection de sortie)
- Exemple d'utilisation de Map-reduce dans MongoDB :

```
db.ma_collection.mapReduce(
   function() { emit(this.categorie, this.valeur); },
   function(cle, valeurs) { return Array.sum(valeurs); },
   {
      out: "resultat_map_reduce"
   }
)
```

## Finalize et options supplémentaires

### Fonction finalize (optionnelle)

- Traite les résultats après la phase de réduction
- Exemple : calculer la moyenne des scores

### Options supplémentaires

- query : filtrer les documents avant le traitement
- sort : trier les documents avant le traitement
- limit : limiter le nombre de documents traités
- out : spécifier la collection de sortie

### Exemples

Calculer la somme des scores par utilisateur :

```
db.scores.mapReduce(map, reduce, { out: "sum_scores" })
```

Filtrer les documents avant le traitement :

```
db.scores.mapReduce(map, reduce, { query: { game: "chess" }, out: "chess_scores" })
```

Trier les documents et limiter le nombre de documents traités :

```
db.scores.mapReduce(map, reduce, { sort: { date: 1 }, limit: 100, out: "latest_scores" })
```

# Comparaison entre Map-reduce et les pipelines d'agrégation

## Map-reduce

- Plus flexible, mais plus complexe
- Convient pour les tâches distribuées et parallèles
- Performances généralement inférieures aux pipelines d'agrégation

## Pipelines d'agrégation

- Plus simples et intuitifs
- Performances généralement meilleures pour la plupart des cas d'utilisation
- Moins adaptés aux tâches distribuées et parallèles

# Exemples de Map-reduce

## Calculer la somme des scores par utilisateur :

```
db.scores.mapReduce(map, reduce, { out: "sum_scores" })
```

## Filtrer les documents avant le traitement :

```
db.scores.mapReduce(map, reduce, { query: { game: "chess" }, out: "chess_scores" })
```

## Trier les documents et limiter le nombre de documents traités :

```
db.scores.mapReduce(map, reduce, { sort: { date: 1 }, limit: 100, out: "latest_scores" })
```

# Réplication asynchrone dans MongoDB

## Principe de réplication asynchrone

- Réplication
  - synchronisation des données entre serveurs pour assurer la cohérence
- Asynchrone
  - réplication se produit sans attendre la confirmation du serveur secondaire
- Replica set
  - groupe de serveurs participant à la réplication
- Membre primaire
  - accepte les écritures, réplique les données vers les secondaires
- Membres secondaires
  - copient les données du primaire, peuvent devenir primaires si besoin

## Avantages et inconvénients de la réplication asynchrone

### Avantages

- Haute disponibilité :
  - Récupération rapide en cas de défaillance du serveur primaire
  - Basculement automatique vers un nouveau primaire en cas de défaillance
- Équilibrage de la charge de lecture
  - Répartition des requêtes de lecture entre les membres
- Tolérance aux pannes
  - Préservation des données même en cas de panne d'un serveur

### Inconvénients

- Latence de réplication
  - données peuvent être légèrement en retard sur les membres secondaires

- Incohérence temporaire
    - état des données peut varier entre les membres du replica set

## Mécanismes internes de la réplication asynchrone

- Oplog
    - journal des opérations effectuées sur le membre primaire, base de la réplication
- Tailing
    - processus par lequel les membres secondaires suivent l'Oplog, et appliquent les modifications
- Initial sync
    - processus de copie initiale des données du primaire vers un nouveau membre secondaire
- Heartbeats
    - messages périodiques entre les membres pour vérifier leur état

## Oplog : structure et fonctionnement

- Collection MongoDB : `"local.oplog.rs"`
- Stocke les opérations sous forme de documents BSON
- Oplog circulaire : taille fixe, écrase les anciennes opérations lorsque la limite est atteinte
- Accès à l'oplog : `db.getReplicationInfo()` (informations sur l'oplog), db.printReplicationInfo() (format lisible)
- Exemple d'opération dans l'oplog : `{ "ts" : Timestamp(1628004626, 1), "t" : NumberLong("1"), "h" : NumberLong("0"), "v" : 2, "op" : "i", "ns" : "test.collection", "ui" : UUID("..."), "wall" : ISODate("..."), "o" : { "_id" : ObjectId("..."), "field" : "value" } }`
- Champs importants :
    - `"ts"` (timestamp),
    - `"op"` (type d'opération),
    - `"ns"` (namespace),
    - `"o"` (opération)

