Dossier 4
MongoDB

MongoDB est un SGBD orienté documents écrit en C++. Il est développé par la société MongoDB Inc (anciennement 10gen) et est utilisé par de nombreuses entreprises comme par exemple ebay, Le Figaro, Axa Assurance, Bouygues Telecom, etc. MongoDB est d'ailleurs le SGBD NoSQL le plus populaire. D'abord parce que la représentation des données en documents JSON (JavaScript Object Notation) permet de stocker directement des flux provenant d'objets connectés, sans avoir à modifier la structure des données. Ensuite car cette structure n'a pas besoin d'être forcément prédéfinie (MongoDB peut être utilisé sans schéma ou schema-less) ce qui donne une certaine flexibilité permettant de stocker des données hétérogènes qui ne sont pas forcément structurées ; et grâce à cette flexibilité MongoDB peut être utilisé dans des domaines métier très différents, et facilite aussi le développement itératif-incrémental. De plus, MongoDB propose une simplicité de développement aux applications clientes qui plait généralement aux développeurs. Et enfin parce qu'il permet une montée en charge horizontale au fur et à mesure de l'augmentation des besoins.

En pratique MongoDB est parfois utilisé dans des projets qui gèrent des gros volumes de données et où des bases de données relationnelles ne pourraient pas être utilisées. Mais on retrouve aussi MongoDB dans des projets plus modestes qui gèrent beaucoup moins de données. En effet, MongoDB est beaucoup plus flexible que d'autres SGBD NoSQL et peut aussi être utilisé lorsqu'on gère peu d'informations (alors qu'il ne nous viendrait pas à l'idée d'utiliser Cassandra pour stocker un petit volume de données). Toutefois, MongoDB ne gèrera pas la cohérence des données aussi bien qu'une base de données relationnelle, et ce sera aux développeurs de programmer les vérifications qui ne pourront pas être faites par MongoDB.

MongoDB possède une architecture centralisée. La connexion du client est réalisée sur un serveur maitre qui redirige la requête sur le bon nœud. En cas de défaillance du nœud maitre, MongoDB promeut automatiquement un autre nœud maitre. Les applications clientes sont averties et redirigent les écritures vers le bon maitre. Afin d'assurer la disponibilité des données, MongoDB duplique automatiquement les données. Les écritures ont une option pour indiquer qu'elles sont validées si elles sont écrites sur un certain nombre de réplicas ou sur le quorum (la moitié plus un). Les dernières versions de MongoDB gèrent l'acidité des transactions. La durabilité est par défaut assurée à travers l'activation d'un journal (mais il est possible de désactiver cette option pour accélérer les requêtes). Depuis la version 4.2, il est possible d'isoler et de rendre atomique une transaction multi-documents et multi-documents distribuée (un verrou est posé sur le document en question).

1 Concepts de base

1.1 Stockage des données dans MongoDB

MongoDB est un SGBD orienté documents. Ces documents sont regroupés dans des collections (sortes de tables) et possèdent une clé qui permet de les identifier dans la collection. Cette clé est stockée dans l'élément `_id` du document. Sa valeur peut être fournie par l'utilisateur ou générée automatiquement par MongoDB si elle n'est pas renseignée. Les documents sont stockés au format BSON qui est un JSON binarisé plus compact et plus pratique. Mais pour l'utilisateur, la structure visible est le JSON.

> Le format JSON (JavaScript Object Notation) provient du JavaScript même s'il peut être manipulé indépendamment de ce langage. Il peut faire penser à du XML mais il est beaucoup moins verbeux et plus facile à lire pour une personne humaine et surtout plus facile à manipuler. Avec JSON, les traitements sont généralement plus rapides, mais il n'est pas possible de faire des vérifications sur le format du fichier (comme par exemple avec une DTD sur un document XML).
>
> Un document JSON permet de décrire des objets grâce à un ensemble de `"clé" : "valeur"`. Une valeur peut être une chaîne de caractères entre doubles guillemets `" "`, ou alors un objet décrit lui-même par un ensemble de `"clé" : "valeur"` encadrées par des accolades `{ }`. Il est possible de décrire une collection d'objets (ou tableau d'objets) grâce à des crochets `[ ]`.

Exemple d'une collection de documents JSON 'films' qui permet de stocker les informations des films.

```
[{
	"_id": "F1",
	"titre": "Les Huit salopards",
	"genre": "Exotique",
	"dateSortie": {
		"jour": 6,
		"mois": 1,
		"annee": 2016
	},
	"acteurs": [{
		"id": "A156",
		"nom": "Bichir",
		"prenom": "Demian",
		"age": 52
	}, {
		"id": "A345",
		"nom": "Jackson",
		"prenom": "Samuel L",
		"age": 67
	}, {
		"id": "A124",
		"nom": "Russell",
		"prenom": "Kurt",
		"age": 64
	}]
}, {
	"_id": "F2",
	"titre": "XP, le film",
	"dateSortie": {
		"jour": 16,
		"mois": 1,
		"annee": 2018
	},
	"acteurs": [{
		"id": "A515",
		"nom": "Palleja",
		"prenom": "Xavier",
		"age": 28
	}, {
		"id": "A672",
		"nom": "Palleja",
		"prenom": "Nathalie",
		"age": 32
	}]
}, {…}, {…}, {…}]
```

premier document JSON de la collection films ; ayant pour `_id` "F1"

deuxième document JSON de la collection films ;

Contrairement aux dernières versions de Cassandra, MongoDB peut être utilisé sans schéma (schéma-less). Les documents n'ont alors pas de structure prédéfinie. Cela donne de la souplesse (on n'est pas obligé d'avoir un genre pour chaque film ; cela évite à avoir à gérer des `null`), mais ce sera aux applications clientes de s'assurer que les données sont cohérentes (par exemple, un film ne doit pas avoir d'adresse).

> En théorie, on pourrait placer dans la même collection, des documents JSON qui n'ont pas du tout la même structure (par exemple des films et des PokemonXP). Mais on évite généralement de faire cela, et on tente de garder les collections cohérentes afin de ne pas compliquer les futures requêtes et de ne pas dégrader les performances (il n'est pas possible d'indexer une collection hétérogène).

1.2 Modélisation des données sous MongoDB

Comme avec Cassandra, il faut structurer les informations d'une base de données MongoDB en fonction des requêtes qu'on souhaite réaliser. Ainsi, lorsqu'on stocke des données imbriquées dans un document JSON, il faut bien réfléchir au point d'entrée que l'on va choisir pour le document (dans l'exemple précédent, veut-on des acteurs comprenant une liste de films ou bien plutôt des films comprenant une liste d'acteurs). Cette problématique ressemble à celles que l'on a lorsqu'on fait de la conception orientée objet et qu'on doit implémenter une association d'un diagramme de classes d'analyse.
Une fois qu'on a fait ce choix, il y a trois stratégies pour structurer des données dans des documents JSON :

1. On stocke dans un seul document, ou alors dans plusieurs documents d'une même collection l'intégralité des données dont la requête a besoin, grâce à divers types de données, notamment des tableaux et des documents imbriqués. Il n'est alors pas utile de créer une collection propre pour ces éléments imbriqués.

Cette solution simplifie les requêtes d'extraction qui se feront sur une seule collection et améliore les performances (pas besoin de faire de jointures). Elle est à privilégier lorsqu'on manipule un grand volume de données et où il est important d'accélère les requêtes. Toutefois, cette solution va généralement engendrer de la redondance ; notamment pour les associations plusieurs-plusieurs (mais pas uniquement). Et si on est amené à réaliser une nouvelle requête, il faudra peut-être créer une autre collection qui éventuellement dupliquera les données de la première collection (cf. Cassandra).

