MongoDB

m2iformation.fr

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

MongoDB

Introduction à MongoDB

MongoDB est une **base de données NoSQL** populaire, **orientée document**, conçue pour stocker, interroger et gérer des données de manière **flexible** et **scalable**. Mais également pour répondre à des besoins modernes où les structures rigides du SQL deviennent un goulot d'étranglement.

Licence: MongoDB est distribué sous licence Server Side Public License (SSPL).
Plateformes: Disponible pour les systèmes d'exploitations majeurs (Windows, macOS, Linux) et en cloud via MongoDB Atlas.

La structure de MongoDB

Une base de données MongoDB, contrairement à une base de données relationnelle, a une structure plus libre car elle ne suit pas un schéma de construction figé. Malgré tout, les éléments sont regroupés dans des ensembles bien définis tels que :

- Les bases des données (database)
- Les collections
- Les documents

Database

Sur un serveur MongoDB, on peut créer plusieurs bases de données. Chaque database est généralement associée à une application ou à un domaine fonctionnel précis.

Dans MongoDB, les bases, les collections et les documents sont créés à la volée. Par exemple, une collection n'a pas besoin d'être déclarée à l'avance : elle est automatiquement créée lors du premier insert.

Collections

- Les **documents** MongoDB sont regroupés dans des **collections**.
- Une **collection** est **l'équivalent d'une table** dans une base relationnelle.
- Contrairement aux tables SQL, **aucun schéma n’est imposé** à une collection. Cela permet une grande souplesse dans la structure des données.

Exemple: deux documents dans la même collection peuvent avoir des champs complètement différents.

Documents

Les données sont stockées sous forme de **documents BSON** (Binary JSON), un format binaire basé sur **JSON** et qui peut contenir des paires clé-valeur, mais avec plus de types de données (dates, entiers, etc.).

- Chaque document représente un enregistrement (similaire à une ligne dans une base SQL).
- Les documents sont **auto-descriptifs**, c'est-à-dire qu'ils contiennent leur propre structure.
- Les documents d'une même collection peuvent avoir **des structures différentes**.

Chaque document a un **identifiant (_id) unique** au sein de la collection. Il peut être généré automatiquement ou spécifié manuellement.

Langage de requête (MQL)

MongoDB utilise son propre langage appelé **MongoDB Query Language** (MQL). Il repose sur une **syntaxe proche de JSON**, ce qui le rend simple à utiliser et intuitif pour les développeurs.

- MQL permet d'effectuer toutes les opérations CRUD :
  - **Create** : insérer des documents (insertOne, insertMany)
  - **Read** : lire des documents avec des filtres (find, findOne)
  - **Update** : modifier les documents (updateOne, updateMany)
  - **Delete** : supprimer des documents (deleteOne, deleteMany)

Indexation

MongoDB prend en charge l'indexation de données, ce qui permet **d'accélérer les opérations de recherches** en créant des index sur des **champs fréquemment utilisés**.

- Il est possible de créer :
  - Des index **simples** (sur un seul champ),
  - Des index **composés** (sur plusieurs champs),
  - Des index **textuels** ou **géospatiaux** selon les besoins.

Les index sont **cruciaux** pour les **performances à grande échelle**.

Évolutivité

MongoDB est conçu pour être **scalable horizontalement** : ce qui signifie qu'il est facile d'ajouter de nouveaux serveurs pour répartir la charge et gérer plus de données.

- Il utilise un mécanisme appelé **sharding** pour répartir les données sur plusieurs machines.
- Cela permet à MongoDB de **gérer de très grands volumes de données** et un grand nombre d’utilisateurs **sans perte de performance**.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Le BSON (Binary JSON)

Qu'est-ce que le BSON ?

JSON est un format de sérialisation de données en texte, lisible par l'humain. MongoDB utilise un format dérivé appelé **BSON** (Binary JSON), une version **binaire et étendue** de JSON.

- Les développeurs **interagissent avec MongoDB avec du JSON**.
- La conversion entre JSON et BSON est **automatiquement gérée par MongoDB**, ce qui rend la syntaxe simple côté client.
- En raison de sa représentation binaire, BSON est **plus compact** et offre **une meilleure efficacité** en terme de **stockage** et de **traitement**.

Les types de BSON

Le format BSON introduit plusieurs types de données supplémentaires que le format JSON ne prend pas en charge nativement tel que :

- `ObjectId`: Identifiant unique de 12 octets généré automatiquement pour chaque document. Il contient un timestamp, ce qui le rend partiellement ordonnable dans le temps.
- `Date`: Date avec heure, au format ISO, avec précision milliseconde. Représenté par `ISODate()` dans MongoDB.

Les différents types numériques

Dans le cadre du stockage de valeurs numériques, il est possible de fonctionner via plusieurs types de valeurs:

- `Int32`: Les entiers classiques encodés sur **32 bits** via `NumberInt()`
- `Int64`: Les entiers longs encodés sur **64 bits** via `NumberLong()`
- `Double`: Les nombres à virgule flottante encodés sur **64 bits**, utilisables par défaut dans le Shell de par son fonctionnement basé sur le Javascript
- `Decimal128` (ou High Precision Double): Les nombres à virgule flottante encodés sur **128 bits** via `NumberDecimal()`

Exemples de BSON - version simple

```
// Requête JSON (simple):
{
  "nom": "Alice",
  "age": 30,
  "actif": true,
  "solde": 1250.75,
}

// Donnée BSON (vue interne MongoDB):
{
  "_id": ObjectId("6634d055e1a07ab1ac54d6c3"), // id généré automatiquement
  "nom": "Alice",    // String
  "age": 30,         // Int32
  "actif": true,     // Boolean
  "solde": 1250.75   // Double
}
```