# Mise en place et entretien d'un replica set

## Configuration initiale d'un replica set

- Installer MongoDB sur chaque serveur
  - utiliser la version stable et compatible entre les membres
- Configurer les fichiers de configuration ( `mongod.conf` ) :
  - `replicaSet` : spécifier le nom unique du replica set
  - `bindIp` : définir l'adresse IP du serveur pour écouter les connexions
  - `port` : déterminer le port d'écoute (par défaut : 27017)
- Démarrer le service MongoDB sur chaque serveur
  - `systemctl start mongod` ou `service mongod start`
- Se connecter à l'instance principale avec mongo shell
  - `mongosh --host adresse_IP_du_serveur --port port_d'écoute`
- Initialiser le replica set :

```
rs.initiate({
  _id: "nom_du_replica_set",
  members: [
    { _id: 0, host: "adresse_IP_serveur1:port" },
    { _id: 1, host: "adresse_IP_serveur2:port" },
    { _id: 2, host: "adresse_IP_serveur3:port" }
  ]
})
```

## Ajout et suppression de membres

Ajouter un membre au replica set :

```
rs.add("adresse_IP_serveur4:port")
```

Supprimer un membre du replica set :

```
rs.remove("adresse_IP_serveur2:port")
```

# Gestion des priorités et votes des membres

## Intérêt

- Les priorités déterminent l'ordre de préférence pour l'élection d'un membre en tant que primaire.
- Les votes servent à attribuer un poids aux membres lors de l'élection d'un nouveau primaire.

## Modifier la priorité d'un membre

```
cfg = rs.conf()
cfg.members[1].priority = 2
rs.reconfig(cfg)
```

## Modifier les votes d'un membre

```
cfg = rs.conf()
cfg.members[1].votes = 0
rs.reconfig(cfg)
```

# Surveillance et maintenance d'un replica set

## Vérifier l'état

- Vérifier l'état du replica set : `rs.status( )`
- Vérifier l'état des membres et les rôles actuels avec `rs.isMaster()`
- Consulter les logs : `db.adminCommand({ logRotate: 1 })`

## Surveiller

- Utiliser les outils de surveillance :
  - `mongostat` : statistiques en temps réel
  - `mongotopa` : activité des collections

## Maintenance

- Gérer la maintenance planifiée :
  - Mettre un membre en mode maintenance : `rs.stepDown(temps)` (où temps est la durée en secondes)

- Redémarrer un membre avec une configuration modifiée : `rs.reconfig(cfg, { force: true })` (attention, forcer une reconfiguration peut entraîner une perte de données)

# Utilisation de "write concern" et "read preference"

## Comprendre le "write concern"

- Définition : niveau de garantie pour les écritures
- Impact sur la durabilité et la disponibilité des données

## Niveaux de "write concern"

- **w:** : aucune garantie, succès immédiat, risque accru de perte de données
- **w: 1** : écriture confirmée sur le nœud primaire, compromis entre performance et sécurité
- **w: >1** : écriture confirmée sur un nombre spécifié de membres, permet de garantir la durabilité sur plusieurs nœuds
- **w: "majority"** : écriture confirmée sur la majorité des membres du replica set, garantit la meilleure durabilité des données

## Réglage du "write concern"

- Configuration au niveau du client
  - Exemple : `db.collection.insertOne({ ... }, { writeConcern: { w: <niveau>, j: <journaling>, wtimeout: <timeout> }})`
  - Permet de définir un "write concern" spécifique pour chaque opération
- Configuration au niveau du serveur
  - Exemple : `mongod --setParameter writeConcernMajorityJournalDefault=<valeur>`
  - Applique un "write concern" par défaut pour toutes les opérations sur le serveur

## Comprendre la "read preference"

- Définition : spécifie la préférence pour la lecture des données, permet d'équilibrer les charges de lecture
- Impact sur la répartition des lectures entre les nœuds et la tolérance aux pannes

## Modes de "read preference"

- `primary` : lecture uniquement depuis le nœud primaire (par défaut), garantit des lectures cohérentes
- `primaryPreferred` : lecture depuis le nœud primaire si disponible, sinon depuis un secondaire, tolère les pannes du nœud primaire
- `secondary` : lecture uniquement depuis les nœuds secondaires, réduit la charge sur le nœud primaire
- `secondaryPreferred` : lecture depuis un nœud secondaire si disponible, sinon depuis le primaire, préfère les lectures sur les nœuds secondaires
- `nearest` : lecture depuis le nœud ayant le temps de réponse le plus court, minimise la latence

## Réglage de la "read preference"

- Configuration au niveau du client
  - Exemple : `MongoClient.connect(uri, { readPreference: '<mode>' })` * Définit un mode de "read preference" par défaut pour toutes les opérations de lecture
- Configuration au niveau de la requête
  - Exemple : `db.collection.find({ ... }).readPref('<mode>', [tagSet])`
  - Applique un mode de "read preference" spécifique pour chaque requête
- Utilisation de "tag sets" pour sélectionner des nœuds avec des caractéristiques spécifiques
  - Exemple : `db.collection.find({ ... }).readPref('secondary', [{ region: 'EU' }, { region: 'US' }])`
  - Permet de diriger les lectures vers des nœuds en fonction de critères précis (ex : région géographique)