Ce cas est illustré dans l'exemple précédent films-acteurs. Les recherches d'informations sur les acteurs qui jouent dans un film vont pouvoir se faire en une seule requête. Mais si un acteur joue dans plusieurs films, on va devoir répéter plusieurs fois ses informations dans la structure, notamment son âge qui est une donnée non-stable. Si nous avions plutôt imbriqué les films dans les acteurs, nous n'aurions pas eu ce problème car, dans notre exemple, les données sur les films paraissent stables ; toutefois cette structure ne serait pas appropriée si on veut chercher les acteurs qui jouent dans un film ; sauf si on gère un petit volume de données.

2. On stocke dans le document uniquement les données stables des éléments imbriqués. Pour récupérer les données non-stables, on interrogera une autre collection où les informations ne sont pas dupliquées.

On ne stocke pas les âges des acteurs dans le film.

```
[{
    "_id": "F1",
    "titre": "Les Huit salopards",
    "acteurs": [{
        "id": "A345",
        "nom": "Jackson",
        "prenom": "Samuel L"
    },{
        "id": "A124",
        "nom": "Russell",
        "prenom": "Kurt"
    }]
}, {…}, {…}]
```

Collection des acteurs

```
[{
    "_id": "A345",
    "nom": "Jackson",
    "prenom": "Samuel L"
    "age": 67

},{
    "_id": "A124",
    "nom": "Russell",
    "prenom": "Kurt"
    "age": 64
},
{…}]
```

Cette solution intermédiaire permet de réaliser la plupart des requêtes sur une seule collection tout en ne dupliquant pas les données non stables. Par contre certaines requêtes peuvent nécessiter de faire des jointures ce qui peut dégrader les performances si on a beaucoup de données.

3. On ne stocke dans le document qu'une collection d'identifiants pour qu'il n'y ait pas de redondance du tout. Mais il faudra faire systématiquement des jointures (ou deux requêtes : une première pour récupérer les identifiants des acteurs et une seconde pour récupérer leurs données dans une autre collection).

On ne stocke que les numéros des acteurs.

```
[{
    "_id": "F1",
    "titre": "Les Huit salopards",
    "acteurs": ["A345","A124"]
}, {…}, {…}]
```

Collection des acteurs

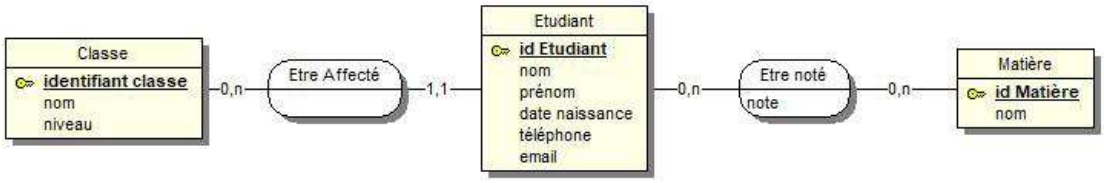
```
[{
    "_id": "A345",
    "nom": "Jackson",
    "prenom": "Samuel L"
    "age": 67

},{
    "_id": "A124",
    "nom": "Russell",
    "prenom": "Kurt"
    "age": 64
},
{…}]
```

Cette solution se rapproche de ce qui est fait dans les bases de données relationnelles où les identifiants renseignés dans la première collection (tableau d'acteurs) sont des sortes de clés étrangères qui référencent les identifiants de la seconde collection. Cette solution est très utile pour modéliser des associations réflexives ou bien bidirectionnelles dans un diagramme de classes. Mais si elle est utilisée systématiquement afin de mimer une base de données relationnelle, elle peut dégrader les performances.

Dans la documentation officielle de MongoDB, il est indiqué que la meilleure pratique de modélisation consiste à favoriser le plus possible la solution 1. Toutefois lorsqu'on a des données non stables, et des associations réflexives ou bidirectionnelles, on peut aussi être amené à utiliser les solutions 2 et 3.

Exemple : on souhaite modéliser dans une base de données MongoDB les informations du Modèle Entités/Associations suivant.



Les utilisateurs du système souhaitent pouvoir récupérer toutes les informations qui concernent un étudiant (nom, prénom, …) mais aussi les informations sur sa classe ainsi que ses notes.

La recherche se faisant par rapport à des étudiants, on peut créer une collection d'étudiants. Comme les données sur les classes et les matières sont suffisamment stables, et que les requêtes que l'on veut réaliser ne nécessitent pas qu'elles soient bidirectionnelles, il n'est pas utile de créer une collection de matières ou une collection de classes. On adopte alors la première solution et on imbrique dans chaque étudiant, de façon redondante, l'intégralité des informations sur les matières qu'il a passées et la classe dans laquelle il est affecté.

```
[{
	"_id": "E20",
	"nom": "Zétofrais",
	"prenom": "Mélanie",
	"dateNaissance": {
		"jour": 30,
		"mois": 10,
		"annee": 1992
	},
	"email": "51@gmail.com",
	"classe": {
		"id": "C1",
		"nom": "LP APIDAE",
		"niveau": "L3",
	},
	"resultats": [{
		"id": "M1",
		"nomMatiere": "UML",
		"note": 4
	},{
		"id": "M2",
		"nomMatiere": "XML",
		"note": 12
	}]
}, {…} ]
```

1.3 Réaliser des requêtes sous MongoDB

1.3.1 Se connecter à une base de données

Il est possible de voir la liste des bases de données avec la commande `SHOW dbs`

On peut sélectionner une base de données (ou la créer si elle n'existe pas) avec l'instruction suivante : **`use db_palleja`**

1.3.2 Consulter, créer ou supprimer une collection

On peut voir la liste des collections de la base de données sélectionnée avec la commande suivante : **`db.getCollectionNames()`**.

Pour créer une collection d'étudiants (collection *etudiants*) on peut éventuellement utiliser l'instruction suivante : **`db.createCollection("etudiants")`**.

Mais cette instruction est surtout utile lorsqu'on veut associer des règles de validation à la collection (un schéma) ou bien une taille maximale. Si on souhaite créer une collection sans schéma, il n'est pas obligatoire de la créer avant d'insérer des documents (la collection sera créée lors la première insertion).

Enfin, on peut supprimer ou renommer une collection avec les instructions qui suivent :

```
db.etudiants.drop()
db.etudiants.renameCollection("sauvageons")
```

1.3.3 Les requêtes de mise à jour

Pour ajouter, modifier ou supprimer des documents dans une collection, il est possible d'appliquer sur une collection les opérations : `insertOne()`, `insertMany()`, `updateOne()`, `updateMany()`, `replaceOne()`, `deleteOne()`, `deleteMany()`, `...`

Par exemple, les instructions suivantes permettent de rajouter des étudiants dans la collection *etudiants*. Si l'`_id` n'est pas renseigné, MongoDB lui attribut automatiquement une valeur.

a ➢
```
db.etudiants.insertOne(
   {"_id": "E10", "nom":"Onbenne", "prenom": "Camille",
                  "dateNaissance": {"jour":1, "mois":1, "annee":2004},
                  "resultats": [{"id": "M1", "nomMatiere": "UML", "note": 8}]}
)
Ou bien
var unEtudiant = {"_id": "E10", "nom":"Onbenne", "prenom": "Camille", ... };
db.etudiants.insertOne(unEtudiant);
```

b ➢
```
db.etudiants.insertMany([
   {"_id": "E20", "nom":"Zétofrais", "prenom": "Mélanie",
                  "dateNaissance": {"jour":2, "mois":9, "annee":2005},
                  "email": "51@gmail.com",
                  "classe": {"id": "C2", "nom": "LP APIDAE", "niveau": "L3"},
                  "resultats": [{"id": "M1", "nomMatiere": "UML", "note": 4},
                                {"id": "M2", "nomMatiere": "XML", "note": 12}]},
   {"_id": "E30", "nom": "Delune", "prenom": "Claire",
                  "dateNaissance": {"jour": 30, "mois": 10, "annee": 2004},
                  "classe": {"id": "C1", "nom": "LP ACPI", "niveau": "L3"},
                  "resultats": [{"id": "M3", "nomMatiere": "BD", "note": 4},
                                {"id": "M1", "nomMatiere": "UML", "note": 11}]}
])
```