Exemples de BSON - avec types BSON

```
// Requête JSON (avec typage explicite via shell ou driver)
{
  "nom": "Alice",
  "age": NumberLong(30),
  "solde": NumberDecimal("1250.75"),
  "dateInscription": ISODate("2024-05-01T10:00:00Z")
}

// Donnée BSON :
{
  "_id": ObjectId("6634d055e1a07ab1ac54d6c3"), // id généré automatiquement
  "nom": "Alice",    // String
  "age": 30,         // Int64
  "solde": 1250.75, // Decimal128
  "dateInscription": "2024-05-01T10:00:00Z" // Date BSON
}
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

MongoDB Compass

Qu'est-ce que MongoDB Compass ?

MongoDB Compass est **l'interface graphique officielle** (GUI) de MongoDB et elle permet de :

- Se connecter facilement à une base MongoDB (locale ou distante)
- Avoir une **vue visuelle** des bases, collections et documents
- **Explorer**, **interroger** et **modifier** les données sans écrire de code
- Afficher les documents BSON de manière **lisible**, avec **détection automatique des types**

Les bases du CRUD - via MongoDB Compass:

- **Insertion (Create):** Sélectionnez la base de données et la collection. Cliquez sur le bouton "Add Data" -> "Insert Document", puis entrez les valeurs du nouveau document.
- **Lecture (Read):** Sélectionner la collection dans Compass. Utilisez l'interface graphique pour construire des requêtes de recherche avec des filtres.
- **Mise à jour (Update):** Au sein de votre collection, utilisez le bouton "Edit" à droite de chaque document.
- **Suppression (Delete):** Utilisez le bouton "Delete" pour supprimer un ou plusieurs documents.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Database & Collection (MongoDB shell)

Création de bases de données

Pour créer une nouvelle base de données dans MongoDB, utilisez simplement la commande `use` suivie du nom de la base.
Si la base de données n'existe pas encore, elle ne sera **effectivement créée que lorsque vous y insérerez un document**.

```
use maNouvelleDB
```

Vous pouvez ensuite vérifier les bases existantes avec `show dbs`:

```
show dbs
```

Remarque : maNouvelleDB n'apparaîtra dans show dbs que si elle contient **au moins** une collection avec des documents.

Méthodes de création de collections

MongoDB crée automatiquement une collection **lors de la première insertion** d'un document, même si elle n'existait pas auparavant :

```
db.livres.insertOne({title: "1001 blagues de dev", author: "Anonyme"});
```

Il est également possible de créer une collection de manière **explicite** à l'aide de la méthode `createCollection()`, notamment si vous souhaitez définir des **options spécifiques** (validation, index, taille maximale,...) :

```
db.createCollection("livres");
```

Créé une collection vide appelée `livres` sans options.

Paramètres d'une collection

Lors de la création d'une collection avec `createCollection()`, vous pouvez spécifier différents **paramètres** :

- **capped:**
  - Si `true`, crée une collection **circulaire** à taille fixe.
  - Les documents les plus anciens sont supprimés lorsque la limite est atteinte.
- **size*:** Taille maximale de la collection, en **octets**.
- **max*:** Nombre maximum de documents dans une collection `capped`.

**Disponible si capped est `true` : `size` est obligatoire, `max` est optionnelle.*

- **validator (document):** Définit des **règles de validation** basées sur le format **JSON Schema**.
- **validationLevel:**
  - "strict" (par défaut): Valide **toutes les opérations** (insertions + mises à jour).
  - "moderate": Valide uniquement les **insertions**.
- **validationAction:**
  - "error" (par défaut): Rejette les documents non conformes.
  - "warn" : Affiche un avertissement, mais **accepte** quand même le document.

Collection avec validator

Le validator permet de définir des règles de validations pour les documents insérés dans une collection. Ces règles sont spécifiées à l'aide de schémas JSON, qui définissent la structure et le contraintes des documents.

`$jsonSchema` : Le champ utilisé pour définir le schéma JSON qui contient les règles de validation dont :

- `bsonType` : Spécifie le type de données du document
- `required` : C'est **un tableau** de noms de champs qui doivent être présents dans chaque document.

- `additionalProperties`: C'est un booléen qui restreint les champs autorisés à ceux définis dans properties.
- `properties` : C'est un **objet** qui définit les contraintes pour chaque champ du document.
  - `bsonType` : Le type de donnée du champ
  - `description` : Une description du champ (facultatif)
  - `minimum` : La valeur min pour les types numériques (facultatif)
  - `maximum` : La valeur max pour les types numériques(facultatif)
  - `pattern` : Une regex pour les chaînes de caractères (facultatif)
  - `enum` : Un tableau de valeurs possibles pour le champ (facultatif)

Exemple de validator

```
db.createCollection("collectionName", {
    validator: {
        $jsonSchema: {
            bsonType: "object",
            required: ["field1", "field2", "field3"],
            properties: {
                field1: {
                    bsonType: "string",
                    description: "Mon premier champ",
                    minimum: 0
                    // Autres contraintes
                },
                field2: {
                    bsonType: "int"
                    // Autres contraintes
                }
            }
        }
    }
})
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Les bases du CRUD (MongoDB shell)

CREATE (insertion de donnée)

Create (insertion unique)

Pour insérer des données dans une base MongoDB, on utilise la méthode `insertOne(data, options)`. Elle prend en paramètre un **objet au format JSON**. Les noms de propriétés peuvent être écrits sans guillemets s'ils sont valides en JavaScript.

```
db.collectionName.insertOne({
  propA: 'Value',
  propB: 12345,
  propC: { propA: true }
})
```

Elle renvoie un objet InsertOneResult contenant des informations sur le résultat de l'information.

Create (insertion multiple)

Il est aussi possible **d'insérer plusieurs documents** d'un coup avec la méthode `insertMany(data, options)`, qui prend un tableau d'objets :

```
db.collectionName.insertMany([
  { prop: 'Value' },
  { prop: 'Value' },
  { prop: 'Value' },
])
```

Suite à l'insertion d'un élément, celui-ci se verra **automatiquement affecté une clé primaire** ayant pour type `ObjectId`. Il est toutefois **possible de la définir manuellement** via l'ajout de la propriété `_id` lors de l'insertion (attention aux duplicats).

Create (option ordered)

Il est possible de passer des options lors d'une insertion multiple via un objet en second argument.