# Gérer les échecs de réplication

## Identifier les causes d'échec de réplication

### Causes courantes :

- Problèmes de connectivité réseau (interruptions, latence élevée)
- Configuration incorrecte ou incohérente du replica set
- Problèmes matériels (panne de disque, utilisation élevée de CPU, manque de mémoire)
- Opérations de maintenance mal gérées (mises à jour, redémarrages)

### Diagnostic des problèmes de réplication :

- Utiliser `rs.status()` pour vérifier l'état des membres
- Examiner les logs MongoDB pour identifier les erreurs spécifiques
- Utiliser des outils de surveillance pour détecter les problèmes en temps réel (ex : MongoDB Cloud Manager, Prometheus)

## Résolution des problèmes de réplication

- Résoudre les problèmes de connectivité réseau :
  - Vérifier les paramètres du pare-feu, du routeur et des interfaces réseau
  - Tester la connectivité entre les membres du replica set avec ping et traceroute
- Corriger la configuration du replica set :
  - Utiliser `rs.reconfig()` pour ajuster la configuration en fonction des besoins
  - Vérifier les paramètres de priorité, de vote et les adresses des membres
- Résoudre les problèmes matériels :
  - Remplacer ou réparer les composants défectueux rapidement
  - Surveiller et optimiser l'utilisation des ressources (CPU, mémoire, disque) avec des outils comme top, vmstat et iostat Gérer correctement les opérations de maintenance :
  - Effectuer des sauvegardes régulières et vérifier leur intégrité

    - Planifier les mises à jour, les redémarrages et les changements de configuration

## Récupération des données après un échec

- Forcer la synchronisation d'un membre avec un autre :
    - Utiliser `rs.syncFrom()` pour choisir un membre source spécifique
- Démettre le membre primaire défaillant :
    - Utiliser `rs.stepDown()a` pour provoquer une élection du membre primaire
- Restaurer les données à partir d'une sauvegarde récente :
    - Utiliser mongorestore pour récupérer les données depuis la sauvegarde
- Appliquer les opérations en attente à partir de l'oplog :
    - Utiliser mongoreplay pour rejouer les opérations non répliquées
- Vérifier que la réplication est rétablie :
    - Utiliser `rs.status()a` et d'autres outils de diagnostic pour confirmer le bon fonctionnement

## Éviter les échecs de réplication

- Planifier et tester les opérations de maintenance en environnement de test
- Mettre en place une surveillance proactive des ressources et des performances avec des outils adaptés
- Configurer des alertes pour les problèmes potentiels (ex : MongoDB Cloud Manager, outils de monitoring tiers)
- Utiliser des techniques de redondance (ex : RAID

# Sharding automatique

## Introduction au sharding automatique

- Objectif : répartition horizontale des données
- Répartition des données en chunks sur plusieurs serveurs
- Amélioration des performances et de la disponibilité

## Fonctionnement du sharding automatique

- Chaque shard stocke une partie des données
- Utilisation de la shard key pour déterminer la répartition des données
- Chaque chunk contient une plage de valeurs de shard key
- Les données sont réparties automatiquement en fonction de la croissance des données

## Configuration du sharding automatique

- Configurer les serveurs de configuration (config servers)
  - Exemple de commande : mongod --configsvr --dbpath /data/configdb --port 27019
- Configurer les routeurs (mongos)
  - Exemple de commande : mongos --configdb configsvr1:27019,configsvr2:27019,configsvr3:27019 --bind_ip localhost --port 27017
- Ajout de shards au cluster
  - Exemple de commande : sh.addShard("shard1:27018")
- Activer le sharding pour une base de données et une collection
  - Exemple de commande : sh.enableSharding("myDatabase") et sh.shardCollection("myDatabase.myCollection", { "shardKeyField" : 1 })

# Avantages et inconvénients du sharding automatique

## Avantages :

- Augmentation de la capacité de stockage
- Répartition de la charge de travail entre les shards
- Tolérance aux pannes grâce à la réplication des données

## Inconvénients :

- Complexité accrue de la configuration et de l'administration
- Nécessité de choisir une shard key appropriée pour éviter les déséquilibres
- Performances dépendantes de la qualité de la répartition des données et des requêtes

# Mise en place d'un cluster de shards MongoDB

## Architecture d'un cluster de shards

- Composants principaux : shards, serveurs de configuration (config servers) et routeurs (mongos)
- Shards : stockent des sous-ensembles de données partitionnées (chunks)
- Config servers : stockent les métadonnées du cluster et la répartition des chunks
- Mongos : routeur de requêtes entre les applications et les shards

## Configuration des serveurs de configuration (config servers)

- Utiliser MongoDB en mode replica set pour les config servers (3 minimum)
- Lancer les serveurs de configuration avec l'option --configsvr (exemple : mongod --configsvr --replSet --dbpath --port )
- Configurer le replica set des config servers avec rs.initiate() dans le shell mongo
- Utiliser les commandes rs.add() et rs.status() pour gérer les membres du replica set