Pour modifier un document, on peut utiliser l'opération `updateOne`. Si plusieurs documents correspondent à la condition indiquée, seul le premier sera modifié. Sinon, il faut utiliser l'opération `updateMany`. Si on rajoute l'option `upsert` (mélange entre update et insert) une insertion est réalisée si aucun document ne correspond à la condition indiquée.

c ➢
```
db.etudiants.updateOne(
   {"_id": "E10"},
   {$set:{"nom": "Onaite"}}
)
```

d ➢
```
db.etudiants.updateOne(
   {"_id": "E100"},
   {$set:{"nom": "Palleja", "prenom": "Xavier",
          "dateNaissance": {"jour":15, "mois":12, "annee":1981}}},
   {upsert: true}
)
```

La suppression d'un document peut se faire avec l'instruction `deleteOne` ou `deleteMany`.

e ➢
```
db.etudiants.deleteOne(
   {"_id": "E100"}
)
```

1.3.4 Les requêtes d'extraction de données avec l'opération `find()`

On peut sélectionner les documents d'une collection avec l'opération `find()` ; s'il n'y a pas de paramètre, tous les documents de la collection sont sélectionnés. Le résultat obtenu est un curseur sur lequel il est possible d'appeler les opérations `limit()` et `skip()` pour paginer le résultat et `sort()` pour le trier. `1` indique alors que les données sont triées par ordre croissant et `-1` par ordre décroissant.

Il est également possible de compter le nombre d'éléments qu'il y a dans le curseur avec l'opération `count()`.

f ➢ **`db.etudiants.find()`**
retourne tous les étudiants de la collection.

g ➢ **`db.etudiants.find().count()`**
retourne le nombre d'étudiants de la collection.

h ➢ **`db.etudiants.find().limit(5)`**
retourne les 5 premiers étudiants de la collection.

i ➢ **`db.etudiants.find().skip(2)`**
retourne tous les étudiants de la collection, à partir du troisième.

j ➢ **`db.etudiants.find().skip(2).limit(2)`**
retourne les troisième et quatrième étudiants de la collection.

k ➢ **`db.etudiants.find().sort({"nom": 1})`**
retourne tous les étudiants classés dans l'ordre ascendant de leur nom.

l ➢ **`db.etudiants.find().sort({"nom": -1})`**
retourne tous les étudiants classés dans l'ordre descendant de leur nom.

m ➢ **`db.etudiants.find().sort({"nom": 1, "prenom": 1})`**
retourne tous les étudiants classés dans l'ordre ascendant de leur nom. En cas de même nom, ils sont triés par le prénom.

La sélection

Pour réaliser une sélection de certains documents de la collection, il suffit d'indiquer dans le `find()`, entre `{`accolades`}` la condition sur laquelle porte la recherche. Pour les inégalités, il faut utiliser les opérateurs `$gt` pour plus grand ; `$gte` pour plus grand ou égal ; `$lt` pour plus petit ; `$lte` pour plus petit ou égal ; `$ne` pour différent (pour l'égalité on peut utiliser `$eq` ou bien mettre directement la valeur après le symbole `:`).

n ➢
```
db.etudiants.find(
   {"nom": "Palleja"}
)
```
retourne tous les étudiants dont le nom est Palleja.

o ➢
```
db.etudiants.findOne(
   {"nom": "Palleja"}
)
```
retourne le premier étudiant dont le nom est Palleja.

p ➢
```
db.etudiants.find(
   {"nom": "Palleja"}
).count()
```
retourne le nombre d'étudiants dont le nom est Palleja.

q ➢
```
db.etudiants.find(
   {"dateNaissance.annee": 2004}
)
```
retourne tous les étudiants qui sont nés en 2004.

r ➢
```
db.etudiants.find(
   {"dateNaissance.annee": {$gte: 2004}}
)
```
retourne tous les étudiants qui sont nés en 2004 ou après.

s ➢
```
db.etudiants.find(
   {"dateNaissance.annee": {$ne: 2004}}
)
```
retourne tous les étudiants qui ne sont pas nés en 2004.

Les opérateurs logiques

Lorsqu'il y a plusieurs conditions dans une requête, on peut utiliser les opérateurs logiques `$and` et `$or`. Il est également possible d'utiliser les opérateurs logiques `$not` (pour la négation d'une condition) et `$nor` (opérateur logique `NOR` – c'est à dire `A NOR B` = $\overline{\texttt{A OU B}}$ = $\overline{\texttt{A}}$ `ET` $\overline{\texttt{B}}$).

t ➢
```
db.etudiants.find(
   {$and: [{"dateNaissance.annee": 2004}, {"classe.nom": "LP APIDAE"}]}
)
```
retourne tous les étudiants qui sont nés en 2004 **et** qui sont dans la LP APIDAE.

On peut aussi écrire la requête comme ci-dessous, en remplaçant le `$and` par une virgule
```
{"dateNaissance.annee": 2004 , "classe.nom": "LP APIDAE"}
```
Mais attention, cette notation est possible uniquement si les deux conditions dans le **et** ne portent pas sur le même attribut. Par exemple, on ne peut pas remplacer le `$and` suivant par une virgule :
```
{$and: [{"dateNaissance.annee": {$lte: 2006}},{"dateNaissance.annee": {$gte: 2004}} ]}
```

u ➢
```
db.etudiants.find(
   {$or: [{"dateNaissance.annee": 2004}, {"classe.nom": "LP APIDAE"}]}
)
```
retourne tous les étudiants qui sont nés en 2004 **ou** qui sont en LP APIDAE.

v ➢
```
db.etudiants.find(
   {$or: [
      {$and:[{"nom": "Palleja"}, {"prenom": "Nathalie"}]},
      {$and:[{"nom": "Macron"}, {"prenom": "Brigitte"}]} ]}
)
```
retourne les étudiants qui s'appellent Nathalie Palleja ou Brigitte Macron.

w ➢
```
db.etudiants.find(
   {"nom": {$not: {$eq: "Palleja"} }}
)
```
retourne les étudiants qui ne s'appellent pas Palleja.

x ➢
```
db.etudiants.find(
   {$nor: [
      {$and:[{"nom": "Palleja"}, {"prenom": "Nathalie"}]},
      {$and:[{"nom": "Macron"}, {"prenom": "Brigitte"}]} ]}
)
```
retourne les étudiants qui ne s'appellent ni Nathalie Palleja ni Brigitte Macron.

Les tableaux

On peut connaitre la taille d'un tableau avec l'opérateur `$size`. Mais il n'est pas possible d'appliquer les opérateurs de comparaison `$lt, $lte, $gt`, ... sur cette taille.

Lorsqu'un tableau contient des objets composés de plusieurs attributs et que la condition logique d'une expression doit porter simultanément sur plusieurs attributs d'un même objet, il faut utiliser impérativement `$elemMatch`

y ➢
```
db.etudiants.find(
   {"resultats": {$size: 2}}
)
```
retourne les étudiants qui ont deux notes.

z ➢
```
db.etudiants.find(
   {$or: [
      {"resultats": {$size: 2}},
      {"resultats": {$size: 1}},
      {"resultats": {$size: 0}}]}
)
```
retourne les étudiants qui ont deux notes ou moins.

a ➢
```
db.etudiants.find(
   {"resultats.note": {$gte: 15}}
)
```
retourne tous les étudiants qui ont eu au moins une note supérieure ou égale à 15 (mais dans leur tableau de notes ils peuvent éventuellement avoir aussi une autre note qui est inférieure à 15).

b ➢
```
db.etudiants.find(
   {"resultats.note": {$not: {$lt: 15}}}
)
```
retourne tous les étudiants qui n'ont pas de note inférieure à 15 : c'est-à-dire qu'ils n'ont que des notes supérieures ou égales à 15. Cela ne donnera pas le même résultat que la requête 'a' car un étudiant peut avoir plusieurs notes (résultats est un tableau).

c ➢
```
db.etudiants.find(
   {"resultats.nomMatiere":"Prog","resultats.note":18}
)
```
retourne tous les étudiants qui ont un résultat en programmation **et** qui ont obtenu un 18.