L'option `ordered`, qui est à `true` par défaut, détermine si MongoDB doit arrêter ou continuer l'insertion en cas d'erreur.

```
db.users.insertMany([
  { _id: 1, name: "Max" },
  { _id: 2, name: "Bob" },
  { _id: 1, name: "Sarah" }, // _id en doublon mais continue
  { _id: 3, name: "Claire" },
], { ordered: false })
```

READ (lecture de donnée)

Read

Si l'on veut accéder aux éléments, il est possible de nous servir de la fonction `find(filter, options)` pour afficher tous les éléments d'une collection.

```
db.collectionName.find() //Renvoie la totalité des documents
```

Si l'on le souhaite, on peut bien entendu rendre notre requête plus sélective en ne récupérant qu'un seul élément via la commande `findOne(filter, options)`.

```
db.collectionName.findOne() //Renvoie le premier élément de la collection
```

Read (avec filtre)

Les méthodes de lecture comme `find()` ou `findOne()` acceptent un **filtre** pour affiner la sélection des documents.

Un filtre est un objet qui définit les critères de recherche. Il est également utilisé dans les opérations de mise à jour et de suppression, pour cibler uniquement les documents souhaités.

Par exemple :

```
//Sélectionne tous les documents dont le champ 'title' vaut "Value"
db.collection.find({ title: "Value" })
```

Read (avec filtre & tableaux)

MongoDB permet également de faire des recherches sur les champs de type tableau.

Si l'on souhaite trouver tous les documents dont un champ de type `array` et qui contient exactement les mêmes valeurs dans un ordre donné, nous pouvons écrire :

```
//Sélectionne tous les documents possédant un tableau identique au champs 'genre'
db.collection.find({ "genre": ["Action", "Aventure"] })
```

Read (curseur)

Lorsque l'on fais une requête avec `find()`, MongoDB retourne un curseur (pas tous les documents en une fois).

- Par défaut, seuls les 20 premiers documents sont affichés dans le shell.
- Pour pouvoir afficher la suite en cas d'un grand nombre de données, il nous faudra taper `it` dans le shell.

```
// On aura les 20 premiers éléments
db.collectionName.find()

it
```

Read (Transformer/Parcourir les résultats)

Nous pouvons récupérer l'ensemble des documents dans un tableau via la méthode `.toArray()`:

```
// On aura tous les éléments transformés en tableau puis affichés
db.collectionName.find().toArray()
```

Si l'on veut réaliser des opérations pour l'ensemble des données, on peut utiliser la méthode `.forEach()` en lui passant une fonction fléchée pour spécifier l'opération à effectuer.

```
// On aura tous les éléments transformés en tableau avec le 'title' mis en majuscule puis affichés
db.collectionName.find().forEach(x => {
  x.title = x.title.toUpperCase()
  printjson(x)
})
```

Les filtres (opérateur de comparaison)

Pour des sélections plus complexes, il est important de connaître les opérateurs majeurs de MongoDB dont les opérateurs de comparaison.

- `$eq` : Égalité
- `$lt` : Plus petit que
- `$lte` : Plus petit ou égal à
- `$ne` : Différence
- `$gt` : Plus grand que
- `$gte` : Plus grand ou égal à

```
//Récupère tout les documents dont le prix est supérieur 10.0
db.collection.find({ price: { $gt: 10.0 } })
```

Les filtres (opérateur logiques)

Les opérateurs logiques, tels que `$and`, `$or`, `$not` sont utilisés pour combiner des conditions.

`$and`: Chercher des documents qui répondent à toutes les conditions

```
db.users.find({ $and: [{ age: 25 }, { first_name: "John" }] })
```

`$or` : Documents qui remplissent au moins une condition.

```
db.users.find({ $or: [{ age: 25 }, { first_name: "John" }] })
```

`$not`: Exclut les documents correspondant à une condition

```
db.users.find({ age: { $not: { $eq: 25 } } })
```

Les filtres (opérateur d'éléments)

Dans les opérateurs, il existe aussi deux opérateurs d'éléments, dont l'objectif est:

- `$exists`: Le document possède ce champ dans sa structure.
- `$type`: Le document possède un champ avec le bon type dans sa structure.

```
// Pour trouver les produits qui possèdent le champ 'sale', même si la valeur est 'null'
db.product.find({ sale: { $exists: true } })

// Pour trouver tous les utilisateurs qui ont un numéro de téléphone sous forme de 'string' ou de 'double'
db.users.find({ phone: { $type: [ 'double', 'string' ] } })
```

Les filtres (opérateur dans un tableau)

Les opérateurs tels que `$in`, `$nin`, `$all` sont utilisés pour effectuer des requêtes dans des tableaux.

`$in`: Chercher des documents qui répondent à au moins une des valeurs spécifiées.

```
// Documents dont l'âge est soit 25, soit 30
db.users.find({ age: { $in: [25, 30] } })
```

`$nin` : Exclut les documents contenant l'une des valeurs spécifiées.

`$all`: Sélectionne les documents ayant toutes les valeurs spécifiées.

```
// Documents dont le tableau "tags" contient à la fois "sport" et "plein air"
db.articles.find({ tags: { $all: ["sport", "plein air"] } })
```

`$size`: Permet de tester si le tableau dispose d'un certain nombre d'éléments.

```
// Sélectionne les utilisateurs ayant exactement 2 loisirs
db.users.find( { "hobbies": { $size: 2 } } )
```

`$elementMatch`: Permet de vérifier qu'un même élément du tableau respecte plusieurs conditions.

```
// Sélectionne les restaurants ayant un score compris entre 80 et 90
// (dans un même élément du tableau)
db.restaurants.find({
  scores: { $elemMatch: { $gte: 80, $lt: 90 } }
})
```

Les filtres (expressions régulières)

L'opérateur `$regex` permet de rechercher des documents via des motifs personnalisés, similaires aux expressions régulières en JavaScript.

```
// Documents dont le nom commence par la lettre "L"
db.users.find({ name: { $regex: /^L/ } })

// Documents dont l'email contient "gmail"
// (le "i" rend la recherche insensible à la casse)
db.users.find({ email: { $regex: /gmail/, $options: "i" } })
```

Read (trier)

Il est possible de trier les résultats d'une requête grâce à la méthode `sort()`.