## Configuration des routeurs (mongos)

- Lancer les instances mongos avec l'option --configdb (exemple : mongos --configdb / --port )
- Plusieurs instances mongos pour la répartition de charge et la tolérance de panne
- Les applications se connectent aux instances mongos, qui gèrent le routage des requêtes vers les shards appropriés

## Ajout de shards au cluster

- Préparer les shards comme replica sets

- Ajouter les shards au cluster avec la commande sh.addShard() dans le shell mongo connecté à une instance mongos (exemple : sh.addShard("/"))
- Utiliser sh.status() pour vérifier l'ajout des shards et les informations sur le cluster
- Activer le sharding pour une base de données avec sh.enableSharding("")
- Choisir et configurer une shard key pour une collection avec sh.shardCollection()

## Monitoring d'un cluster de shards

- Utiliser des outils de supervision intégrés tels que mongotop et mongostat
- Utiliser MongoDB Cloud Manager ou MongoDB Ops Manager pour la surveillance et la gestion du cluster
- Configurer des alertes pour détecter les problèmes de performance, d'équilibrage ou de capacité
- Vérifier régulièrement les logs de MongoDB et les rapports de diagnostic pour identifier les problèmes

# Choisir judicieusement une shard key

## Importance de la shard key

- La shard key détermine la répartition des données dans le cluster
- Une bonne shard key assure un équilibre de charge et une bonne performance
- La shard key influe sur la gestion de l'espace disque et la vitesse de réponse
- Choisir une shard key adéquate est crucial pour le succès du sharding

## Types de shard keys

- Shard key simple : basée sur un seul champ du document
- Shard key composée : basée sur plusieurs champs du document
- Shard key hashed : basée sur un hachage du champ choisi pour une répartition uniforme

## Critères de choix d'une shard key

- Sélectionner un champ fréquemment utilisé dans les requêtes
- Assurer une distribution équilibrée des données
- Privilégier les champs ayant une cardinalité élevée (diversité des valeurs)
- Éviter les champs ayant une forte croissance séquentielle
- Prendre en compte les besoins en lecture et écriture de l'application
- Utilisez sh.shardCollection() pour définir la shard key

## Conséquences d'un mauvais choix de shard key

- Déséquilibre dans la répartition des données (hotspots)
- Performances dégradées en lecture et écriture
- Augmentation des coûts de stockage et de traitement
- Difficultés pour redimensionner le cluster et gérer la montée en charge
- Utilisez db.collection.getShardDistribution() pour vérifier la distribution des données

Exemple de code pour définir la shard key :

```
sh.shardCollection("<database>.<collection>", { <field>: <type> })
```

# Administration avancée d'un cluster de shards

## Gestion des index

- Comprendre l'importance des index dans un cluster de shards
- Créer et supprimer des index dans un cluster
    - Utiliser db.collection.createIndex() pour créer un index
    - Utiliser db.collection.dropIndex() pour supprimer un index
- Index globaux vs index locaux
    - Index globaux : couvrent l'ensemble du cluster
    - Index locaux : spécifiques à un shard
- Conseils pour optimiser les index
    - Éviter les index inutiles
    - Utiliser des index partiels
    - Utiliser des index "sparse" pour les champs optionnels

## Modification de la shard key

- Comprendre les limitations pour modifier la shard key
    - La modification directe de la shard key n'est pas autorisée
- Méthodes pour modifier la shard key
    - Méthode 1 : Créer une nouvelle collection avec la nouvelle shard key
        - Copier les données de l'ancienne collection vers la nouvelle
        - Supprimer l'ancienne collection
    - Méthode 2 : Utiliser la fonction refineCollectionShardKey() (MongoDB 4.4+)
        - Permet d'affiner la shard key existante en ajoutant des champs

## Optimisation des performances du cluster

- Utiliser le profileur de requêtes pour identifier les requêtes lentes
  - Activer le profileur avec db.setProfilingLevel(level, options)
  - Analyser les résultats avec db.system.profile.find()
- Utiliser mongotop et mongostat pour surveiller les performances en temps réel
- Optimiser les requêtes avec les "Explainable objects"
  - Utiliser db.collection.explain() pour analyser les requêtes
- Configurer les routeurs (mongos) pour répartir les requêtes
  - Utiliser les options readPreference, readPreferenceTags, maxStalenessSeconds
- Exploiter la compression de données (WiredTiger)
  - Configurer la compression avec storage.wiredTiger.collectionConfig.blockCompressor

## Gestion des erreurs et dépannage

- Surveiller les logs MongoDB pour identifier les erreurs
  - Logs accessibles dans le dossier /var/log/mongodb sur Linux
- Gérer les problèmes de connexion aux serveurs de configuration (config servers)
  - Vérifier la connectivité réseau
  - Vérifier l'état des serveurs de configuration avec rs.status()
- Résoudre les problèmes liés aux routeurs (mongos)
  - Vérifier la configuration du routeur
  - Redémarrer le routeur si nécessaire
- Diagnostiquer les problèmes de performances
  - Utiliser les outils de monitoring mentionnés précédemment
  - Identifier les goulets d'étranglement et ajuster la configuration en conséquence

