Introduction aux BDD noSQL avec MongoDB

Master TIIL & ILIADE 1ère année

Eric Cariou

Université de Bretagne Occidentale
UFR Sciences et Techniques – Département Informatique

Eric.Cariou@univ-brest.fr

Janvier 2023

Base de données relationnelles

- SGBD (Système de Gestion de Base de Données) relationnel de type SQL
  - Données typées et fortement structurées
    - Définies par des tables et un schéma
    - Normalisées pour éviter la redondance des données
      - Utilisation de jointures entre tables pour liens entre données
  - Importance de l'intégrité des données
    - Ex. du schéma de sports précédent
      - Une discipline doit être rattachée à un sport existant (vérification de l'existence de la clé primaire)
      - Un sport ne peut pas être supprimé si des disciplines lui étant rattachées existent encore
  - SQL : Structured Query Language
    - Langage de requête très puissant

Big Data

- Problème des 3V
  - *Volume* (volume)
    - Le nombre de données progresse de manière exponentielle
    - Google (2021) : 130 000 milliards de pages Web indexées par 110 millions de Go de données
  - *Variety* (varieté)
    - Types de données très hétérogènes : textes (structurés ou non), binaires (vidéos, documents…), de taille quelconque
    - Sources : sites/services Web, réseaux sociaux, IoT, BDD diverses…
  - *Velocity* (vélocité)
    - Capacité à gérer les flux de données
      - Twitter (2022) : 850 millions de nouveaux Tweets par jour
      - Google (2021) : 20 milliards de sites analysés par jour
    - Capacité à accéder rapidement aux données
      - Google (2021) : 7 milliards de requêtes par jour
      - Youtube (2021) : 1 milliard d'heures de vidéo visionnées par jour

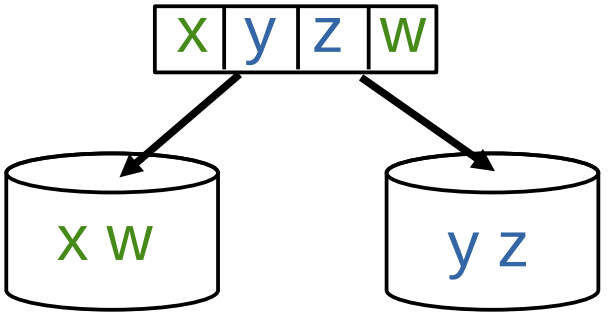
Big Data

- Pour gérer les 3V, nécessité de
  - Distribuer/partitionner les données sur de multiples serveurs
  - Paralléliser les accès aux données
  - Google : estimation de 2,5 millions de serveurs en 2016
- Problème
  - SGBD relationnels SQL mal adaptés à la distribution des données et à l'accès concurrent massifs
  - SGBD noSQL (Not Only SQL)
    - Passage du paradigme ACID à BASE
      - Relâcher des contraintes sur l'intégrité ou cohérence des données
    - Pour faciliter la distribution et l'accès à une grande masse de données

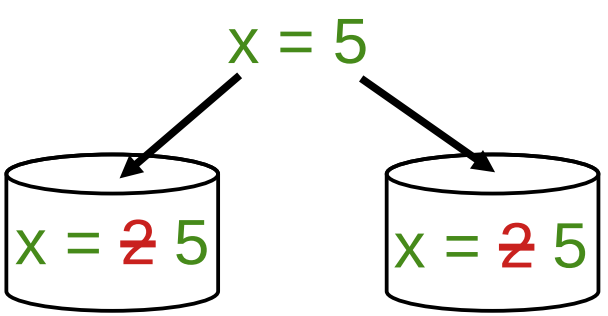
Données distribuées

- Données sur plusieurs serveurs

distribution/partitionnement des données



réplication des données



- Exemple de données partitionnées
  - Transfert d'argent d'un compte d'une banque vers une autre banque
    - Requiert un débit sur un serveur puis un crédit sur un autre serveur
    - Doit faire les deux actions ou aucune pour avoir un état cohérent
    - Débiter (banque A, #1244, 1000€)
      Créditer (banque B, #8812, 1000€)
- Réplication des données pour tolérance aux pannes
  - Quand un serveur tombe, on bascule sur l'autre
  - Nécessite de faire les mêmes modifications sur les 2 serveurs : cohérence constante du contenu des deux BDD

Transaction : propriétés ACID

- Modifications sur un ensemble de serveurs
  - En mode transaction pour un SGBD SQL
- Propriétés ACID [Härder & Reuter, 83]
  - *Atomicity* (atomicité)
    - Tout ou rien : les actions de la transaction sont entièrement réalisées ou pas du tout, pas d'intermédiaire à moitié fait
  - *Consistency* (cohérence)
    - L'exécution d'une transaction fait passer le système d'un état cohérent à un autre
  - *Isolation* (isolation)
    - Les transactions n'interfèrent pas entre elles
  - *Durability* (durabilité)
    - Les effets de la transaction sont enregistrés de manière permanente

D'ACID vers BASE

- Transaction en distribué
  - Nécessite des algorithmes assez complexes
    - Algorithme de validation atomique (2PC ou 3PC)
      - Synchronisation à distance de serveurs : quid des pannes ?
    - Utilisation de verrous sur les données de la transaction
      - Empêche une autre transaction de modifier ces données le temps de la transaction
      - Peut entraîner des interblocages
  - S'exécute en un *certain* temps : trop long ?
  - Difficulté voire impossibilité d'assurer les 4 propriétés ACID à la fois en distribué
- SGBD noSQL
  - Favoriser la vélocité d'accès aux données ou la tolérance aux pannes
  - Quitte à se retrouver momentanément dans des états incohérents
  - Propriétés BASE au lieu d'ACID