- Pour un **tri croissant**, on utilise `1`.
- Pour un **tri décroissant**, on utilise `-1`.

```
db.users.find().sort({ age: 1}); // Tri croissant

db.users.find().sort({ age: -1}); // Tri décroissant
```

Read (limiter et paginer)

MongoDB propose deux méthodes pour gérer la pagination des résultats :

- `limit(n)` : ne renvoie **que les n premiers** documents.
- `skip(n)` : **ignore les n premiers** documents du résultat.

```
db.users.find().limit(5) // Renvoie les 5 premiers documents
db.users.find().skip(10) // Ignore les 10 premiers documents
db.users.find().skip(10).limit(5)  // Saute les 10 premiers et prend les 5 suivants
```

La Projection

Pour éviter de récupérer des informations inutiles et alléger les requêtes, on utilise la **projection** qui permet de **choisir précisément les champs à inclure ou à exclure** dans les options.

Pour inclure un champ, on lui associe la valeur `1` et **tous les autres seront exclus**. A l'exception du champs `_id` qui est **inclus par défaut** et doit être **exclu manuellement** avec la valeur `0`.

Exemple :

```
// Ici, seul le champ name sera affiché, sans l'identifiant.
db.collectionName.find({}, { _id: 0, name: 1 })
```

Exemple de combinaison avancée

```
db.users.find(
  {
    age: 25,
    gender: "femme",
    country: { $in: ["France", "Belgique"] }
  },
  {
    first_name: 1,
    age: 1,
    country: 1,
    _id: 0
  }
).sort({ first_name: -1 }).limit(5);
```

UPDATE (modification de donnée)

Update

Si l'on souhaite modifier un ou plusieurs documents, on peut utiliser les méthodes suivantes :

- `updateOne(filter, data, options)` : Modifie le **premier document** correspondant au filtre.
- `updateMany(filter, data, options)` : Modifie **tous les documents** correspondant au filtre.
- `replaceOne(filter, data, options)` : **Remplace complètement** un document par un autre.

Pour réaliser une modification, il nous faut utiliser des opérateurs en

Update (modifier/ajouter)

Pour ajouter ou mettre à jour un champ, l'opérateur à utiliser est `$set` qui prend le nom du champs à ajouter/modifier suivis de sa valeur:

```
// Modifie le champ 'age' du document avec l'ID spécifié
db.users.updateOne({ _id: "unID" }, { $set: { age: 30 } })

// Ajoute un champ 'newField' à tous les documents (ou le modifie s'il existe)
db.users.updateMany({}, { $set: { newField: "Value" } })

// Remplace entièrement le premier document dont le champ 'name' est "John"
db.users.replaceOne({ name: "John" }, { firstName: "Jonathan", age: 40 })
```

Update (incrémenter/supprimer)

Il est possible d'incrémenter un champs numérique avec l'opérateur `$inc` en option.

```
// Incrémente de 1 tout les documents ayant un champ 'views'
db.users.updateMany({views: {$exists: true}}, {$inc: {views: 1 }})
```

Nous pouvons aussi supprimer un champs d'un document avec `$unset`

```
// Supprime le champ 'email' pour tout les documents
db.users.updateMany({}, {$unset: {email: "" }})
```

Update (tableaux)

Pour les tableaux, nous avons 2 méthodes permettant l'ajout et la suppression d'une valeur :

- `$push` : Ajoute une valeur à un tableau

```
// Ajoute une valeur 'new' dans le tableau 'tags'
db.users.updateOne({name: "Bob"}, {$push: {tags: "new"}})
```

Il est également possible d'éviter les doublons avec l'opérateur `$addToSet`

- `$pull` : Retire une valeur d'un tableau

```
// Retire la valeur 'old' dans le tableau 'tags'
db.users.updateOne({name: "Bob"}, {$pull: {tags: "old"}})
```

DELETE (suppression de donnée)

Delete

Pour supprimer un ou plusieurs documents dans MongoDB, on utilise principalement deux méthodes :

- `deleteOne(filter, options)` : **Supprime le premier document** qui correspond au filtre.

```
// Supprime le premier document dont le champ 'name' vaut "Loick"
db.users.deleteOne({ name: "Loick" })
```

- `deleteMany(filter, options)`: **Supprime tous les documents** qui correspondent au filtre.

```
// Supprime tous les documents dont l'âge est inférieur à 18
db.users.deleteMany({ age: { $lt: 18 } })
```

Si l'on veut supprimer d'un seul coup l'ensemble d'une collection, il suffit d'utiliser la méthode de supression multiple avec un filtre vide.

```
// Supprime tous les documents
db.users.deleteMany({})
```

Attention : Ces opérations sont **irréversibles** : une fois les documents supprimés, ils ne **peuvent pas être récupérés**. En bonne pratique, **toujours tester les filtres** avec un `find()` avant de faire un `deleteMany()` pour éviter de supprimer des données par erreur.

Delete (suppression DB & collection)

Pour supprimer une base de données entière, on peut utiliser la commande :

```
db.dropDatabase()
```

Si c'est la collection que l'on veut supprimer, il est possible de le faire via:

```
db.collectionName.drop()
```

Cela supprime uniquement la collection 'users', mais pas la base de données.

LES DOCUMENTS LIÉS (structures imbriquées)

Les documents liés

MongoDB étant un système NoSQL orienté documents, il permet de structurer les données de manière hiérarchique.
Il est possible :

- **D'imbriquer un document** dans un autre document (jusqu'à 100 niveaux de profondeur maximum),
- D'utiliser des tableaux comme valeur d'un champ pour stocker plusieurs objets ou valeurs,
- De stocker jusqu'à **16 Mo** de données par document (toutes structures comprises).