# Gérer un cluster de shards déséquilibré

## Identification d'un cluster déséquilibré

- Utiliser la commande db.printShardingStatus() pour vérifier le statut du sharding
- Vérifier la distribution des chunks entre les shards
  - Un déséquilibre est visible si certains shards ont beaucoup plus de chunks que d'autres
- Utiliser MongoDB Atlas ou MongoDB Cloud Manager pour une visualisation graphique

## Analyse des causes d'un déséquilibre

- Shard key inappropriée
  - Clés trop concentrées ou trop dispersées
  - Clés non monotones
- Taille des chunks trop grande ou trop petite
  - Ajuster avec sh.splitFind() et sh.mergeChunks()
- Croissance inégale des données
  - Ajout ou suppression massive de données

## Utilisation de la fonction "Balancer"

- Balancer permet de redistribuer les chunks entre les shards
- Activer le Balancer automatique :
  - sh.startBalancer() pour démarrer
  - sh.stopBalancer() pour arrêter
- Vérifier le statut du Balancer :
  - sh.getBalancerState()
  - sh.isBalancerRunning()
- Configurer des fenêtres de temps pour l'exécution du Balancer

```
use config
db.settings.updateOne({_id: "balancer"}, {$set: {activeWindow: {start: "00:00", stop: "04:00"}}}, {upsert: true})
```

## Conseils pour prévenir le déséquilibre

- Choisir une shard key adaptée
  - Prendre en compte la distribution des données et la croissance prévue
- Surveiller régulièrement la distribution des chunks
  - Utiliser les outils de monitoring comme MongoDB Atlas
- Configurer les seuils de scission et de migration de chunks

```
use config
db.settings.updateOne({_id: "chunksize"}, {$set: {value: <taille_du_chunk>}}, {upsert: true})
```

- Utiliser le Balancer de manière proactive pour rééquilibrer les shards
  - Planifier des fenêtres d'exécution régulières

# Gérer les chunks (scission, fusion, migration)

## Comprendre les chunks

- Chunks : unités de répartition des données dans un cluster de shards
- Chaque chunk contient un intervalle continu de valeurs de shard key
- Chaque chunk appartient à un shard unique
- Taille de chunk par défaut : 64 Mo

## Processus de scission de chunks

- Scission : division d'un chunk en deux nouveaux chunks plus petits
- Déclenchée lorsque la taille d'un chunk dépasse la taille maximale configurée
- MongoDB 5.0+ surveille la taille des chunks automatiquement
- Commande pour forcer une scission manuelle : sh.splitAt(collection, splitPoint)

## Processus de fusion de chunks

- Fusion : combinaison de deux chunks adjacents en un seul chunk
- Peut être utile pour réduire la fragmentation des données
- Commande pour forcer une fusion manuelle : sh.mergeChunks(collection, lowerBound, upperBound)

## Migration de chunks

- Migration : transfert d'un chunk d'un shard vers un autre shard
- Permet d'équilibrer la répartition des données et de la charge entre les shards
- Géré automatiquement par le "Balancer" dans MongoDB 5.0+
- Commande pour démarrer une migration manuelle : sh.moveChunk(collection, find, to)

## Optimisation de la gestion des chunks

- Surveiller la taille et la répartition des chunks avec sh.status()
- Choisir une shard key appropriée pour minimiser le nombre de migrations et de scissions
- Configurer la taille maximale des chunks si nécessaire avec sh.updateZoneKeyRange(collection, min, max, options)
- Planifier les maintenances de scission et fusion de chunks en dehors des heures de pointe

# Authentification et autorisation dans les replica sets et les clusters de shards

## Introduction à l'authentification et l'autorisation

- Authentification : vérification de l'identité d'un utilisateur
- Autorisation : détermination des permissions accordées à un utilisateur
- Importance pour assurer la sécurité et la confidentialité des données
- Dans MongoDB, ces concepts s'appliquent aux replica sets et aux clusters de shards

## Authentification et autorisation pour les replica sets

- Les replica sets nécessitent une configuration sécurisée pour éviter les accès non autorisés
- Création d'un utilisateur administrateur avec la commande db.createUser()
- Exemple : db.createUser({user:"admin", pwd:"password", roles:["root"]})
- Activation de l'authentification en modifiant le fichier de configuration mongod.conf
- Ajout de l'option security.authorization: enabled
- Redémarrage de mongod avec la nouvelle configuration

## Authentification et autorisation pour les clusters de shards

- Les clusters de shards requièrent une configuration similaire à celle des replica sets
- Utilisation d'un utilisateur administrateur pour gérer l'ensemble du cluster
- Activation de l'authentification sur chaque shard et le serveur de configuration
- Exemple de commande pour ajouter un shard avec authentification : sh.addShard("shard1.example.com:27017", {user:"admin", pwd:"password"})