Ce n'est pas la même chose que d'avoir les étudiants qui ont eu 18 en programmation (avec la requête précédente, on va récupérer les étudiants qui ont eu 18 en BD et 4 en Prog).

d ➢
```
db.etudiants.find(
   {"resultats": {$elemMatch: {"nomMatiere": "Prog", "note": 18}}}
)
```
retourne tous les étudiants qui ont eu 18 en Prog.

Attention, il y a ici un **et** dans le `$elemMatch`. Et comme nous l'avons vu précédemment, si les deux conditions du **et** portaient sur le même attribut, il faudrait utiliser l'opérateur `$and`.

Les opérateurs ensemblistes

Il est possible d'utiliser les opérateurs ensemblistes `$in $all` et `$nin` (pour NOT IN).

e ➢
```
db.etudiants.find(
   {"resultats.nomMatiere": {$in: ["XML","BD"]}}
)
```
retourne tous les étudiants qui ont un résultat en XML **ou** en BD.

f ➢
```
db.etudiants.find(
   {"resultats.nomMatiere": {$all: ["XML","BD"]}}
)
```
retourne tous les étudiants qui ont un résultat en XML **et** en BD.

g ➢
```
db.etudiants.find(
   {"resultats.nomMatiere": {$nin: ["XML","BD"]}}
)
```
retourne tous les étudiants qui ont un résultat **ni** en XML, **ni** en BD.

L'opérateur exists

L'opérateur `$exists` permet de savoir si on document contient un élément.

h ➢
```
db.etudiants.find(
   {"telephone": {$exists: true}}
)
```
retourne tous les étudiants qui ont un téléphone.

i ➢
```
db.etudiants.find(
   {$nor: [{"telephone": {$exists: true}}, {"email": {$exists: true}}]}
)
```
retourne tous les étudiants qui n'ont pas un téléphone et qui n'ont pas un email.

Les expressions régulières

Les expressions régulières permettent d'indiquer des conditions de recherche par rapport au format d'une chaîne de caractères. En pratique elles sont très utiles pour savoir si un attribut commence, se termine ou contient une chaîne de caractères (comme le `LIKE` en SQL).

`/a/` permet de trouver les chaînes qui contiennent 'a'. `/a$/` permet de trouver celles qui se terminent par un 'a', et `/^a/` celles qui commencent par un 'a'. `/a.*a/` permet de trouver les chaînes qui contiennent deux 'a' même s'ils ne sont pas accolés. Alors que `/aa/` permet de trouver celles qui contiennent 'aa'. Si on place un `i` à la fin de l'expression régulière, cela signifie que la recherche n'est pas sensible à la casse.

j ➢
```
db.etudiants.find(
   {"nom": /palleja/i}
)
```
retourne les étudiants dont le nom contient 'palleja' avec ou sans majuscule.

La projection

Pour réaliser une projection, il suffit de rajouter un second argument à l'opération `find()`. On peut alors spécifier les attributs qu'on souhaite avoir dans le résultat (en indiquant pour ces attributs toute autre valeur que `0` ou `null` – par exemple `1`) ou alors en spécifiant tous les attributs qu'on ne souhaite pas avoir dans le résultat (en indiquant `0` ou `null`). A moins d'indiquer explicitement le contraire, l'identifiant `_id` sera toujours retourné.
Une alternative est d'utiliser l'opération `projection()` sur le curseur produit par le `find()`, mais cette façon de faire est moins utilisée en pratique.

k ➢
```
db.etudiants.find(
   {"resultats.nomMatiere": "UML"},
   {"nom": 1, "prenom": 1}
)
```
```
db.etudiants.find(
   {"resultats.nomMatiere": "UML"},
).projection({"nom": 1, "prenom": 1})
```
retourne l'identifiant, le nom et le prénom de tous les étudiants qui ont un résultat en UML.

l ➢
```
db.etudiants.find(
   {"_id": "E10"},
   {"resultats.nomMatiere": 1}
)
```
retourne l'identifiant de l'étudiant E10 ainsi que le nom des matières qu'il a passées.

m ➢
```
db.etudiants.find(
   {},
   {"nom": 0, "prenom": 0}
)
```
retourne tous les attributs à l'exception du nom et du prénom de tous les étudiants de la collection.

n ➢
```
db.etudiants.find(
   {"classe": {$exists: true}},
   {"nom": 1, "prenom": 1, "_id": 0}
)
```
retourne uniquement le nom et prénom (et pas l'identifiant) des étudiants qui ont une classe.

1.3.5 Autres fonctions pour extraire des données

En plus de l'opération `find()` il est aussi possible d'utiliser sur une collection les opérations `distinct()` et `countDocuments()`. La fonction `distinct()` ne retourne pas un curseur mais un tableau. Si on veut connaitre la taille de ce tableau il ne faut donc pas utiliser l'opération `count()` mais la propriété `length`.

o ➢ **`db.etudiants.distinct("classe.id")`**
retourne l'identifiant des classes sans doublon.

p ➢ **`db.etudiants.countDocuments()`**
retourne le nombre d'étudiants de la collection (équivalent à **`db.etudiants.find().count()`**).

q ➢ **`db.etudiants.countDocuments({"dateNaissance.annee": 2005})`**
retourne le nombre d'étudiants qui sont nés en 2005.

1.4 Les index

Comme dans les bases de données relationnelles, les index permettent d'accélérer significativement les requêtes de recherche ; mais d'un autre côté, ils vont ralentir légèrement les requêtes de mises à jour. Il est possible de consulter la liste de tous les index d'une collection avec la commande suivante : `db.etudiants.getIndexes()`

Dans MongoDB, on peut créer des index sur des attributs d'une collection. Mais lorsqu'on a besoin de rechercher un mot à l'intérieur d'un long texte, on utilise généralement un index textuel déclaré sur la collection et qui fonctionne comme un moteur de recherche.

1.4.1 Les index portant sur un ou plusieurs attributs

Un index peut porter sur un attribut se trouvant dans les documents d'une collection. On peut aussi créer un index composé sur plusieurs attributs, mais l'ordre dans lequel on place ces attributs a son importance. Lors de la création d'un index il est possible d'indiquer si l'index doit trier les données par ordre croissant (`1`) ou par ordre décroissant (`-1`) mais cela n'aura pas d'impact sur l'efficacité des requêtes.
Le code ci-dessous permet de créer cinq index. Les quatre premiers portent sur un attribut de la collection. Le dernier est un index composé (il porte sur deux attributs).

```
db.etudiants.createIndex({"nom": 1})
db.etudiants.createIndex({"prenom": 1})
db.etudiants.createIndex({"resultats.note": 1})
db.etudiants.createIndex({"dateNaissance.annee": 1})
db.etudiants.createIndex({"nom": 1, "prenom": 1})
```

Lorsqu'on exécute une requête, on peut voir si un index est utilisé ou non en regardant le plan d'exécution de la requête grâce à l'opération `explain()`.

r ➢
```
db.etudiants.find(
    {"nom": "Bricot", "prenom": "Judas"}
).explain("executionStats")
```

Si un attribut possède un index, on peut alors utiliser les opérations `min()` et `max()` sur le curseur retourné par un `find()`.

s ➢ `db.etudiants.find().min({"dateNaissance.annee": 2006}).hint({"dateNaissance.annee": 1})`
retourne les étudiants qui sont nés en 2006 ou après (leur date de naissance est supérieur ou égale à 2006 ; elle vaut 2006 au minimum).

t ➢
```
db.etudiants.find(
    {},
    {"resultats.note": 1}
).max({"resultats.note": 10}).hint({"resultats.note": 1})
```
retourne les étudiants qui ont une note inférieure ou égale à 10 (qui est à 10 au maximum).