Exemple de document imbriqué

- Le champ `job` contient un sous-document.
- Le champ `favColors` est un tableau de chaînes de caractères.
- Le champ `dogs` est un tableau de sous-documents.

```
{
  "firstName": "John",
  "lastName": "DOE",
  "job": {
    "name": "Gardener",
    "hireDate": "2007-05-18"
  },
  "favColors": [ "Red", "Green", "Blue" ],
  "dogs": [
    {
      "name": "Zoé",
      "breed": "German Shepard",
      "age": 10,
      "isMale": false
    },
    {
      "name": "Rex",
      "breed": "Doberman",
      "age": 3,
      "isMale": true
    }
  ]
}
```

Requêter dans des sous-documents

Pour rechercher des valeurs dans un sous-document ou dans un tableau de sous-documents, on utilise la notation par chemin de propriété avec un point `.` :

```
// Recherche tous les documents où au moins un chien est un mâle
db.users.find({ "dogs.isMale": true })
```

Ce type de requête permet à MongoDB de parcourir les documents **imbriqués** ou les éléments d'un tableau pour vérifier la présence d'une correspondance.

COUNT

(compter les documents)

Compter les documents en MongoDB

MongoDB propose deux principales méthodes pour effectuer cette opération:

- `countDocuments()`: Fournit un **comptage précis** basé sur un filtre donné.
- `estimatedDocumentCount()`: Donne une **estimation rapide** du nombre total de documents, sans filtre.

Chaque méthode répond à un besoin différent en termes de **précision** et **performance**.

Compter avec countDocuments()

Cette méthode permet de **compter précisément** les documents qui correspondent à un critère donné.

```
collection.countDocuments(filter, options)

// Compter tous les documents dans la collection
db.users.countDocuments()

// Compter les utilisateurs âgés de plus de 18 ans
db.users.countDocuments({ age: { $gt: 18 } })
```

- `filter` : (facultatif) Pour filtrer les documents à compter.
- `options` : (facultatif) Options comme `skip` ou `limit`.

Compter avec estimatedDocumentCount()

Elle renvoie **une estimation rapide** du nombre total de documents dans une collection via les métadonnées internes de MongoDB.

```
// Estimer le nombre total d'utilisateurs
db.users.estimatedDocumentCount()
```

- `options`: (Facultatif) Options supplémentaires, comme `maxTimeMS` pour limiter le temps d'exécution.

Limites:

- Ne supporte **aucun filtre**.
- L'estimation peut être **inexacte** si la collection est très active (insertions/suppressions fréquentes).

INDEX

(accélérer les recherches)

Qu’est-ce qu’un index dans MongoDB ?

Un index est une structure spéciale stockée dans une collection, qui permet de retrouver rapidement les documents lors d’une recherche. Ils sont utilisés pour:

- **Accélérer les recherches:** Réduisent considérablement le temps nécessaire pour localiser les documents dans une collections.
- **Trier les résultats efficacemment:** Ils permettent à MongoDB de trier les données sans avoir à les charger intégralement.
- **Assurer l'unicité des champs:** Grâce à des index uniques, on peut imposer des contraintes sur un champ (ex: un email unique).

Mais ils ont pour inconvénient :

- **Espace disque:** Les index nécessitent un espace supplémentaire.
- **Temps de mise à jour:** Lors des opérations d'insertion, de mise à jour ou de suppression, les index aussi doivent être mis à jour, ce qui peut ralentir ces opérations.

Sans index, MongoDB est obligé de scanner chaque document d’une collection (opération coûteuse appelée COLLSCAN), ce qui peut être inefficace sur de grandes collections.

Index (création/suppression)

- Création d'un index:

```
db.users.createIndex({email: 1});
```

- Suppression d'un index:

```
// Suppression d'un index spécifique (par son nom)
db.users.dropIndex("nom_de_l'index");

// Suppression de tous les index de la collection.
db.users.dropIndexes();
```

- Vérification des index existants:

```
db.users.getIndexes();
```

Les types d'index

- **Index unique** : Permet de garantir qu'un champ est unique dans la collection (ex : email).

```
db.users.createIndex({email: 1}, {unique: true});
```

- **Index composé**: Index sur plusieurs champs : utile pour optimiser des recherches combinées.

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

Ordre des champs: MongoDB utilise l'ordre des champs pour optimiser la requête. Donc un index sur `{firstName:1, lastName:1}` est différent d'un index sur `{lastName:1, firstName:1 }`

- **Index texte**: Permet de faire des recherches en texte intégral (`$text`).

```
db.products.createIndex({description: "text"});
```

- **Index géospatial**: Optimise les requêtes sur les données géographiques.

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

- **Index partiel**: Index uniquement sur les documents qui remplissent une condition.

```
db.collection.createIndex({age: 1}, {partialFilterExpression: {age: { $gt: 18}}});
```

- **Index TTL (Time-to-live)**: C'est une fonctionnalité qui permet de supprimer automatiquement les documents d'une collection après un certain temps.

Cela est particulièrement utile pour les applications qui ont besoin de gérer des données temporaires, comme les sessions utilisateurs, les logs, ou les caches. **Ils doivent être appliqués sur un champ de type Date**.

```
db.sessions.insertOne({userId: "user123", createdAt: new Date()})
db.sessions.createIndex({createdAt: 1}, {expireAfterSeconds: 3600});
```

Vérifier les performances avec explain()

MongoDB permet de visualiser le **plan d'exécution d'une requête** avec `explain()` :

```
db.collection.find({age: 25}).explain("executionStats");
```

- **COLLSCAN**: Indique un scan complet de la collection (lent).
- **IXSCAN**: Indique que MongoDB a utilisé un index (rapide).

AGRÉGATION

(transformer et analyser les données)

Le Pipeline d'Agrégation

Le pipeline d'agrégation dans MongoDB est un mécanisme puissant qui permet de transformer, filtrer, regrouper, trier ou encore **joindre les documents** d'une collection.