## Utilisation de x.509 pour l'authentification

- X.509 : standard de certificats numériques pour l'authentification
- MongoDB Enterprise prend en charge l'authentification x.509
- Nécessite l'obtention d'un certificat pour chaque membre du replica set ou du cluster
- Modification du fichier de configuration mongod.conf pour ajouter les options x.509
- Exemple : net: { ssl: { mode: "requireSSL", PEMKeyFile: "/path/to/ssl/mongodb.pem", CAFile: "/path/to/ssl/ca.pem" } }

## Utilisation de SCRAM-SHA pour l'authentification

- SCRAM-SHA : mécanisme d'authentification basé sur un échange de défis-réponses
- Mécanisme par défaut dans MongoDB
- Création d'un utilisateur avec SCRAM-SHA en utilisant la commande db.createUser()
- Exemple : db.createUser({user:"example", pwd:"password", roles: ["readWrite"]})
- Utilisation de la commande mongo pour se connecter avec SCRAM-SHA
- Exemple : mongo --username example --password password --authenticationDatabase admin

# Gestion des privilèges et des rôles personnalisés

## Présentation des rôles et privilèges

- Rôles: ensemble de privilèges attribués à un utilisateur
- Privilèges: actions autorisées sur des ressources spécifiques
- Objectif: contrôler l'accès et les actions des utilisateurs

## Rôles et privilèges intégrés

- read: permet de lire les données d'une base de données spécifique
- readWrite: autorise la lecture et l'écriture de données dans une base de données spécifique
- dbAdmin: permet la gestion administrative d'une base de données spécifique
- userAdmin: permet la gestion des utilisateurs d'une base de données spécifique
- clusterAdmin: autorise la gestion administrative d'un cluster
- userAdminAnyDatabase: permet la gestion des utilisateurs pour toutes les bases de données
- dbAdminAnyDatabase: autorise la gestion administrative pour toutes les bases de données
- readWriteAnyDatabase: permet la lecture et l'écriture de données pour toutes les bases de données
- root: rôle avec tous les privilèges sur toutes les bases de données

## Création et gestion de rôles personnalisés

- Utiliser db.createRole() pour créer un nouveau rôle personnalisé
- Exemple: db.createRole({ role: "customRole", privileges: [], roles: [] })
- Utiliser db.updateRole() pour mettre à jour un rôle personnalisé
- Utiliser db.dropRole() pour supprimer un rôle personnalisé

## Attribution de privilèges aux rôles personnalisés

- Ajouter des privilèges à un rôle personnalisé avec db.updateRole()
- Exemple: db.updateRole("customRole", { privileges: [{ resource: { db: "exampleDB", collection: "exampleCollection" }, actions: ["find", "update"] }], roles: [] })
- Attribuer un rôle personnalisé à un utilisateur avec db.updateUser()
- Exemple: db.updateUser("exampleUser", { roles: ["customRole"] })

## Gestion des utilisateurs associés aux rôles

- Créer un utilisateur avec un rôle intégré ou personnalisé avec db.createUser()
  - Exemple: db.createUser({ user: "exampleUser", pwd: "examplePassword", roles: ["customRole"] })
- Modifier les rôles d'un utilisateur avec db.updateUser()
  - Exemple: db.updateUser("exampleUser", { roles: ["readWrite"] })
- Supprimer un utilisateur avec db.dropUser()
  - Exemple: db.dropUser("exampleUser")

# Recommandations pour un déploiement sûr

## 3.1 Configuration de la sécurité réseau

- Activer l'authentification et l'autorisation pour sécuriser l'accès aux données
- Configurer les règles de pare-feu pour limiter l'accès aux ports MongoDB
- Utiliser des connexions TLS/SSL pour chiffrer les communications entre les clients et le serveur
    - Exemple de commande pour activer TLS/SSL : mongod --tlsMode requireTLS --tlsCertificateKeyFile /etc/ssl/mongodb.pem
- Configurer les "bind_ip" pour restreindre l'accès aux interfaces réseau spécifiques
    - Exemple de configuration dans mongod.conf : net: { bindIp: "192.168.1.2" }

## 3.2 Chiffrement des données

- Chiffrement au repos (Encryption at Rest) avec le moteur de stockage WiredTiger
    - Activer le chiffrement au repos dans mongod.conf : storage: { wiredTiger: { engineConfig: { encrypt: { keyFile: "/path/to/keyfile" } } } }
- Chiffrement des sauvegardes avec mongodump et mongorestore
    - Exemple de commande pour chiffrer une sauvegarde : mongodump --archive=/path/to/encrypted_backup --gzip --encryptionKeyFile=/path/to/keyfile
- Chiffrement des données en transit avec TLS/SSL (mentionné précédemment)