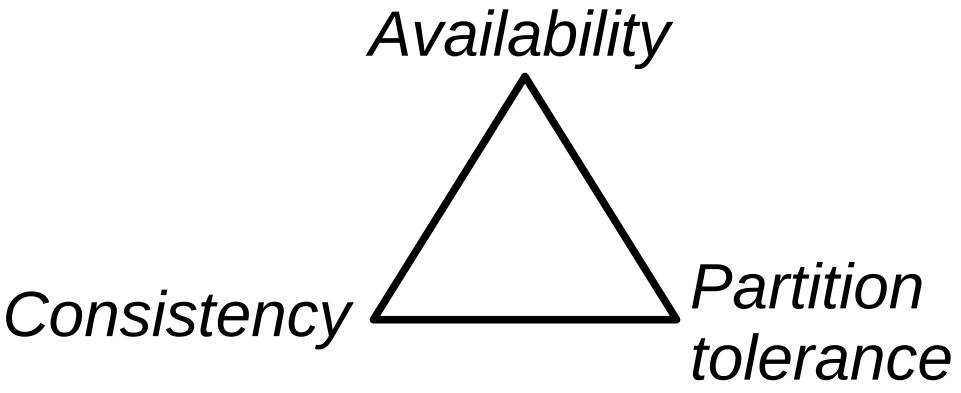
Propriétés BASE

- ***B**asically **A**vailable* (disponibilité minimale)
  - Assure un taux minimal de réponse aux requêtes
  - Indépendamment de la taille des données et du nombre de requêtes
- ***S**oft-state* (état faible)
  - Quand on utilise de la réplication : des serveurs peuvent planter ou d'autres peuvent être rajoutés dynamiquement
  - Les serveurs ne sont pas tous dans le même état en même temps
- ***E**ventually consistent* (finalement cohérent)
  - On finira tout de même par arriver dans un état cohérent où tous les réplicats de données auront la même valeur
  - Le temps d'assurer la réplication sur tous les serveurs

Théorème CAP

- Théorème de [Brewer 2000 & Gilbert, Lynch 2002] définit 3 propriétés pour un SGBD distribué
  - *Consistency* (cohérence)
    - Une donnée n'a qu'une seule valeur visible sur tous les serveurs de réplication
  - *Availability* (disponibilité)
    - La donnée est disponible tant que le système n'est pas totalement planté
  - *Partition tolerance* (tolérance aux pannes)
    - La partition des données ne doit pas empêcher l'accès aux donnés malgré des pannes de serveurs
- Théorème
  - Pour un SGBD distribué, on ne peut assurer que deux propriétés à la fois
  - Les trois propriétés ne peuvent pas être garanties en même temps

Théorème CAP



- Un SGBD ne sera que sur un coté du triangle à la fois
  - C & A : systèmes relationnels SQL classiques
    - Données toujours valides et disponibles
    - Mais difficulté à distribuer/partitionner ou répliquer
  - C & P : systèmes noSQL
    - Se focalisant sur la tolérance aux pannes quitte à rendre les données momentanément indisponibles
  - A & P : systèmes noSQL
    - Se focalisant sur l'accès aux données sans perte quitte à être momentanément incohérent

4 familles de SGBD noSQL

- Clés-valeurs
  - Fonctionne comme une *Map* : couple clé-valeur
    - La valeur a un contenu quelconque, non défini
  - Langage de requête basique
    - Rajouter un couple clé-valeur, récupérer une valeur par sa clé …
    - Pas de traitement par le SGBD du contenu de la valeur
  - Avantage
    - Très facile à partitionner/distribuer, passage à l'échelle
- Orienté document
  - Similaire à clé-valeur mais avec la valeur dont le contenu possède une structure (flexible)
    - Permet de faire des requêtes sur le contenu de la valeur
  - Valeur peut être une clé-valeur : hiérarchie de documents
    - Avec duplication possible des données à différents endroits

4 familles de SGBD noSQL

- Orienté colonnes
  - Au lieu de ranger par ligne comme en SQL, on range les données par colonne
  - Avantages
    - Facile à partitionner/distribuer
    - Accès direct aux données associées à une colonne
  - Inconvénient
    - Plus difficile de faire les liens entre les éléments des colonnes (pour retrouver l'équivalent d'une ligne)

Id	Nom	Ville	Sports
1	Roger	Brest	100m, brasse, fléchettes
2	Simone		pétanque, curling
3	Gérard	Rennes	
4	Régine	Brest	pétanque, fléchettes

Nom	Id
Roger	1
Simone	2
Gérard	3
Régine	4

Ville	Id
Brest	1, 4
Rennes	3

Sports	Id
100m	1
brasse	1
fléchettes	1, 4
pétanque	2, 4
curling	2

Orienté lignes (à la SQL)

Orienté colonnes

4 familles de SGBD noSQL

- Orientés graphes
  - Met l'accent sur les relations entre les éléments
  - Adapté pour stocker les données d'un réseau social, d'un réseau routier, géographique …
    - Calcul du plus court chemin par exemple
  - Langage de requête assez complexe
- Performances (vélocité, élasticité ...)
  - Dépendent du type de la famille
  - De la façon dont les données sont indexées et partitionnées
    - Voir documentations spécialisées à ce sujet
    - https://chewbii.com/ onglet enseignement (page de Nicolas Travers)
    - https://openclassrooms.com/fr/courses/4462426-maitrisez-les-bases-de-donnees-nosql

SQL ou noSQL ?

- Cela dépend des besoins
  - Relationnel SQL
    - Données très structurées, gestion de l'intégrité et de la cohérence des données, non duplication de données (jointures)
    - Langage de requête évolué
    - Mais plus rigide et plus difficile à distribuer/partitionner pour très grande masse de données
  - noSQL
    - Structure des données généralement plus flexible voire non définie (peut être basé sur du JSON par exemple)
    - Avantages / inconvénients dépendent de la famille choisie (cf transparents précédents)
    - Plus adaptés pour du partitionnement et de la distribution à large échelle (Big Data)

Introduction à MongoDB

MongoBD

- SGBD noSQL orienté document
  - Les données sont définies en JSON
    - Sauvegardées physiquement en BSON : Binary JSON
  - Document : ensemble de clés-valeurs (un objet JSON)
    - Une valeur peut être un document : structure hiérarchique en arbre
  - Une collection est nommée et est un tableau de documents
  - Pas de schéma définissant la structure des données
    - On peut rajouter des champs ou en enlever pour des documents dans la même collection
  - Langage de requête sur les clés et valeurs
- Performances (distribution, réplication ...)
  - Par défaut, fonctionne en « cohérence » et « tolérance aux pannes »
  - Mais peut être passé en « disponibilité » et « tolérance aux pannes »
  - *Non abordé dans ce cours*

Rappels sur JSON

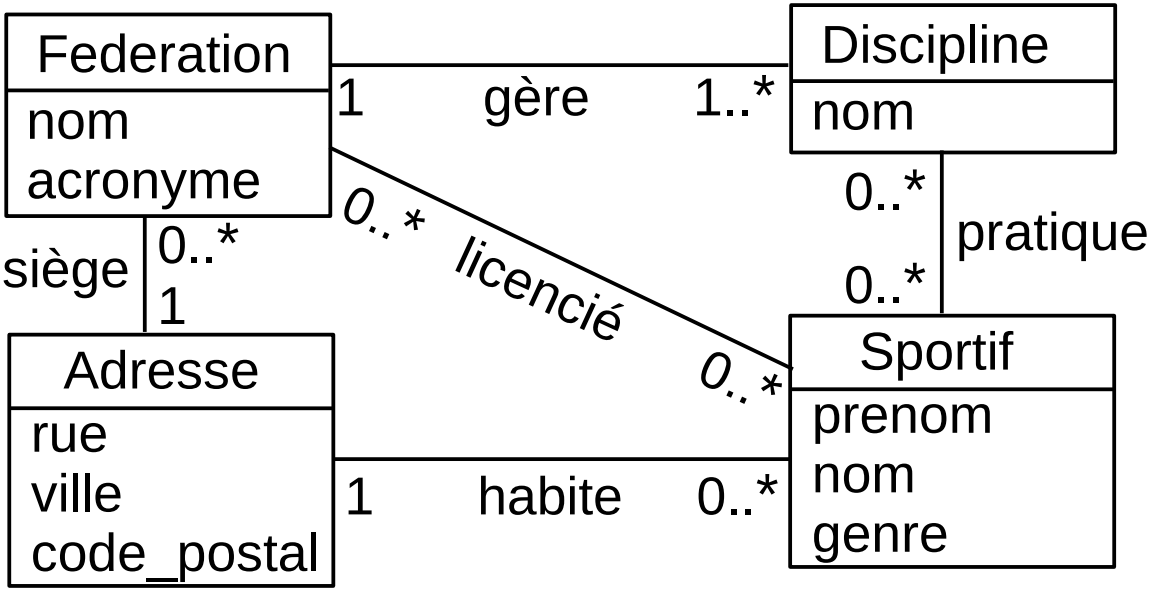
- JSON : JavaScript Object Notation
  - En pratique, aucun lien spécifique avec JavaScript
- Format textuel peu typé et moins verbeux que XML
  - 4 types de données
    - chaîne, nombre, booléen, null
  - 2 structures
    - Objet : ensemble de propriétés
      - Propriété = clé-valeur
      - Clé : chaîne
      - Valeur : un élément d'un type de base, un tableau ou un objet
      - Propriétés séparées par des virgules et encadrées par des accolades
    - Tableaux définis par des crochets et éléments séparés par des virgules
      - Élément : type de base, objet ou tableau

```
{
  "nom": "Michu",
  "prenom": "Robert",
  "age": 70,
  "marie": false,
  "adresse": {
    "rue": "12 rue de Siam",
    "ville": "Brest",
    "codePostal": 29200
  },
  "emploi": null,
  "hobbies": [
    "pétanque",
    "belote",
    "pêche"
  ]
}
```

Principes noSQL vs SQL

- Modélisation des données en SQL
  - Normalisation pour éviter les redondances de données
  - Jointures entre les tables pour lier les données
- Exemple : fédérations sportives avec licenciés
  - federation(id_fed, nom, acronyme, *id_adr*)
    discipline(id_disc, nom, *id_fed*)
    sportif(id_sp, prenom, nom, genre, *id_adr*)
    adresse(id_adr, rue, ville, code_postal)
    licencie(*id_fed*, *id_sp*)
    pratique(*id_disc*, *id_sp*)

clé_primaire
clé_étrangère



Contrainte d'intégrité : un sportif pratique une discipline s'il est licencié de la fédération gérant la discipline (à implémenter par un trigger)

Principes noSQL vs SQL

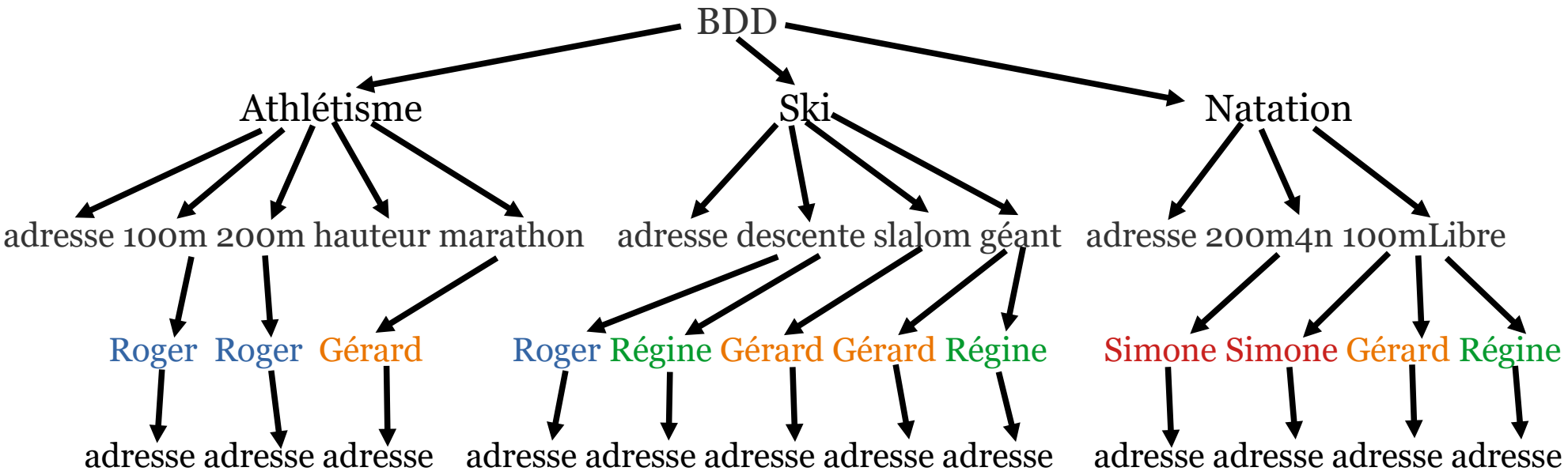
- Schéma SQL défini
  - 4 structures de données : fédération, discipline, sportif, adresse
  - Nécessite 6 tables SQL avec 5 jointures dont 2 via des tables d'association
  - Données avec beaucoup de relations entre elles
- Modélisation en noSQL
  - Éviter les jointures
    - Très coûteux en temps d'accès
    - Rend plus difficile la répartition/parallélisation
  - Autorise alors la duplication de données
  - On va commencer avec une hiérarchie unique de données
    - Une fédération contient des disciplines
    - Une discipline contient des sportifs
    - Privilégie un point d'accès aux données par les fédérations

Jeu de données

- Fédérations
  - Athlétisme : 100 mètres, 200 mètres, saut en hauteur, marathon
  - Ski : descente, slalom, géant
  - Natation : 200m 4 nages, 100m nage libre
- Sportifs
  - Roger pratique 100m, 200m, descente
  - Simone pratique 200m 4 nages, 100m nage libre
  - Gérard pratique marathon, slalom, géant, 100m nage libre
  - Régine pratique descente, géant, 100m libre
- Chaque sportif et fédération a également une adresse

Modélisation purement hiérarchique

- Avec une hiérarchie unique, arbre suivant :



- Énormément de redondances
  - Notamment les sportifs qui sont en multiples exemplaires à différents endroits de l'arbre
  - Si on change l'age de Gérard, il faut le faire 4 fois
    - La gestion de la cohérence des données est complexifiée
  - Pour trouver les disciplines de Gérard : parcourt de tout l'arbre
    - Pas très optimal pour l'efficacité

Modélisation intermédiaire

- Qu'est-ce qui peut être redondant ?
  - Une adresse
    - Très peu modifiée et quasi-spécifique à un sportif ou fédération
  - On peut la dupliquer sans surcoût de gestion
    - Remarque : en SQL, on pourrait aussi avoir les 3 champs d'adresse dans les tables « fédération » et « sportif » pour éviter là aussi une jointure peu pertinente
- Qu'est-ce qui peut être simplifié ?
  - Une discipline est juste définie par un nom
  - Pas besoin d'objet/structure dédié : une simple chaîne
    - Peut être redondant là-aussi sous les hypothèses :
      - Deux disciplines n'ont pas le même nom parmi toutes les fédérations
      - Ne change pas le nom d'une discipline (pas de pb de duplication à gérer)
    - Une fédération contient une liste de noms de discipline
    - Un sportif contient une liste de noms de discipline

Modélisation intermédiaire

- Reste le cas des sportifs
  - On peut les sortir de la collection de fédérations
  - Formeront une collection à part
    - Un sportif aura une adresse et une liste de noms de discipline
  - Une fédération fera une jointure sur la collection de sportifs
    - On pourrait faire le contraire mais c'est plus important d'accéder à la liste des sportifs d'une fédération qu'à la liste des fédérations d'un sportif
- Limite des jointures en MongoDB
  - La jointure reste une opération coûteuse en temps
  - Elle empêche le partitionnement des données
    - On ne peut pas faire une jointure entre une collection sur un serveur et une collection sur un autre serveur
    - Ici, les deux collections de sportifs et de fédérations doivent être sur le même serveur
  - La jointure n'est pas explicitement définie comme en SQL, pas de gestion de validité des identifiants : à faire par le développeur

Modélisation intermédiaire

- Dans la base MongoDB, on aura deux collections de documents (équivalents de deux tables SQL mais sans structure définie)
  - Collection de documents « federations »
    - Propriétés nom, acronyme de type chaîne
    - Propriété adresse qui est un document contenant les 3 propriétés de l'adresse
    - Propriété disciplines qui est un tableau de noms de discipline
    - Propriété sportifs qui est un tableau d'identifiants de sportifs
  - Collection de documents « sportifs »
    - Propriétés prenom, nom, genre de type chaîne, age de type entier
    - Propriété adresse qui est un document contenant les 3 propriétés de l'adresse
    - Propriété disciplines qui est un tableau de noms de discipline
- Information technique
  - Tout document inséré dans une collection possède un attribut d'identification nommé « _id » (de type quelconque) dont la valeur est unique
  - Soit explicite et défini par l'utilisateur, soit créé automatiquement par MongoBD à l'insertion du document dans une collection

Collection « federations »

```
[
{
        "nom": "Fédération Française d'Ahtlétisme",
        "acronyme": "FFA",
        "adresse": {
                "rue": "33 avenue Pierre-de-Coubertin",
                "ville": "Paris",
                "codePostal": 75013
        },
        "disciplines" : ["100m", "200m", "saut hauteur", "marathon"],
        "sportifs" : [1, 3]
},
{
        "nom": "Fédération Française de Natation",
        "acronyme": "FFN",
        "adresse": {
                "rue": "104 rue Martre",
                "ville": "Clichy",
                "codePostal": 92110
        },
        "disciplines" : ["200m 4 nages", "100m libre"],
        "sportifs" : [2, 3, 4]
},
{
        "nom": "Fédération Française de Ski",
        "acronyme": "FFS",
        "adresse": {
                "rue": "50 rue des Marquisats",
                "ville": "Annecy",
                "codePostal": 74000
        },
        "disciplines" : ["descente", "slalom", "géant"],
        "sportifs" : [1, 3, 4]
}
]
```

Collection « sportifs »


```
[
    {
        "_id": 1,
        "prenom": "Roger",
        "nom": "Blanchard",
        "age": 55,
        "genre": "homme",
        "adresse": {
            "rue": "12 rue de Siam",
            "ville": "Brest",
            "codePostal": 29200
        },
        "disciplines": [
            "100m",
            "200m",
            "descente"
        ]
    },
    {
        "_id": 2,
        "prenom": "Simone",
        "nom": "Blanchard",
        "age": 52,
        "genre": "femme",
        "adresse": {
            "rue": "12 rue de Siam",
            "ville": "Brest",
            "codePostal": 29200
        },
        "disciplines": [
            "200m 4 nages",
            "100m libre"
        ]
    },
    {
        "_id": 3,
        "prenom": "Gérard",
        "nom": "Lebreton",
        "age": 24,
        "genre": "homme",
        "adresse": {
            "rue": "20 rue Saint-Michel",
            "ville": "Rennes",
            "codePostal": 35000
        },
        "disciplines": [
            "marathon",
            "slalom",
            "géant",
            "100m libre"
        ]
    },
    {
        "_id": 4,
        "prenom": "Régine",
        "nom": "Mouvier",
        "age": 31,
        "genre": "femme",
        "adresse": {
            "rue": "153 rue Jean Jaures",
            "ville": "Brest",
            "codePostal": 29200
        },
        "disciplines": [
            "descente",
            "géant",
            "100m libre"
        ]
    }
]
```

Création de la base

- Dans ce qui suit : commandes entrées dans le Shell MongoBD
- test> use sports
  switched to db sports
  - Utilise la base « sports »
  - Si elle n'existait pas, elle est créée (même principe avec les collections)
- Insertions dans les collections
  - Insertion dans la collection « federations » du tableau JSON précédent
    - sports> db.federations.insertMany([...])

```
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("63c94228d9b92c6462014bed"),
    '1': ObjectId("63c94228d9b92c6462014bee"),
    '2': ObjectId("63c94228d9b92c6462014bef")
  }
}
```

      - Retourne un JSON précisant les 3 insertions avec la création d'un identifiant pour chaque document (ObjectId)
  - Pour insérer un seul document
    - sports> db.federations.insertOne ({ ... })

Sélection d'éléments

- find sur une collection pour récupérer des documents
  - Critères de filtre exprimés par un objet JSON
  - sports> db.sportifs.find ({})
    - Retourne tous les documents de la collection sportifs
  - sports> db.sportifs.find ({ "genre": "femme" })
    - Retourne les sportifs qui sont des femmes
  - sports> db.sportifs.find ({ "adresse.ville": "Brest" })
    - Retourne tous les sportifs habitants à Brest
  - sports> db.sportifs.find ({ "genre": "femme", "adresse.ville": "Brest" })
    - Retourne les sportives qui habitent à Brest
  - sports> db.sportifs.find ({ "marie": true })
    - La propriété « marié » n'existe pas pour un sportif mais pas d'erreur
    - Les documents dans les collections ne sont pas typés et n'ont pas forcément tous les mêmes propriétés
- findOne({ ...}) : variante qui ne retourne qu'un objet satisfaisant le filtre (le premier trouvé dans la collection)

Sélection d'éléments

- Contraintes sur les nombres
  - $gt, $gte, $lt, $lte, $eq, $ne
  - sports> db.sportifs.find ({ "age": {$gt: 50 } })
    - Retourne les sportifs dont l'age est « greater than » 50 ans
  - sports> db.sportifs.find ({ "age": {$gt: 40, $lt:50 } })
    - Retourne les sportifs dont l'age est entre 40 et 50 ans
- Valeurs dans un tableau
  - $in, $nin : valeur dans ou pas dans le tableau
  - sports> db.sportifs.find({ "disciplines": { $in: ["géant", "descente"] }})
    - Retourne les sportifs qui ont « géant » ou « descente » dans leur tableau de disciplines
- Taille d'un tableau
  - sports> db.federations.find ({ "sportifs": {$size :2} })
  - Retourne les fédérations avec 2 sportifs

Sélection d'éléments

- Position dans un tableau
  - sports> db.sportifs.find({ "disciplines.2": "descente" })
    - Les sportifs qui ont la descente à l'index 2 de disciplines
- Tous les éléments d'un tableau vérifient un ensemble de critères
  - { <field>: { $elemMatch: { <query1>, <query2>, ... } } }
- Opérateurs logiques : $and, $or, $not, $nor
  - Syntaxe : { $op : [{ cond1 }, { cond2 }, ...] }
  - sports> db.sportifs.find(
    { $or: [{"adresse.ville": "Rennes" } , { "genre": "femme" }] })
    - Retourne les sportifs qui habitent à Rennes ou qui sont des femmes
  - Le « et » logique est fait automatiquement quand on précise plusieurs filtres, ces deux commandes font la même chose :
    - sports> db.sportifs.find(
      { $and: [{"adresse.ville": "Brest" } , { "genre": "femme" }] })
    - sports> db.sportifs.find({"adresse.ville": "Brest" , "genre": "femme" })

Sélection de propriétés

- find ({ <filtre> }, { <liste_propriétés> })
  - Sur les éléments d'une collection filtrée, ne conserve que les propriétés sélectionnées avec la valeur 1
  - sports> db.sportifs.find({}, { "nom":1, "prenom":1 })
    [{ _id: 1, prenom: 'Roger', nom: 'Blanchard' },
    { _id: 2, prenom: 'Simone', nom: 'Blanchard' },
    { _id: 3, prenom: 'Gérard', nom: 'Lebreton' },
    { _id: 4, prenom: 'Régine', nom: 'Mouvier' }]
    - Par défaut, propriété _id conservée en plus des propriétés mises à 1
    - Pour l'enlever : on la met à 0
  - sports> db.sportifs.find({"genre": "homme"}, {"_id":0, "nom":1, "prenom":1})
    [{ prenom: 'Roger', nom: 'Blanchard' },
    { prenom: 'Gérard', nom: 'Lebreton' }]
  - Si on ne met que des champs à 0 : garde tous les autres champs
    - sports> db.sportifs.find({}, { "_id":0, "adresse": 0 })
      [{ prenom: 'Roger', nom: 'Blanchard', age: 55, genre: 'homme',
      disciplines: ['100m', '200m', 'descente'] },
      { prenom: 'Simone', nom: 'Blanchard', age: 52, genre: 'femme',
      disciplines: ['200m 4 nages', '100m libre'] }, ...

Modification des données

- insertMany([...]) ou insertOne({ ... }) pour insérer des éléments
- Supprimer des éléments : deleteMany({ ... }) ou deleteOne({ ... })
  - sports> db.sportifs.deleteMany({ "nom": "Blanchard" })
  - Supprime Roger et Simone des sportifs
- Supprimer toute une collection avec drop()
  - sports> db.sportifs.drop()
- Modification : update[One|Many] (<document(s)>,<modification>) avec opérations de modifications :
  - $set : ajouter ou modifier une propriété
  - $unset : supprimer une propriété
  - $inc : incrémenter un entier
  - $push / $pull : ajouter / retirer un élément dans un tableau
  - $addToSet : ajouter dans un ensemble (ne fait rien si la valeur existait)
- Remplacer un document par un autre : replaceOne(<ancien>, <nouveau>)

Modification des données

- sports> db.sportifs.updateMany(
  { "nom": "Blanchard" }, { $set: { "marie": true }})
  - Rajoute une propriété « marié » aux époux Blanchard
- sports> db.sportifs.updateMany({ "prenom": "Roger" }, { $inc: { "age": 2 }})
  - Rajoute 2 années à l'age de tous les Roger
- sports> db.sportifs.updateOne (
  { "prenom": "Régine"} , { $push: {"disciplines": "marathon" }})
  - Rajoute le marathon dans les disciplines de Régine
- sports> db.sportifs.updateOne (
  { "prenom": "Régine"} , { $addToSet: {"disciplines": "géant" }})
  - Ne fait rien car le géant est déjà dans les disciplines de Régine
- sports> db.sportifs.updateOne ({ "prenom": "Gérard"} ,
  { $addToSet: {"disciplines": { $each: ["descente", "100m"] } },
  $currentDate: { lastModified: true }})
  - Ajoute la descente et le 100m aux disciplines de Gérard avec une propriété marquant la dernière modification avec la date courante
    - lastModified: ISODate("2023-01-21T21:32:35.774Z")

Agrégation

- aggregate permet d'effectuer une série de traitements
  - Résultat du premier traitement est mis en entrée du second ...
  - Avec $lookup, permet de faire une jointure (avec un seul traitement ici)

```
{
  $lookup:
  {
    from: <collection to join>,
    localField: <field from the input documents>,
    foreignField: <field from the documents of the "from" collection>,
    as: <output array field>
  }
}
```

  - Pour notre jointure de sportifs dans les fédérations
    - « from » est la collection sportifs, « localField » (dans federations) est l'attribut sportifs, le « foreignField » (dans sportifs) est l'identifiant et on rajoute une propriété nommée listeSportifs
    - db.federations.aggregate([{
      $lookup: { from: "sportifs", localField: "sportifs",
      foreignField: "_id", as: "listeSportifs" } }])

Agrégation

- Résultat de la jointure :

```
[
  {
    _id: ObjectId("63c943a7d9b92c6462014bf0"),
    nom: "Fédération Française d'Ahtlétisme",
    acronyme: 'FFA',
    adresse: {rue: '33 avenue Pierre-de-Coubertin',ville: 'Paris', codePostal: 75013 },
    disciplines: [ '100m', '200m', 'saut hauteur', 'marathon'],
    sportifs: [ 1, 3 ],
    listeSportifs: [
      {
        _id: 1,
        prenom: 'Roger',
        nom: 'Blanchard',
        age: 57,
        genre: 'homme',
        adresse: { rue: '12 rue de Siam', ville: 'Brest', codePostal: 29200 },
        disciplines: [ '100m', '200m', 'descente' ],
        marie: true
      },
      {
        _id: 3,
        prenom: 'Gérard',
        nom: 'Lebreton',
        age: 24,
        genre: 'homme',
        adresse: { rue: '20 rue Saint-Michel', ville: 'Rennes', codePostal: 35000 },
        disciplines: ['marathon','slalom','géant','100m libre','descente','100m'],
        lastModified: ISODate("2023-01-21T21:32:35.774Z")
      }
    ]
  },
  . . . // mêmes types de d'information pour les 2 autres fédérations
```

Agrégation

- Jointure
  - Que se passe-t'il si on met dans la propriété sportifs d'un document de federations, un identifiant de sportif qui n'existe pas ?
  - Rien, il est ignoré : il n'y aucune vérification sur la jointure qui n'est pas explicitement définie comme en SQL
- Autres fonctionnalités des agrégations
  - $match : conserve les documents qui respectent un filtre
  - $group : groupe les éléments sur un champ dont les valeurs sont communes
  - $sum : fait la somme d'une propriété des éléments d'un groupe ($avg pour la moyenne …)
  - $sort : pour trier
  - …
  - Voir documentation MongoDB sur le sujet : https://www.mongodb.com/docs/v6.0/aggregation/
- De manière générale, la documentation complète de MongoDB : https://www.mongodb.com/docs/v6.0/

Accès à une base MongoDB en Java

MongoDB en Java

- On peut accéder à une BDD MongoBD en Java (ou dans tout autre langage)
  - Comme en JDBC, on a une phase de connexion avec une URL identifiant le serveur et l'utilisateur
- Requêtes sur la base
  - En mode synchrone : on fait la requête, on reste bloqué le temps que les données/réponses arrivent
    - Cf les exemples à suivre
  - En mode asynchrone : on lance la requête en arrière-plan, on abonne des méthodes qui seront appelées quand les données seront disponibles
    - Non abordé ici
- Deux niveaux de gestion des données
  - « A la JDBC » : on fait des requête et on récupère des documents génériques (du JSON manipulable en Java)
  - « A la JPA » : on crée des POJO Java et on récupère directement des instances de ces classes avec les requêtes

Connexion serveur MongoDB

- URL de connexion
  - mongodb://[username:password@]host1[:port1][,...hostN[:portN]][/[defaultauthdb][?options]]
  - On peut préciser plusieurs serveurs
  - Options pour le chiffrement ou la configuration des requêtes/serveur
- Pour notre BDD : pas d'utilisateur, un seul serveur local
  - Une fois connecté au serveur
    - Sélectionne la base de données (« sports » ici)
    - Récupère une collection de documents par son nom
  - ```
    ConnectionString connectionString =
            new ConnectionString("mongodb://localhost:27017");
    MongoClient mongoClient = MongoClients.create(connectionString);
    // utilise la base sports
    MongoDatabase database = mongoClient.getDatabase("sports");
    // récupère la collection sportifs
    MongoCollection<Document> sportifs = database.getCollection("sportifs");
    ```
    ```

Parcours des collections

- Récupérer le JSON directement (peu pratique)
  - ```
    System.out.println("Il y a "+sportifs.countDocuments()+ " sportifs :");
    for (Document doc : sportifs.find()) {
        System.out.println(doc.toJson());
    }
    ```
- On peut récupérer les propriétés JSON « à la JDBC »
  - get[Type](nomPropriete)
  - Exemple : find pour ne garder que les femmes puis une sélection de champs pour n'avoir que le nom et le prénom (sans l'id)
  - ```
    Bson champs = Projections.fields(
            Projections.include("prenom", "nom"),
            Projections.excludeId());
    FindIterable<Document> femmes =
            sportifs.find(eq("genre", "femme")).projection(champs);
    System.out.println("Identité des femmes sportives :");
    for (Document femme : femmes)
            System.out.println(" -> "+femme.getString("prenom")+
                                    " "+femme.getString("nom")) ;
    ```

Parcours des collections

- Affichage de l'exemple précédent :

```
Identité des femmes sportives :
 -> Simone Blanchard
 -> Régine Mouvier
```

- Le JSON fabriqué est le suivant :

```
[ {"prenom": "Simone", "nom": "Blanchard"},
{"prenom": "Régine", "nom": "Mouvier"} ]
```

- Classe org.bson.Document
  - Représente un document JSON dans une collection
- On retrouve avec le find Java les mêmes fonctionnalités qu'avec le Shell MongoDB
  - En écrivant les requêtes de manière programmatique
  - Cf documentation et API MongoDB

Création de documents

- Ajout d'un sportif nommé « Saturnin »
  - Version 1 : basique avec un JSON textuel
    - ```
      String json = " { _id: 12, nom: 'Legrand', prenom: 'Saturnin',"
                  + " age: 23, genre: 'homme',"
                  + " adresse: { rue: '12 rue de Navarre', ville: 'Pau', codePostal: 64000 },"
                  + "disciplines: [ 'marathon', 'descente' ] }";
      Document doc = Document.parse(json);
      sportifs.insertOne(doc);
      ```
  - Version 2 : création programmatique du document à insérer
    - ```
      Document saturnin = new Document();
      Document adr = new Document();
      saturnin.append("_id", 12);
      saturnin.append("nom", "Legrand").append("prenom", "Saturnin");
      saturnin.append("age", 23).append("genre", "homme");
      adr = new Document();
      adr.append("rue", "12 rue de Navarre");
      adr.append("ville", "Pau").append("codePostal", 64000);
      saturnin.append("adresse", adr);
      ArrayList<String> disciplines = new ArrayList<>();
      disciplines.add("marathon");
      disciplines.add("descente");
      saturnin.append("disciplines", disciplines);
      sportifs.insertOne(saturnin);
      ```

Utilisation de POJO

- Mêmes types de POJO que pour JPA
  - Classe simple (sans héritage) qui définit des attributs de même nom et type que les propriétés des documents
    - « id » pour l'identifiant (au lieu de « _id »)
  - Un getter et un setter par attribut
  - Un constructeur sans paramètre
- On définit un POJO par type de document
  - Une classe Sportif, une classe Federation et une classe Adresse

```
public class Sportif {
   private int id;
   private String prenom;
   private String nom;
   private int age;
   private String genre;
   private Adresse adresse;
   private List<String> disciplines;
   private boolean marie;
   ...
```

```
public class Federation {
   private ObjectId id;
   private String nom;
   private String acronyme;
   private Adresse adresse;
   private List<String> disciplines;
   private List<Integer> sportifs;
   ...

public class Adresse {
   private String rue;
   private String ville;
   private int codePostal;
   ...
```

Utilisation de POJO

- Au lieu d'utiliser Document dans les collections, on les type par les POJO
- La création des collections a une étape de plus
  - Doit préciser les codecs qui font le lien POJO vers JSON
  - Ici on utilise les codecs par défaut

```
// configuration des codecs par défaut
CodecProvider pojoCodecProvider = PojoCodecProvider.builder().automatic(true).build();
CodecRegistry pojoCodecRegistry =
                fromRegistries(getDefaultCodecRegistry(), fromProviders(pojoCodecProvider));

// connexion au serveur en utilisant le codec
ConnectionString connectionString = new ConnectionString("mongodb://localhost:27017");
MongoClient mongoClient = MongoClients.create(connectionString);
MongoDatabase database = mongoClient.getDatabase("sports").withCodecRegistry(pojoCodecRegistry);

// instanciation des collections typées par les POJO
MongoCollection<Federation> federations = database.getCollection("federations", Federation.class);
MongoCollection<Sportif> sportifs = database.getCollection("sportifs", Sportif.class);
```

Utilisation des POJO

- Pour afficher toutes les fédérations avec leurs sportifs
  -
```
for(Federation fed : federations.find()) {
   System.out.println("\n** "+fed.getNom()+" ("+fed.getAcronyme()+") **");
   System.out.println("Adresse : "+fed.getAdresse());
   System.out.print("Disciplines : ");
   for(String disc : fed.getDisciplines())
      System.out.print(disc+ " ");
   System.out.println("\nSportifs : ");
   for(Integer idSportif : fed.getSportifs()) {
      // on fait la jointure à la main avec un find
      Sportif sportif = (Sportif) sportifs.find(eq("_id",idSportif)).first();
      System.out.println(" - "+sportif.getPrenom()+ " "+sportif.getNom());
   }
}
```
  - Résultat :

```
** Fédération Française d'Ahtlétisme (FFA) **
Adresse : 33 avenue Pierre-de-Coubertin - 75013 Paris
Disciplines : 100m 200m saut hauteur marathon
Sportifs :
 - Roger Blanchard
 - Gérard Lebreton
...
```

Utilisation des POJO

- Insertion d'un sportif
  - Manipule directement une instance de la classe Sportif
  - ```
    Sportif saturnin = new Sportif();
    saturnin.setId(12);
    saturnin.setNom("Legrand");
    saturnin.setPrenom("Saturnin");
    saturnin.setAge(21);
    saturnin.setAdresse(new Adresse("12 rue de Navarre", "Pau", 64000));
    ArrayList<String> disciplines = new ArrayList<>();
    disciplines.add("marathon");
    disciplines.add("descente");
    saturnin.setDisciplines(disciplines);
    sportifs.insertOne(saturnin);
    ```
  - Si des propriétés / attributs manquent ou sont en trop :
    ils sont ignorés ou remplis avec des valeurs par défaut de Java
    - S'adapte à la structure flexible des documents dans les collections
    ```

Conclusion MongoDB en Java

- API Java permet de faire toutes les manipulations de documents et collections de MongoBD
  - Toutes les opérations CRUD (on ne les a pas toutes vues)
  - Utilisation de POJO
    - Très simple : pas besoin d'annotation
    - Mais mappings basiques : pas de gestion des jointures, pas de références croisées ...
- Documentations officielles sur MongoDB en Java
  - Accès synchrone (cf exemples précédents) : https://www.mongodb.com/docs/drivers/java/sync/
  - Accès asynchrone avec les reactive streams : https://www.mongodb.com/docs/drivers/reactive-streams/
  - Les Javadoc des API : https://mongodb.github.io/mongo-java-driver/4.8/apidocs/