L'ordre des étapes est important car chaque **étape** (appelée "stage") du pipeline **prend les documents en entrée**, les **transforme**, puis les **transmet à l'étape suivante**.

```
db.collection.aggregate([
  { $stage1: { ... } },
  { $stage2: { ... } },
  ...
])
```

Opérateurs d'Agrégation ($match)

- `$match` : Filtre les documents selon des critères. C'est souvent la première étape du pipeline, pour réduire le nombre de documents à traiter ensuite.

```
//Afficher les commandes dont le total est supérieur à 10.
db.orders.aggregate([
  { $match: { total: { $gt: 10 } } }
]);

// Produits en stock dans la catégorie "informatique"
db.products.aggregate([
  { $match: { stock: { $gt: 0 }, category: "informatique" } }
])
```

Opérateurs d'Agrégation ($project)

- `$project`: Projette des champs spécifiques et effectue des transformations.

```
// Afficher uniquement le client et son total,
// tout en ajoutant un champ calculé `taxes` (total x 0.2).
db.orders.aggregate([
  {
    $project: {
      customer: 1,
      total: 1,
      taxes: { $multiply: ["$total", 0.2] }
    }
  }
]);
```

Opérateurs d'Agrégation ($unwind)

- `$unwind` : Décompose un tableau dans une collection de documents. Il prend chaque élément d'un tableau et crée un document distinct pour chaque élément, ce qui peut être utile pour effectuer des opérations d'agrégation sur les éléments individuels du tableau.

```
// Décompose les items en documents individuels.
db.orders.aggregate([
  { $unwind: "$items" }
]);
```

Opérateurs d'Agrégation ($group)

- `$group` : Regroupe les documents par une ou plusieurs clés spécifiées et appliquer des fonctions d'agrégation sur chaque groupe de documents.

Cela permet de réaliser des opérations telles que le calcul de sommes ou de moyennes. Par exemple :

```
{ "_id": 1, "customer": "Alice", "amount": 100 }
{ "_id": 2, "customer": "Bob", "amount": 200 }
{ "_id": 3, "customer": "Alice", "amount": 150 }
```

```
// Compte le nombre de commandes par client :
db.orders.aggregate([
  {
    $group: {
      _id: "$customer",
      totalOrders: { $sum: 1 }
    }
  }
])
```

Opérateurs d'Agrégation ($sort & $addFields)

- `$sort` : Trie les documents.

```
//Trier les commandes par `total` décroissant.
db.orders.aggregate([
  { $sort: { total: -1 } }
]);
```

- `$addFields` : Ajoute de nouveaux champs ou modifie des champs existants.

```
//Ajouter un champ `orderDateYear` pour extraire l'année de la date.
db.orders.aggregate([
  { $addFields: { orderDateYear: { $year: { $dateFromString: { dateString: "$date" } } } } }
]);
```

Opérateurs d'Agrégation ($limit & $skip)

- `$limit` : Limite le nombre de documents renvoyés.

```
//Afficher les 2 premières commandes.
db.orders.aggregate([
  { $limit: 2 }
]);
```

- `$skip` : Ignore les premiers documents.

```
//Ignorer la première commande.
db.orders.aggregate([
  { $skip: 1 }
]);
```

Opérateurs d'Agrégation ($lookup)

L'opérateur `$lookup` en MongoDB est utilisé pour effectuer des jointures entre deux collections, similaire à une jointure SQL.
Il permet de combiner des documents de deux collections en une seule opération d'agrégation.

```
db.collection.aggregate([
  {
    $lookup: {
      from: <collection to join>,
      localField: <field from the input documents>,
      foreignField: <field from the documents of the from collection>,
      as: <output array field>
    }
  }
])
```

- `from` : Le nom de la collection à joindre.
- `localField` : Le champ de la collection d'entrée (celle sur laquelle vous effectuez l'agrégation) à comparer.
- `foreignField` : Le champ de la collection à joindre à comparer.
- `as` : Le nom du champ de sortie où les documents joints seront stockés.

Exemple

Supposons que vous avez deux collections : `orders` et `customers`. Vous souhaitez joindre les documents de `customers` aux documents de `orders` en fonction de l'ID du client.

- `orders` :

```
{ "_id": 1, "item": "apple", "price": 10, "quantity": 2, "customerId": 1 }
{ "_id": 2, "item": "banana", "price": 5, "quantity": 5, "customerId": 2 }
```

- `customers` :

```
{ "_id": 1, "name": "Alice", "email": "alice@example.com" }
{ "_id": 2, "name": "Bob", "email": "bob@example.com" }
```

```
db.orders.aggregate([
  {
    $lookup: {
      from: "customers",
      localField: "customerId",
      foreignField: "_id",
      as: "customerDetails"
    }
  }
])
```

```
[
  {
    "_id": 1,
    "item": "apple",
    "price": 10,
    "quantity": 2,
    "customerId": 1,
    "customerDetails": [
      {
        "_id": 1,
        "name": "Alice",
        "email": "alice@example.com"
      }
    ]
  },
  {
    "_id": 2,
    "item": "banana",
    "price": 5,
    "quantity": 5,
    "customerId": 2,
    "customerDetails": [
      {
        "_id": 2,
        "name": "Bob",
        "email": "bob@example.com"
      }
    ]
  }
]
```

Résumé des Opérateurs

Opérateur	Description
`$match`	Filtre les documents.
`$project`	Projette des champs et calcule des valeurs.
`$group`	Regroupe les documents et calcule des stats.
`$sort`	Trie les documents.
`$unwind`	Décompose des tableaux.
`$addFields`	Ajoute ou modifie des champs.
`$limit`	Limite le nombre de documents.
`$skip`	Ignore un certain nombre de documents.
`$lookup`	Joint deux collections.