## 3.3 Auditer et surveiller l'accès

- Activer l'audit pour enregistrer les actions des utilisateurs dans le système * Exemple de configuration dans mongod.conf : auditLog: { destination: "file", format: "BSON", path: "/path/to/auditlog" }
- Utiliser des outils de surveillance tels que MongoDB Cloud Manager, Ops Manager ou MongoDB Atlas

- Analyser les journaux d'audit pour détecter les accès non autorisés ou les actions suspectes

## 3.4 Gestion des mises à jour et des correctifs

- Surveiller les annonces de sécurité et les notes de version de MongoDB
- Planifier et appliquer les mises à jour de sécurité en suivant les meilleures pratiques
- Tester les mises à jour dans un environnement de préproduction avant de les déployer en production

## 3.5 Protection contre les injections et les attaques

- Valider et nettoyer les entrées utilisateur pour prévenir les injections de code
- Utiliser des bibliothèques et des pilotes MongoDB officiels pour interagir avec la base de données
- Limiter les privilèges des utilisateurs et les accès au minimum nécessaire
- Surveiller les tentatives d'intrusion et mettre en place des mécanismes de blocage des adresses IP malveillantes

# Stratégies basées sur le système de fichiers

## Introduction aux stratégies de sauvegarde basées sur le système de fichiers

- Principe : copie des fichiers de données MongoDB directement du système de fichiers
- Deux approches principales : copie à chaud et copie à froid
- Importance de la cohérence des données lors de la sauvegarde

## Sauvegarde des fichiers de données MongoDB

- Localisation des fichiers de données : répertoire dbPath spécifié dans le fichier de configuration mongod.conf
- Fichiers de données : extension .wt pour le moteur de stockage WiredTiger
- Sauvegarde de l'ensemble des fichiers du répertoire dbPath

## Copie à chaud et à froid

- Copie à chaud :
  - Sauvegarde des fichiers de données pendant que le serveur MongoDB est en cours d'exécution
  - Nécessite des précautions pour assurer la cohérence des données
  - Moins d'impact sur la disponibilité du système
- Copie à froid :
  - Arrêt du serveur MongoDB avant la sauvegarde
  - Garantit la cohérence des données
  - Impact sur la disponibilité du système

## Utilisation de LVM (Logical Volume Manager) pour les snapshots

- LVM : outil de gestion des volumes logiques sous Linux

- Création de snapshots pour garantir la cohérence des données
- Commandes LVM :
  - lvcreate pour créer un snapshot
  - lvremove pour supprimer un snapshot
  - lvs pour lister les snapshots
- Exemple : lvcreate -L 10G -s -n my_mongodb_snapshot /dev/myvg/mongodblv

## Avantages et inconvénients des stratégies basées sur le système de fichiers

- Avantages :
  - Rapidité des sauvegardes et des restaurations
  - Peu d'impact sur les performances du serveur MongoDB
  - Flexibilité avec LVM pour les snapshots
- Inconvénients :
  - Complexité accrue pour assurer la cohérence des données
  - Peut nécessiter un arrêt du serveur MongoDB (copie à froid)
  - Pas de sauvegarde granulaire (collections spécifiques)

## Cas d'utilisation et recommandations

- Utiliser la copie à chaud avec LVM pour les environnements de production
- Privilégier la copie à froid pour les environnements de test et de développement
- Combiner avec d'autres stratégies (mongodump, réplication) pour une solution de sauvegarde complète et robuste

# Utilisation de mongodump et mongorestore

## Présentation de mongodump et mongorestore

- Outils en ligne de commande pour la sauvegarde et la restauration des données MongoDB
- mongodump : extrait les données et les métadonnées
- mongorestore : importe les données et les métadonnées

## Sauvegarde des données avec mongodump

- Commande de base : mongodump
- Options et paramètres :
  - --host : spécifie l'hôte MongoDB
  - --port : spécifie le port MongoDB
  - --db : sauvegarde une base de données spécifique
  - --collection : sauvegarde une collection spécifique
  - --out : spécifie le répertoire de sortie
- Sauvegarde de collections spécifiques :
  - Exemple : mongodump --db myDatabase --collection myCollection --out /backup

## Restauration des données avec mongorestore

- Commande de base : mongorestore
- Options et paramètres :
  - --host : spécifie l'hôte MongoDB
  - --port : spécifie le port MongoDB
  - --db : restaure une base de données spécifique
  - --collection : restaure une collection spécifique
  - --drop : supprime la collection avant la restauration
  - --dir : spécifie le répertoire d'entrée

- Restauration de collections spécifiques :
  - Exemple : mongorestore --db myDatabase --collection myCollection --dir /backup/myDatabase

## Bonnes pratiques et précautions lors de l'utilisation de mongodump et mongorestore

- Effectuer des sauvegardes régulières
- Stocker les sauvegardes dans un emplacement sûr et séparé
- Vérifier l'intégrité des sauvegardes
- Effectuer des tests de restauration réguliers
- Utiliser l'option --oplog pour inclure les opérations de journalisation
- Utiliser l'authentification pour sécuriser l'accès