1.4.2 Les index textuels

Lorsqu'on souhaite rechercher un ou plusieurs mots à l'intérieur d'un long texte, afin d'améliorer les performances, plutôt que d'utiliser une expression régulière, on peut créer un index textuel sur la collection. Il est à noter que l'index est placé sur la collection et non pas sur un attribut particulier de la collection. Les recherches se feront ensuite sur la totalité des documents de la collection et non pas sur un attribut particulier.
Par exemple, l'instruction suivante va créer un index textuel sur la collection *etudiants* :

```
db.etudiants.createIndex({"$**" : "text"})
```

Ensuite, grâce à cet index, il sera possible de réaliser les requêtes suivantes :

u ➢
```
db.etudiants.find(
    {$text : {$search : "jeune sauvageon"}}
)
```
retourne les étudiants qui ont le mot 'jeune' ou 'sauvageon' dans leur document.

v ➢
```
db.etudiants.find(
    {$text : {$search : "\"jeune sauvageon\""}}
)
```
retourne les étudiants qui ont la chaîne 'jeune sauvageon' dans leur document.

w ➢
```
db.etudiants.find(
    {$text : {$search : "jeune -sauvageon"}}
)
```
retourne les étudiants qui ont le mot 'jeune' mais pas le mot 'sauvageon' dans leur document.

x ➢
```
db.etudiants.find(
    {$text: {$search: "sauvageon"}},
    {"name": true, "score": {$meta: "textScore"}}
).sort({score: {$meta: "textScore"}})
```
retourne le score du mot 'sauvageon' dans chacun des documents de la collection.

2 Concepts avancés

2.1 Le pipeline d'agrégation

Lorsqu'on souhaite faire des requêtes plus complexes, il est possible d'utiliser le pipeline d'agrégation `aggregate()`. Cette commande permet de choisir quels opérateurs (`$projet`, `$match`, `$unwind`, ...) on souhaite appliquer successivement à une collection afin de la modifier étapes par étapes. La sortie de chaque étape du pipeline sera l'entrée de l'opération suivante. Chaque étape du pipeline va permettre de rajouter, modifier ou supprimer des documents à la collection ; si bien que la collection générée à la sortie du pipeline n'aura pas forcément le même nombre de documents que la collection qui était à l'entrée du pipeline.

Dans ce cours nous ne verrons pas tous les opérateurs qu'il est possible d'utiliser dans un pipeline d'agrégation. Nous nous contenterons de voir que les principaux. A savoir :

- **$project :** permet de faire une projection ou de créer un nouvel attribut.
- **$match :** permet de faire une sélection.
- **$count :** permet de compter un nombre de lignes.
- **$sortByCount :** permet de compter le nombre de lignes dans des groupes et ensuite de classer ces groupes par rapport aux résultats obtenus.
- **$group :** permet d'utiliser les fonctions `$sum`, `$avg`, `$max`, `$min`, `$count`, `$addToSet`, `$push` avec ou sans groupement.
- **$unwind :** aplatit les données d'un tableau. Pour un document contenant un tableau de N valeurs, cet opérateur génère N documents.
- **$sort :** trie les données.
- **$skip :** pagine le résultat (comme avec le `find()`).
- **$limit :** pagine le résultat (comme avec le `find()`).
- **$lookup :** permet de réaliser une jointure externe entre deux collections.

Il est à noter que depuis la version 4.2 de Mongo, ces opérateurs peuvent aussi être utilisés avec les commandes `findAndModify` et `update`.

Le pipeline d'agrégation est généralement utilisé pour faire des requêtes complexes. Mais on peut aussi l'utiliser pour faire des requêtes simples qu'il serait également possible de faire avec l'opération `find()`.

2.1.1 $project

L'opérateur `$project` ne modifie pas le nombre de documents qu'il y a en entrée. Par contre il permet de remodeler chaque document de la collection en faisant une projection sur certains attributs (avec des `1` ou des `0`) ou en créant des nouveaux attributs.
Il est important de bien comprendre que l'opérateur du pipeline d'agrégation qui suivra le `$project` ne pourra manipuler que les attributs qui ont été produits par le `$project` (il ne pourra pas manipuler des attributs de la collection qui ont été supprimés par le `$project`)

a ➢
```
db.etudiants.aggregate([
   {$project: {"nom": 1, "prenom": 1}}
])
```
retourne l'identifiant, le nom et le prénom de tous les étudiants (les autres attributs sont éliminés).

b ➢
```
db.etudiants.aggregate([
   {$project: {"nomDeFamille": "$nom", "nbNotes": {$size: "$resultats"}, "_id": 0}}
])
```
retourne le nom de famille et le nombre de résultats de tous les étudiants (attention, il faut que tous les étudiants aient un résultat, sinon cette requête déclenchera une erreur). `nomDeFamille` n'existe pas dans la collection *etudiants*, il est créé par la requête.

c ➢
```
db.etudiants.aggregate([
   {$project: {"type": {
                    $cond: {if: {$gte:["$dateNaissance.annee", 2005]},
                            then: "Sauvageon",
                            else: "Perimé"}
                           },
               "nom": 1,
               "prenom": "$prenom"}}
])
```
retourne l'identifiant, le nom, le prénom et le type ('Sauvageon' ou 'Périmé') de tous les étudiants. Il est à noter que `type` n'existe pas dans la collection *etudiants,* il est créé dans la requête.

2.1.2 $match

L'opérateur `$match` permet de faire une sélection en filtrant les documents d'une collection. En pratique, cet opérateur est souvent le premier du pipeline d'agrégation mais cela n'est pas toujours le cas.

d ➢
```
db.etudiants.aggregate([
    {$match: {"classe.nom": "LP ACPI"}}
])
```
retourne tous les étudiants de la classe LP ACPI.

e ➢
```
db.etudiants.aggregate([
    {$match: {"resultats.nomMatiere": "BD"}},
    {$project: {"nom": 1, "prenom": 1}}
])
```
retourne l'identifiant, le nom et le prénom de tous les étudiants qui ont un résultat en BD.

f ➢
```
db.etudiants.aggregate([
    {$project: {"type": {
                    $cond: {if: {$eq:["$nom", "Palleja"]},
                            then: "Ninja",
                            else: "Apprenti Ninja"}
                           }}},
    {$match: {"type": "Ninja"}}
])
```
retourne l'identifiant et le type de tous les étudiants 'Ninja'.

2.1.3 $count

L'opérateur `$count` permet de compter le nombre de documents qu'il y a dans la collection. En sortie de cet opérateur il n'y aura qu'un seul attribut dont le nom doit être indiqué dans l'opérateur.

g ➢
```
db.etudiants.aggregate([
    {$count: "nbEtudiants"}
])
```
retourne le nombre d'étudiants.

h ➢
```
db.etudiants.aggregate([
    {$match: {"classe.nom": "LP APIDAE"}},
    {$count: "nbEtudiantsACPI"}
])
```
retourne le nombre d'étudiants de la LP APIDAE.

2.1.4 $sortByCount

L'opérateur `$sortByCount` réalise un groupement par rapport à l'attribut qui est indiqué. Puis il compte le nombre de documents qui correspondent à chacun des groupes. En sortie on aura les noms des groupes avec leur nombre d'occurrences classés par ordre décroissant du nombre d'occurrences. En SQL cela correspondrait à un `GROUP BY + COUNT + ORDER BY DESC`.

i ➢
```
db.etudiants.aggregate([
    {$sortByCount: "$classe.id"}
])
```
retourne l'identifiant et le nombre d'étudiants de toutes les classes triées de celle qui a le plus d'étudiants à celle qui en a le moins.

2.1.5 $group

L'opérateur `$group` permet d'utiliser une fonction d'ensemble avec ou sans groupement. Il est possible d'utiliser les fonctions `$sum`, `$avg`, `$max`, `$min` et plus récemment `$count` (avant il fallait faire un `$sum: 1` pour simuler un `$count`).
Dans l'opérateur `$group`, lorsque le `"_id"` vaut `null`, la fonction est appliquée sur la totalité de la collection sans groupement. Par contre, lorsque `"_id"` est renseigné, un groupement est effectué sur la valeur de cet `"_id"` et la fonction est appliquée sur chacun des groupes (comme un `GROUP BY` en SQL).

j ➢
```
db.etudiants.aggregate([
    {$group: {"_id": null, "tailleMoyenne": {$avg: "$taille"}}},
    {$project: {"_id": 0}}
])
```
retourne la taille moyenne des étudiants.

k ➢
```
db.etudiants.aggregate([
    {$group: {"_id": "$classe.id", "taillePlusGrand": {$max: "$taille"}}}
])
```
retourne pour chaque classe, l'identifiant de la classe et la taille maximale des étudiants de la classe.

l ➢
```
db.etudiants.aggregate([
   {$group: {"_id": null, "nbEtudiants": {$count: {}}}},
   {$project: {"_id": 0}}
])
```
retourne le nombre d'étudiants.
On aurait aussi pu utiliser l'opérateur `$count` du pipeline d'agrégation (§ 2.1.3).
A l'époque où la fonction `$count` n'existait pas, il fallait simuler le count avec un `$sum` comme dans l'exemple qui suit :
```
db.etudiants.aggregate([{$group: {"_id": null, "nbEtudiants": {$sum: 1}}}, $project: {"_id": 0}}])
```

m ➢
```
db.etudiants.aggregate([
   {$group: {"_id": "$classe.id", "nbEtudiants": {$count: {}}}}
])
```
retourne pour chaque classe, l'identifiant de la classe et le nombre d'étudiants de la classe.
On aurait aussi pu utiliser l'opérateur `$sortByCount` du pipeline d'agrégation (§ 2.1.4).

n ➢
```
db.etudiants.aggregate([
   {$match: {"nom": {$in: ["Palleja", "Gasquet"]}}},
   {$group: {"_id": null, "nbNinjas": {$count: {} }}},
   {$project: {"_id": 0}}
])
```
retourne le nombre d'étudiants Ninjas

o ➢
```
db.etudiants.aggregate([
   {$match: {"nom": {$in: ["Palleja", "Gasquet"]}}},
   {$group: {"_id": "$classe.id", "nbNinjas": {$count: {} }}},
   {$match: {"nbNinjas": {$gte: 2}}},
   {$project: {"_id": 1}}
])
```
retourne l'identifiant des classes qui ont deux étudiants Ninjas ou plus.

2.1.6 $unwind

L'opérateur `$unwind` est le seul opérateur qui va retourner plus de documents à la sortie qu'il n'en a pris à l'entrée. Il permet d'aplatir le tableau d'un document. Pour un document contenant un tableau de N valeurs, cet opérateur génère N documents. Si un document n'a pas le tableau ou a un tableau vide, il sera évincé du résultat à moins de le spécifier explicitement avec l'option : `$unwind: {path: "$tableauCible", preserveNullAndEmptyArrays: true}`,

p ➢
```
db.etudiants.aggregate([
   {$unwind: "$resultats"},
   {$group: {_id: null, "min": {$min: "$resultats.note"},
                        "max": {$max: "$resultats.note"}}},
   {$project: {"_id": 0}}
])
```
retourne la note minimale et la note maximale.

q ➢
```
db.etudiants.aggregate([
   {$match: {"classe.nom": "LP PSGI"}},
   {$unwind: "$resultats"},
   {$group: {"_id": null, "moyenne": {$avg: "$resultats.note"}}},
   {$project: {"_id": 0}}
])
```
retourne la moyenne des notes attribuées aux étudiants de la classe LP PSGI.

r ➢
```
db.etudiants.aggregate([
    {$match: {"classe.nom": "LP APIDAE"}},
    {$unwind: {path: "$resultats", preserveNullAndEmptyArrays: true}},
    {$group: {"_id": "$_id", "moyenne": {$avg: "$resultats.note"}}}
 ])
```
retourne l'identifiant et la moyenne de tous les étudiants de la LP APIDAE (même ceux qui n'ont pas de note).

s ➢
```
db.etudiants.aggregate([
   {$match: {"classe.nom": "LP APIDAE"}},
   {$unwind: "$resultats"},
   {$group: {"_id": "$_id", "moyenneEtud": {$avg: "$resultats.note"}}},
   {$group: {"_id": null, "moyenneClasse": {$avg: "$moyenneEtud"}}},
   {$project: {"_id": 0}}
])
```
retourne la moyenne de la classe LP APIDAE.

t ➢
```
db.etudiants.aggregate([
   {$match: {"classe.nom": "LP PSGI"}},
   {$unwind: "$resultats"},
   {$group: {"_id": "$_id", "moyenne": {$avg: "$resultats.note"}}},
   {$match: {"moyenne": {$gte: 10}}}
])
```
retourne l'identifiant et la moyenne des étudiants de la LP PSGI qui ont la moyenne générale.

2.1.7 $sort

L'opérateur `$sort` trie les documents d'une collection en fonction d'un ou plusieurs attributs. Comme pour le `find()`, le `1` permet de réaliser un tri croissant et le `-1` un tri décroissant. Il est fortement conseillé de créer un index pour les attributs sur lesquels on réalise des tris.

U ➢
```
db.etudiants.aggregate([
   {$match: {"classe.nom": "LP ACPI"}},
   {$sort: {"dateNaissance.annee": 1, "dateNaissance.mois": 1,
                                      "dateNaissance.jour": 1}},
   {$project: {"nom": 1}}
])
```
retourne l'identifiant et le nom des étudiants de la LP APIDAE classés du plus vieux au plus jeune.

V ➢
```
db.etudiants.aggregate([
   {$unwind: "$resultats"},
   {$group: {"_id": "$_id", "moyenne": {$avg: "$resultats.note"}}},
   {$sort: {"moyenne": -1}}
])
```
retourne l'identifiant et la moyenne des étudiants classés par ordre de mérite.

2.1.8 $skip et $limit

Les opérateurs `$skip` et `$limit` permettent de paginer un résultat. Ils fonctionnent comme avec l'opération `find()`.

W ➢
```
db.etudiants.aggregate([
   {$sort: {"_id": 1}},
   {$skip: 1},
   {$limit: 2}
])
```
retourne le 2ème et 3ème étudiant (par ordre de leur identifiant).

X ➢
```
db.etudiants.aggregate([
   {$unwind: "$resultats"},
   {$group: {"_id": "$_id", "moyenne": {$avg: "$resultats.note"}}},
   {$sort: {"moyenne": -1}},
   {$limit: 1},
])
```
essaye maladroitement de retourner l'identifiant et la moyenne du major de promo. Mais cette requête n'est pas digne d'un développeur Ninja. En effet s'il y a deux premiers ex-aequo, cette requête n'en retournera qu'un seul des deux.

Pour remédier à cela on peut utiliser l'opération `$push` à l'intérieur de l'opérateur `$group`. Cette opération permet de rajouter un élément dans un tableau. Il existe également l'opération `$addToSet` qui s'assure que les éléments du tableau sont uniques mais elle n'est pas utile ici.
On regroupe donc dans le même groupe tous les étudiants qui ont la même moyenne, et pour chacun de ces groupes, on crée un tableau contenant les identifiants de tous les étudiants qui ont cette moyenne. On garde ensuite uniquement le groupe de la moyenne la plus élevée. Puis on aplatit le tableau de ce groupe pour récupérer les identifiants de tous les étudiants qui ont la meilleure moyenne.

```
db.etudiants.aggregate([
   {$unwind: "$resultats"},
   {$group: {"_id": "$_id", "moyenne": {$avg: "$resultats.note"}}},
   {$group: {"_id": "$moyenne", "eleves": {$push: "$_id"}}},
   {$project: {"moyenne": "$_id", "eleves": 1, "_id":0}},
   {$sort: {"moyenne": -1}},
   {$limit: 1},
   {$unwind: "$eleves"}
])
```

Une autre solution consisterait à faire deux requêtes. Une première qui calcule la moyenne maximale des étudiants et on stocke le résultat de cette requête dans une variable `moyenneMax`. Et ensuite, une requête qui retourne tous les étudiants qui ont une moyenne égale à `moyenneMax`.

```
var moyenneMax = db.etudiants.aggregate([
   {$unwind: "$resultats"},
   {$group: {"_id": "$_id", "moyenne": {$avg: "$resultats.note"}}},
   {$sort: {"moyenne": -1}},
   {$limit: 1},
   {$project: {"moyenne": 1, "_id":0}}
]).toArray()[0].moyenne;

db.etudiants.aggregate([
   {$unwind: "$resultats"},
   {$group: {"_id": "$_id", "moyenne": {$avg: "$resultats.note"}}},
   {$match: {"moyenne": moyenneMax}}
])
```

2.1.9 $lookup

L'opérateur `$lookup` permet de réaliser une jointure externe entre la collection en entrée et une collection spécifiée dans l'attribut `from`. La jointure est réalisée entre les attributs `localField` qui est la clé de la collection manipulée par le pipeline d'agrégation et `foreignField` qui est la clé de la collection spécifiée dans le `from`. Cet opérateur est l'équivalent du `LEFT OUTER JOIN` de SQL où la collection manipulée par le pipeline serait la collection de gauche et la collection dans l'attribut `from` serait celle de droite.

Par exemple, on rajoute une collection *absences* qui permet de stocker les absences de nos étudiants. Pour une absence, on veut la date, le motif et l'identifiant de l'étudiant qui est concerné par cette absence. Cet identifiant (idEtudiant) est une sorte de clé étrangère qui fait référence à un _id de la collection *etudiants*. Et on rajoute ensuite quelques absences qui ont eu lieu aujourd'hui.

```
var aujourdhui = new Date("2023-12-18");
db.absences.insertMany([
   {"_id" : "A1", "date": aujourdhui, "idEtudiant" : "E10", "motif": "Flemme"},
   {"_id" : "A2", "date": aujourdhui, "idEtudiant" : "E20", "motif": "Panne Oreiller"}
])
```

Il est maintenant possible de réaliser des jointures entre *etudiants* et *absences*. Comme toutes les absences concernent un étudiant mais que des étudiants peuvent ne pas avoir d'absence, une jointure externe permettra de garder les étudiants qui n'ont pas d'absence.

y

- ```
  var hier = new Date("2023-12-18");
  db.absences.aggregate([
     {$match: {"date": hier}},
     {$lookup: {
         from: "etudiants",
         localField: "idEtudiant",
         foreignField: "_id",
         as: "etudiant"
         }},
       {$project: {"_id": 0, "etudiant.nom": 1, "etudiant.prenom": 1, "motif": 1}}
  ])
  ```

  pour toutes les absences du 18 décembre, retourne le nom et le prénom de l'étudiant absent ainsi que le motif de l'absence.

z

- ```
  db.etudiants.aggregate([
     {$match: {"classe.nom": "LP APIDAE"}},
     {$lookup: {
         from: "absences",
         localField: "_id",
         foreignField: "idEtudiant",
         as: "absence"
         }},
     {$project: {"nb": {$size: "$absence"}}},
     {$sort : {"nb": 1}}
  ])
  ```

  pour chaque étudiant de la LP APIDAE, retourne l'identifiant de l'étudiant ainsi que le nombre d'absences. Les étudiants qui n'ont pas d'absence vont apparaitre puisqu'une jointure externe gauche est réalisée (la collection de gauche est la collection *étudiants*).

2.2 Schéma de validation d'une collection

Au moment de la création d'une collection (opération `createCollection`) il est possible d'indiquer un schéma de validation (`validator`) qui permettra de vérifier que les documents qui vont être insérés dans la collection respecteront un certain format.

Grâce à ce schéma il est possible d'indiquer les attributs obligatoires. Il est également possible d'indiquer pour tous les attributs (obligatoires ou non) le type, la valeur maximale ou minimale, la structure (pour les objets), le nombre d'occurrence (pour les tableaux), etc.

Toutefois, actuellement, dans le cas où on utilise des références, il n'est pas possible de gérer la contrainte d'intégrité référentielle.

Par exemple, pour la collection *etudiants*, on pourrait imposer le schéma suivant où les étudiants ont obligatoirement un nom, un prénom et une date de naissance :

```
db.createCollection("etudiants", {
   validator: {
      $jsonSchema: {
         bsonType: "object",
         required: [ "_id", "nom", "prenom", "dateNaissance" ],
         properties: {
            _id: {
               bsonType: "string",
               description: "'_id' est la clé et est obligatoire"
            },
```

```
            nom: {
               bsonType: "string",
               description: "'nom' doit être un string et est obligatoire"
            },
            prenom: {
               bsonType: "string",
               description: "'prenom' doit être un string et est obligatoire"
            },
            dateNaissance: {
               bsonType: "object",
               required: [ "jour", "mois", "annee"],
               properties: {
                  jour: {
                     bsonType: "int",
                     minimum: 1,
                     maximum: 31,
                      description: "'jour' est entier compris entre 1 et 31 et est obligatoire"
                  },
                  mois: {
                     bsonType: "int",
                     minimum: 1,
                     maximum: 12,
                      description: "'mois' est entier compris entre 1 et 12 et est obligatoire"
                  },
                  annee: {
                     bsonType: "int",
                      description: "'annee' est entier et est obligatoire"
                  }
               }
            },
            email: {
               bsonType: "string",
               description: "'email' doit être un string"
            },
            classe: {
               bsonType: "object",
                  properties: {
                     id: {
                     bsonType: "string",
                      description: "'id' est un string"
                     },
                     nom: {
                        bsonType: "string",
                        description: "'nom' est un string"
                     },
                     niveau: {
                        bsonType: "string",
                        description: "'niveau' est un string"
                     }
                  }
            },
            resultats: {
                bsonType: "array",
                items: {
                   bsonType:"object",
                   maxItems: 5,
                    description: "5 résultats max",
                   properties:{
                      id: {
                         bsonType: "string",
                         description: "'id' est un string"
                      },
                      nomMatiere: {
                         bsonType: "string",
                         description: "'nomMatiere' est un string"
                      },
                      note: {
                         bsonType: "int",
                         minimum: 0,
                         maximum: 20,
                         description: "'note' est un entier"
                      }
                   }
                }
            }
         }
      }
   }
})
```

3 Manipulation de MongoDB avec un langage de programmation

Pour se connecter à une base de données MongoDB à partir d'une application Java, il est possible d'utiliser le driver développé par MongoDB et qui est codé en Java (il en existe d'autres comme par exemple celui de Spring mais ils sont moins populaires et seraient également moins performants). Avec la version 6.0 du MongoDB utilisé à l'IUT, il faut utiliser au minimum la version 4.7 de ce driver.
Pour intégrer ce driver à un projet Maven, il suffit d'écrire la dépendance suivante dans pom.xml.

```
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-driver-sync</artifactId>
    <version>4.8.2</version>
</dependency>
```

3.1 Connexion et déconnexion à la base de données

Lorsque MongoDB est installé en local, la connexion doit être faite par défaut sur le port `27017` de `localhost`. Pour se connecter au MongoDB de l'IUT, il faut se connecter au port `27018` de `162.38.222.152`. A l'IUT cette connexion doit être réalisée sur la base de données `iut`, et une fois la connexion établie, vous devrez utiliser la base de données qui a le même nom que votre login.
La connexion est réalisée en utilisant la méthode statique `create()` de la classe `MongoClients`. En paramètre, cette méthode attend une chaîne de caractères qui doit avoir la forme suivante :

```
mongodb://login:motDePasse@url:port/bdDeConnexion
```