Exemple d'une pipeline d'agrégation

```
db.orders.aggregate([
  // 1. Filtrer les commandes dont le total est supérieur à 10
  {
    $match: { total: { $gt: 10 } }
  },
  // 2. Décomposer les éléments du tableau `items`
  {
    $unwind: "$items"
  },
  // 3. Calculer le montant total pour chaque produit
  {
    $project: {
      product: "$items.product",
      totalAmount: { $multiply: ["$items.quantity", "$items.price"] }
    }
  },
  // 4. Regrouper les résultats par produit
  {
    $group: {
      _id: "$product",
      totalSales: { $sum: "$totalAmount" }
    }
  },
  // 5. Trier les résultats par montant total décroissant
  {
    $sort: { totalSales: -1 }
  },
  // 6. Limiter les résultats aux 2 produits les plus vendus
  {
    $limit: 2
  }
]);
```

Résultat final :

```
[
  { "_id": "Pencil", "totalSales": 15 },
  { "_id": "Notebook", "totalSales": 10 }
]
```

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

CRUD avancé

Create - Importer des données

Si l'on veut importer des éléments directement dans notre base de données, avec par exemple un fichier JSON, il est possible de le faire via `mongoimport`. Cette commande est disponible en cas d'installation du binaire et son ajout à la variable d'environnement `PATH`:

```
mongoimport file.json -d databaseName -c collectionName --jsonArray --drop
```

Create - Importer des données

Les options sont les suivantes:

- `-d`: Permet d'informer de la database que l'on cherche à peupler
- `-c`: Permet d'informer de la collection que l'on cherche à peupler
- `--jsonArray`: Permet d'informer que l'on aura plusieurs éléments à insérer sous la forme d'un tableau
- `--drop`: Permet la suppression de la collection au préalable pour être sur de partir sur une ré-écriture et non un ajout

Read - Les opérateurs d'évaluation

Un autre opérateur d'évaluation, `$expr`, permet le calcul de valeurs via des expressions. Il s'utilise ainsi :

```
// Obtenir les éléments dont le prix a baissé, en comparant l'ancien et le nouveau prix
db.products.find({ $expr: { $gt: [ "$oldPrice", "$newPrice" ]}})

// Pour avoir les produits dont le stock est : - soit inférieur à 150 tonnes,
// - soit supérieur à 150 tonnes, on y soustrait alors 50 tonnes
db.products.find({ $expr: { $gt: [ $cond: { if: { $gte: [ "$stock", 150 ] },
    then: { $substract: [ "$stock", 50 ]} },
    else:  "$stock",
    "$target" ]
  }})
```

Read - Fonctions usuelles

Dans le cadre de la projection et des tableaux, il est possible d'utiliser l'opérateur `$slice` qui a pour objectif de projeter une tranche du tableau, par exemple seulement les éléments de l'indice X à Y.

```
// On prend les 3 premiers éléments du tableau
db.movies.find({ rating: { $gte: 70 } }, { genres: { $slice: 3 }}})

// On passe les 2 premiers éléments du tableaux puis on prend les 4 suivants
db.movies.find({ rating: { $gte: 70 } }, { genres: { $slice: [ 2, 4 ] } })
```

Update - Éditer nos documents

Via les opérateurs `$min` et `$max`, il est possible de changer des valeurs numériques de façon conditionnelle, de sorte à avoir une valeur ne dépassant par un seuil saisi par le développeur:

```
// Les éléments de la collection verront leur prix modifié à 19.89
// si leur prix précédent était d'une valeur dépassant ce nombre
dc.collection.updateMany({}, { $max: { price: 19.89 } })
```

Update - Éditer nos documents

Il est également possible de procéder à des calculs simples via les opérateurs mathématiques :

- `$add`: Pour additioner la valeur actuelle à un certain nombre.
- `$sub`: Pour soustraire la valeur actuelle par un certain nombre.
- `$mul`: Pour multiplier la valeur actuelle par un certain nombre.
- `$div`: Pour diviser la valeur actuelle par un certain nombre.

Update - Éditer nos documents

Il est aussi possible de renommer nos champs via l'opérateur `$rename`.
Son fonctionnement est le suivant : il faut lui passer un objet avec un ensemble de clés-valeurs, avec comme clé l'ancien nom du champ, et comme valeur son nouveau nom

```
db.collection.updateMany({ ... }, { $rename: { oldName: "newName" } })
```

Update - Éditer nos documents

La méthode d'édition est compatible avec une insertion en cas d'utilisation de l'option `upsert` (qui par défaut a pour valeur false) de la sorte:

```
db.collection.updateOne({ ... }, { ... }, { upsert: true })
```

Update - Travailler avec des tableaux

Dans le cadre de la modification de champs qui sont dans des tableaux faisant eux-même partie des champs d'un document, la méthodologie de la modification des valeurs diffère légèrement.
Pour l'ajout d'un champ dans l'un des documents contenus dans le tableau X des documents trouvés, il est possible de procéder ainsi :

```
// On ajoute ici le champ 'units' aux document imbriqués se trouvant dans le tableau des ingrédients de nos recettes
db.recipes.updateMany({ ingredients: $elemMatch: { name: "Tomato", quantity: 3 } }, { $set: { "ingredients.$.units": "kilograms" }})

// On remplace le premier élément imbriqué respectant les critères du filtre et on le remplace par un nouveau document imbriqué
db.recipes.updateMany({ ingredients: $elemMatch: { name: "Tomato", quantity: 3 } }, { $set: { "ingredients.$": { field: "Value"} }})
```

Update - Travailler avec des tableaux

Si l'on veut cette fois-ci changer tous les éléments imbriqués d'un tableau d'un coup, la syntaxe ressemblera à celle-ci:

```
// Via cette requête, on va incrémenter de 5 tous les éléments imbriqués du champ 'ingredient' ayant un champ 'quantity' d'une valeur de 5
db.recipes.updateMany({ ingredients: $elemMatch: { name: "Tomato", quantity: 3 } }, { $inc: { "ingredients.$[].quantity": 5 }})
```