## Comparaison avec les stratégies basées sur le système de fichiers

- Avantages de mongodump et mongorestore :
  - Portabilité des sauvegardes
  - Flexibilité pour restaurer des collections spécifiques
- Avantages des stratégies basées sur le système de fichiers :
  - Sauvegardes plus rapides
  - Restauration plus rapide en cas de panne complète
- Choisir la stratégie appropriée en fonction des besoins et des contraintes

# Récupération de type point-in-time

## Introduction à la récupération de type point-in-time (PITR)

- Objectif : restaurer les données à un instant précis
- Permet de minimiser les pertes de données en cas de corruption ou de suppression accidentelle
- Nécessite les journaux d'opérations (oplog) pour la mise en œuvre

## Utilisation des journaux d'opérations (oplog) pour la récupération PITR

- Oplog : journal des opérations de modification des données (insertions, mises à jour, suppressions)
- Stocké dans la collection "oplog.rs" du "local" database
- Commande pour vérifier l'oplog : db.getReplicationInfo()
- Taille et durée de rétention de l'oplog configurables
- Extraction des opérations depuis l'oplog pour une période donnée

## Processus de restauration avec PITR

* Sauvegarder l'état actuel des données avec mongodump ou une méthode basée sur le système de fichiers * Extraire les opérations de l'oplog pour la période cible avec mongoexport * Exemple : mongoexport --db local --collection oplog.rs --query "{'ts': {'$gte': startTimestamp, '$lte': endTimestamp}}" --out oplog-export.json * Restaurer les données sauvegardées à l'étape 1 avec mongorestore ou une méthode basée sur le système de fichiers * Appliquer les opérations extraites de l'oplog à l'aide de mongoimport * Exemple : mongoimport --db targetDb --collection targetCollection --file oplog-export.json --mode upsert

## Limitations et considérations lors de l'utilisation de la récupération PITR

* Nécessite un replica set pour utiliser l'oplog * La durée de rétention de l'oplog doit être suffisante pour couvrir la période de récupération souhaitée * La

capacité de stockage doit être prise en compte lors de la configuration de la taille de l'oplog * Ne fonctionne pas avec les transactions multi-documents (à partir de MongoDB 4.0)

## Cas d'utilisation et recommandations

* Récupération après une suppression accidentelle de données * Récupération après une corruption de données due à une erreur de manipulation * Recommandation : utiliser la récupération PITR en complément d'autres stratégies de sauvegarde et de restauration * Tester régulièrement le processus de récupération PITR pour s'assurer de son bon fonctionnement

# Capture et rejeu

## Capture et rejeu avec mongoreplay

- Objectif : enregistrer et rejouer les opérations sur MongoDB pour des tests de performance, de charge ou de débogage
- Fonctionne en interceptant le trafic entre le client et le serveur MongoDB
- ⚠️ L'outil n'est plus supporté / maintenu depuis la v5.

## Installation et prérequis

- Disponible dans MongoDB Database Tools
- Installer MongoDB Database Tools pour utiliser mongoreplay

## Capture des opérations avec mongoreplay

- Utiliser l'option --record pour capturer les opérations
  - Exemple : mongoreplay record --port 27017 --outputFile traffic_capture.bson
- Filtre les opérations enregistrées avec l'option --filter
  - Exemple : mongoreplay record --port 27017 --outputFile traffic_capture.bson --filter '{"ns": "myDb.myCollection"}'

## Rejeu des opérations avec mongoreplay

- Utiliser l'option --playback pour rejouer les opérations capturées
  - Exemple : mongoreplay playback --inputFile traffic_capture.bson --host myMongoHost:27017
- Modifie le comportement du rejeu avec des options supplémentaires
  - Exemple : mongoreplay playback --inputFile traffic_capture.bson --host myMongoHost:27017 --speed 2 --repeat 3

## Limitations et considérations lors de l'utilisation de mongoreplay

- Nécessite un accès réseau pour capturer le trafic entre le client et le serveur
- Peut introduire une latence lors de la capture en temps réel
- Ne prend pas en charge certaines fonctionnalités, telles que les opérations de changement de flux

## Cas d'utilisation et recommandations

- Test de charge et de performance pour identifier les goulets d'étranglement et les optimisations possibles
- Débogage des problèmes de production en rejouant les opérations dans un environnement de test
- Comparaison de l'impact des modifications de l'infrastructure ou du code sur les performances
- Recommandation : utiliser mongoreplay en complément d'autres outils de surveillance et de diagnostic de MongoDB

## Références


- https://www.mongodb.com/docs/manual/reference/program/mongoreplay/
- https://github.com/mongodb-labs/mongoreplay

# Astuces

- https://www.percona.com/blog/2016/12/05/mongodb-troubleshooting-top-5/