Une fois la connexion établie, on peut sélectionner la base de données de l'utilisateur avec la méthode `getDatabase()`.

```
String login = "zetofraism";
String mdp = "jaimeLePanda";
String url = "162.38.222.152";
String port = "27018";
String bdDeConnexion = "iut";
String bdUtilisateur = "zetofraism";
String uriConnexion = "mongodb://"+ login + ":" + mdp + "@" + url + ":" + port + "/" + bdDeConnexion;
MongoClient mongoClient = MongoClients.create(uriConnexion);
MongoDateBase database = mongoClient.getDatabase(bdUtilisateur);
```

La déconnexion à la base de données est réalisée en utilisant la méthode close.

```
mongoClient.close() ;
```

3.2 Requêtes CRUD

3.2.1 Requêtes de mises à jour

Pour mettre à jour une collection ou modifier un document de la collection, il est possible d'utiliser les mêmes opérations que celles vues au paragraphe § 1.3.3 (`insertOne()`, `insertMany()`, `updateOne()`, `updateMany()`, `replaceOne()`, `deleteOne()`, `deleteMany()`).

```
private void ajouterUnEtudiant(String idEtudiant, String nom, String prenom,
                        int jour, int mois, int annee,
                        String idClasse, String nomClasse, String niveauClasse,
                        String telephone, String email) {
   Document dateNaissance = new Document()
            .append("jour", jour)
            .append("mois", mois)
            .append("annee", annee);
   Document classe = new Document()
            .append("id", idClasse)
            .append("nom", nomClasse)
            .append("niveau", niveauClasse);
   Document etudiant = new Document()
            .append("_id", idEtudiant)
            .append("nom", nom)
            .append("prenom", prenom)
            .append("dateNaissance", dateNaissance)
            .append("classe", classe);
   if (telephone != null) etudiant.append("telephone", telephone);
   if (email != null)  etudiant.append("email", email);
   MongoCollection<Document> collectionEtudiants = database.getCollection("etudiants");
   collectionEtudiants.insertOne(etudiant);
}
```

```
private void ajouterUneNote(String idEtudiant,
                            String idMatiere, String nomMatiere, int note) {
   Document resultat = new Document()
           .append("id", idMatiere)
           .append("nomMatiere", nomMatiere)
           .append("note", note);
   MongoCollection<Document> collectionEtudiants = database.getCollection("etudiants");
   collectionEtudiants.updateOne(new Document("_id", idEtudiant),
                                 new Document("$addToSet", new Document("resultats", resultat)));
}
```

```
private void supprimerUneNote(String idEtudiant, String idMatiere) {
   MongoCollection<Document> collectionEtudiants = database.getCollection("etudiants");
   collectionEtudiants.updateOne(new Document("_id", idEtudiant),
                                 new Document("$pull", new Document("resultats",
                                                  new Document("id", idMatiere))));
}
```

```
private void modifierNomEtudiant(String idEtudiant, String nom) {
   MongoCollection<Document> collectionEtudiants = database.getCollection("etudiants");
   collectionEtudiants.updateOne(new Document("_id", idEtudiant),
                                 new Document("$set", new Document("nom", nom)));
}
```

```
private void modifierDateNaissanceEtudiant(String idEtudiant,
                                           int jour, int mois, int annee) {
   Document dateNais = new Document()
           .append("jour", jour)
           .append("mois", mois)
           .append("annee", annee);
   MongoCollection<Document> collectionEtudiants = database.getCollection("etudiants");
   collectionEtudiants.updateOne(new Document("_id", idEtudiant),
                                 new Document("$set", new Document("dateNaissance", dateNais)));
}
```

```
private void supprimerEtudiant(String idEtudiant) {
   MongoCollection<Document> collectionEtudiants = database.getCollection("etudiants");
   collectionEtudiants.deleteOne(new Document("_id", idEtudiant));
}
```

```
private void viderCollection() {
   MongoCollection<Document> collectionEtudiants = database.getCollection("etudiants");
   collectionEtudiants.deleteMany(new Document());
}
```

3.2.2 Requêtes d'extraction de données

Pour rechercher des données dans un document, on peut utiliser la méthode `find()` (§ 1.3.4) sans paramètre pour afficher tous les documents de la collection ou bien avec un document en paramètre afin de filtrer le résultat. On peut aussi utiliser la méthode `aggregate()` (§ 2.1) pour faire des recherches plus complexes ou réaliser des jointures.

```
private void afficherLaListeDesEtudiantsSousFormeDeDocumentsJson() {
   MongoCollection<Document> collectionEtudiants = database.getCollection("etudiants");
   FindIterable<Document> resultatRequete = collectionEtudiants.find();
   for (Document doc : resultatRequete) {
       System.out.println(doc.toJson());
   }
}
```

On peut simplifier le code précédent en utilisant un curseur (dans MongoDB, un `find()` retourne un curseur) ou un itérateur.

```
   MongoCursor<Document> cursor = collectionEtudiants.find().iterator();
   while (cursor.hasNext()) {
      var doc = cursor.next();
      System.out.println(doc.toJson());
   }
```

Ou bien, de façon plus synthétique

```
   collectionEtudiants.find().forEach(doc-> System.out.println(doc.toJson()));
```

```
private void afficherIdNomPrenomDesEtudiantsSousFormeDAttributs() {
   MongoCollection<Document> collectionEtudiants = database.getCollection("etudiants");
   for (Document doc : collectionEtudiants.find()) {
      var etud = new ArrayList<>(doc.values());
      System.out.println(etud.get(0) + " " + etud.get(1) + " " + etud.get(2));
}
```

```
private void afficherLaListeDesEtudiantsParNom(String leNom) {
   MongoCollection<Document> collectionEtudiants = database.getCollection("etudiants");
   collectionEtudiants.find(new Document("nom", leNom))
                                          .forEach(doc -> System.out.println(doc. toJson()));
}
```

```
private void afficherLaListeDesEtudiantsQuiOntEuUn20() {
   MongoCollection<Document> collectionEtudiants = database.getCollection("etudiants");
   AggregateIterable<Document> resultatRequete = collectionEtudiants.aggregate(Arrays.asList(
            new Document("$unwind", "$resultats"),
            new Document("$match", new Document("resultats.note", 20)),
            new Document("$project", new Document("_id", 0)
                                                 .append("nom", 1)
                                                 .append("prenom", 1))
    ));
    resultatRequete.forEach(doc -> System.out.println(doc.toJson()));
}
```

Il est possible de simplifier le code en rajoutant l'import statique suivant :

```
import static com.mongodb.client.model.Aggregates.*;
```

On peut alors écrire le code de l'agrégation comme cela :

```
   AggregateIterable<Document> resultatRequete = collectionEtudiants.aggregate(Arrays.asList(
           unwind("$resultats"),
           match(new Document("resultats.note", 20)),
           project(new Document("_id", 0)
                           .append("nom", 1)
                           .append("prenom", 1))
   ));
```

```
private void afficherLeNombreDeNotesDansChaqueMatiere() {
   MongoCollection<Document> collectionEtudiants = database.getCollection("etudiants");
   AggregateIterable<Document> resultatRequete = collectionEtudiants.aggregate(Arrays.asList(
           new Document("$unwind", "$resultats"),
           new Document("$group", new Document("_id", "$resultats.id")
                                              .append("nb", new Document("$sum", 1))),
           new Document("$sort", new Document("nb", -1))
   ));
   resultatRequete.forEach(doc -> System.out.println(doc.toJson()));
}
```

```
private void rechercherGenerique(String attribut, String valeur) {
   MongoCollection<Document> collectionEtudiants = database.getCollection("etudiants");
   for (Document doc : collectionEtudiants.find(Filters.eq(attribut, valeur)))
      System.out.println(doc.toJson());
   }
}
```