# 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, ... . MongoDB est d'ailleurs le SGBD NoSQL le plus populaire. D'abord parce que la représentation des données en documents JSON (JavaScript Object Notation) dont la structure n'a pas besoin d'être prédéfinis (sans schéma ou schema-less) lui donne une grande flexibilité qui peut être intéressante pour stocker des données hétérogènes qui ne sont pas forcément structurées ; et grâce à cette flexibilité il peut être utilisé dans des domaines métier très différents, et facilite aussi le développement itératif-incrémental. Ensuite parce que MogoDB 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 utilisés. 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 MogoDB 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 un chaine 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 un 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.

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

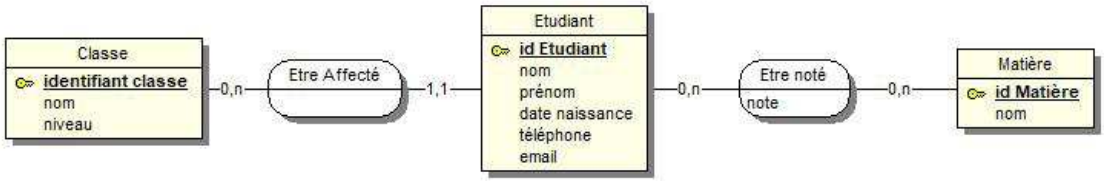
```
[{
    "_id": "A345",
    "nom": "Jackson",
    "prenom": "Samuel L"
    "age": 67

},{
    "_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 de 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}]}
)
```

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` (pour not or).

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 autres note 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 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 $nin` (pour NOT IN) ainsi que l'opérateur `$exist`. Mais contrairement au SQL il n'est pas possible de les utiliser avec une requête imbriquée.

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

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 ou 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 chaine de caractères. En pratique elles sont très utiles pour savoir si un attribut commence, se termine ou contient une chaine de caractères (comme le `LIKE` en SQL).

`/a/` permet de trouver les chaines qui contiennent 'a'. `/a$/` permet de trouver celles qui se terminent par un 'a', et `/^a/` celles qui commencent par un 'a'. `/aa*/` permet de trouver les chaines 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()`. `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 le nom 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 chaine '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.