Update - Travailler avec des tableaux

Si, au sein d'un tableau contenant des éléments imbriqués, nous souhaitons désormais filtrer et modifier seulement certains champs, et non le premier ou tous, il nous faut utiliser l'option `arrayFilter`, qui va donner un identifiant utilisable dans la syntaxe vue précédemment pour ne modifier qu'une portion des éléments imbriqués:

```
// Si l'on veut ajouter le champs 'isVegan' avec pour valeur 'true' pour tous les éléments imbriqués du champ ingrédient de nos documents, qui est un array
db.recipes.updateMany({ "ingredients.quantity": { $gte: 2} }, { $set: { "ingredients.$[el].isVegan": true }, { arrayFilters: [ { "el.quantity": { $gte: 2 } } ]}})
```

Update - Travailler avec des tableaux

Pour ajouter ou supprimer des éléments du tableau imbriqué, on peut utiliser les opérateurs suivants :

```
// Pour ajouter un élément imbriqué
db.recipes.updateOne({ name: "Tarte" }, { $push: { ingredients: { title: "flour", quantity: 5}}})

// Pour ajouter plusieurs éléments imbriqués d'un coup
db.recipes.updateOne({ name: "Tarte" }, { $push: { ingredients: { $each: [ { title: "flour", quantity: 5}, { title: "egg", quantity: 3}]}}})

// Il est possible d'ajouter des fonctionnalités supplémentaires, par exemple :  trier les éléments imbriqués avant leur ajout au tableau
db.recipes.updateOne({ name: "Tarte" }, { $push: { ingredients: { $each: [ { title: "flour", quantity: 5}, { title: "egg", quantity: 3 } ], $sort: { title: 1 }}}})
```

Si l'on le veut, il est possible d'éviter les doublons en usant à la place de l'opérateur `$addToSet`.

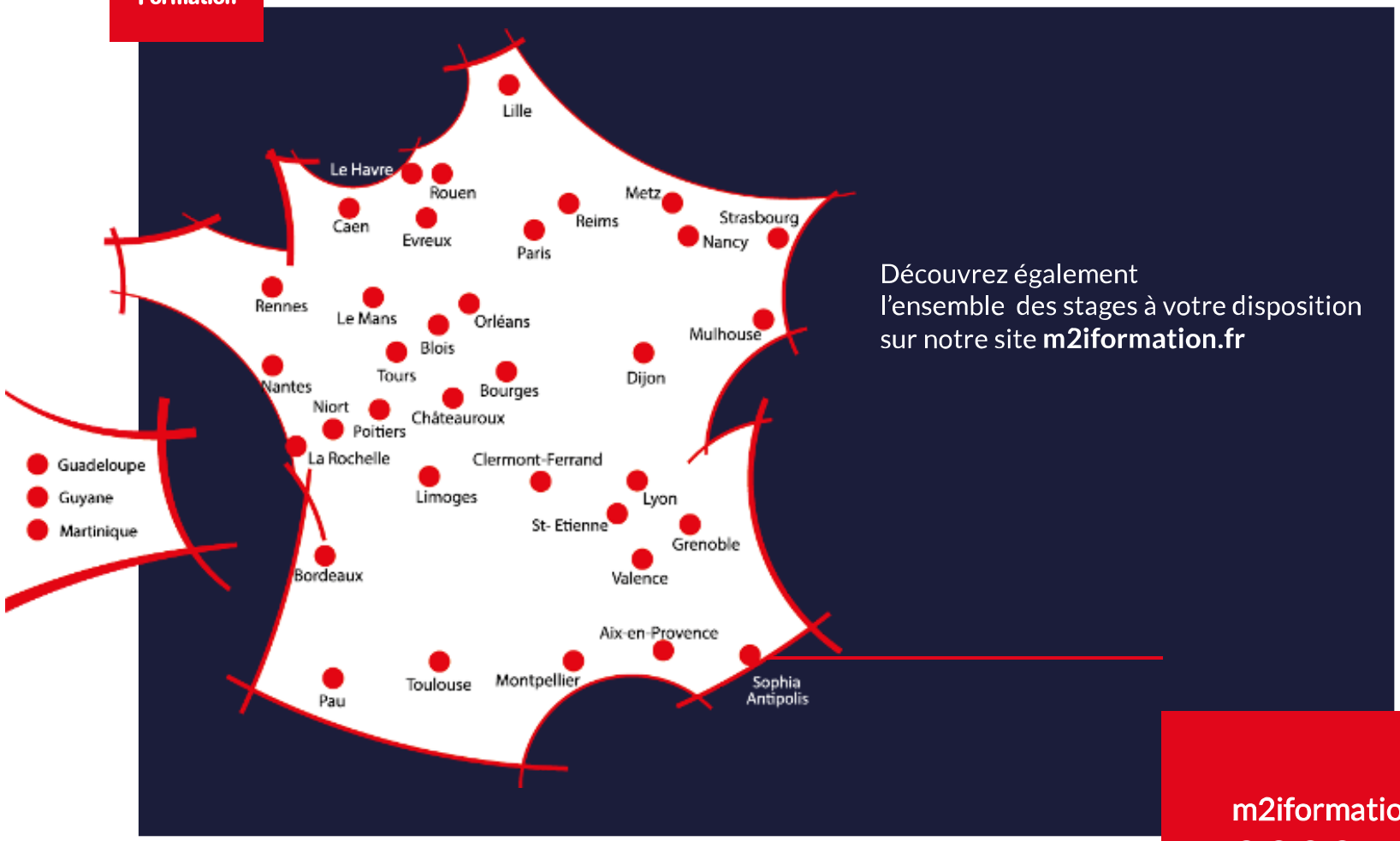
Update - Travailler avec des tableaux

Pour supprimer les éléments d'un tableau, il est possible de le faire via l'opérateur `$pull` ou `$pop` de la façon suivante :

```
// On va retirer, dans les documents et dans leurs champs ingrédients, les éléments correspondant à de la farine
db.recipes.updateOne({ name: "Tarte" }, { $pull: { ingredients: { title: "flour" } } })

// On supprime le premier élément du tableau
db.recipes.updateOne({ name: "Tarte" }, { $pop: { ingredients: 1 } })

// On supprime le dernier élément du tableau
db.recipes.updateOne({ name: "Tarte" }, { $pop: { ingredients: -1 } })
```



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr