TD06

Base de données non traditionnelles

Introduction - Le format JSON

- Notation en langage JavaScript
- Les propriétés d'un objet JSON sont matérialisée par une ou plusieurs paires de ‘‘clé’’ : ‘‘valeur‘‘.
  - La clé est le nom de l'attribut et la valeur la donnée associée
  - { ‘‘clé1’’ : ‘‘valeur de C1‘‘ , ‘‘clé2’’ : ‘‘valeur valeur de C2‘‘ }
  - Un objet vide est représenté par { }
- Une valeur peut être à son tour un objet
- Les Types de données natives :
  - Booléen (true/false), numérique, chaîne de caractères, objet,
  - Tableau : des ensembles de valeurs des types précédents
  - null (marqueur d'absence de valeur)

Introduction - Le format JSON

- La relation Personne(nom, prénom, âge) peut être exprimée en :

Modèle :

```
{
    nom,
    prenom,
    age
}
```

Donnée JSON :

```
[
    {
      ‘‘nom’’ : ‘‘BERNARD‘‘ ,
      ‘‘prénom’’ : ‘‘alice‘‘ ,
      ‘‘age’’ : ‘‘22‘‘
    },
        ...
    {
      ‘‘nom’’ : ‘‘DUPONT‘‘ ,
      ‘‘prénom’’ : ‘‘marc‘‘ ,
      ‘‘age’’ : ‘‘20‘‘
    }
]
```


Introduction - MongoDB

- MongoDB fait usage de la notation JSON et du langage JS
  - Stocke une représentation binaire du JSON dit BSON
- Ajoute aux types prédéfinis dans JSON :
  - Objectid : sur 12 octets, garanti l'unicité des identifiants générés par la BDD
  - Date : entier signé de 8 octets
  - NumberLong et NumberInt : entier sur 8 resp. 4 octets
  - NumberDecimal : flottant sur 16 octets
  - BinData : pour des chaînes de caractères ne pouvant pas être représentées en codage UTF-8
  - Et d'autres … qu'on utilisera pas dans ce cours

1 Génération de données d'index

1.1Proposer la modélisation de la collection index en écrivant un fragment de JSON

aire 1044 1063 2078 2520

aires 262 263 264 267 268 273 277 279 280 281 282

airiau 906 2095 2096 2097

ais 793 805 1527 2053 2711 3488 3612

aisance 3269

aise 198 205 206 233 234 235 236 237 238 240

aisé 1333 1334 1606 1607

aisée 595 597 3419

aisément 152 755 1766 1930

Quel modèle se dégage de cet extrait ?

1.1Proposer la modélisation de la collection index en écrivant un fragment de JSON

```
[
  { mot
    index [ ]
  },
  { mot
    index [ ]
  },
  ...
  { mot
    index [ ]
  }
]
```

aire 1044 1063 2078 2520

aires 262 263 264 267 268 273 277 279 280 281 282

airiau 906 2095 2096 2097

ais 793 805 1527 2053 2711 3488 3612

aisance 3269

aise 198 205 206 233 234 235 236 237 238 240

aisé 1333 1334 1606 1607

aisée 595 597 3419

aisément 152 755 1766 1930

1 Génération de données d'index

```
[
  { mot
    index [ ]
  },
  { mot
    index [ ]
  },
  ...
  { mot
    index [ ]
  }
]
```

aire 1044 1063 2078 2520

aires 262 263 264 267 268 273 277 279 280 281 282

airiau 906 2095 2096 2097

ais 793 805 1527 2053 2711 3488 3612

aisance 3269

aise 198 205 206 233 234 235 236 237 238 240

aisé 1333 1334 1606 1607

aisée 595 597 3419

aisément 152 755 1766 1930

La donnée au format JSON

```
[
  [
    "aire",
    "1044",
    "1063",
    "2078",
    "2520"
  ],
…
  [
    "aisément",
    "152",
    "755",
    "1766",
    "1930"
  ]
]
```


On reprend l'exemple introduit précédemment :

1. la réparation 1 utilise un filtre à air et 5l d'huile.
2. la réparation 2 utilise 4 pneus.

Un de vos camarades, débutant en bases de données traditionnelles, propose la structuration suivante pour les réparations :

```
[
  {
    "reparation": 1,
    "pieces": [
      {
        "filtreAir": 1
      },
      {
        "huile1litre": 5
      }
    ]
  },
  {
    "reparation": 2,
    "pieces": [
      {
        "pneu": 4
      }
    ]
  }
]
```

Cette modélisation vous semble-elle correcte ?
Pourquoi ?
Si non, proposez une modélisation correcte

On reprend l'exemple introduit précédemment :

1. la réparation 1 utilise un filtre à air et 5l d'huile.
2. la réparation 2 utilise 4 pneus.

Un de vos camarades, débutant en bases de données traditionnelles, propose la structuration suivante pour les réparations :

```
[
  {
    "reparation": 1,
    "pieces": [
      {
        "filtreAir": 1
      },
      {
        "huile1litre": 5
      }
    ]
  },
  {
    "reparation": 2,
    "pieces": [
      {
        "pneu": 4
      }
    ]
  }
]
```

Pas d'information dans la clé. L'information est à mettre dans la valeur

```
[
 { ''répartition'':1,
      ''pièces'' : [
      {
       ''nom'' :''filtreAir''
      ''Qte'' : ''1''
      },
      {
       ''nom'' :''huile''
      ''Qte'' : ''5''
      }
      ]
 },
 {''répartition'':2,
      ''pièces'' : [
      {
       ''nom'' :''pneu''
      ''Qte'' : ''4''
      }
      ]
 }
]
```


1.2 Proposer un script jq permettant de transformer les données du fichier texte en BSON que l'on peut insérer dans MongoDB

- Avant d'injecter un fichier data MongoDB, il faut s'assurer qu'il est bien indenté
- MongoDB :
  - Demande en entrée un BSON = Binary JSON
  - L'enregistrement dans un fichier est dit DOCUMENT
  - La BDD n'est pas constituée de tables mais de collections
- On va utiliser l'utilitaire jq
  - Est un outil en ligne de commande pour manipuler du JSON
  - Syntaxe : $ jq ' requete ' nomFichiersurlequelAppliquerReq.json
    Il prend une entrée et produit une sortie
  - On peut utiliser la VM ou l'outil en ligne https://jqplay.org/

1.2 Accès aux données selon des sélecteurs exo1

- $ jq . immo.json
  Si on a de la couleur en sortie c'est que le fichier immo.json a pu être bien indenté
  Le . ici indique la racine de la donnée
- $ jq .[ ] immo.json
  Demande l'ensemble des documents dans immo.json
  Dans les deux cas : Si le fichier n'est pas au format JSON alors erreur
- Ces deux commandes ne font aucune sélection, elles ne font qu'afficher l'entrée. Il faut ajouter un sélecteur pour effectuer une sélection

1.2 Accès aux données selon des sélecteurs exo1

- $ jq .[1] immo.json
  Demande le document d'index 1 dans immo.json (le 1er index est 0)
  Ici le . pointe sur l'élément actuellement parcourue
- $ jq .[2,4] immo.json
  Demande les documents d'index 2 et 4 dans immo.json
  Si on demande l'index k et qu'il n'existe pas renvoie null
  Pour ne pas se tromper vérifier le nombre de documents en mode compact
  $ jq -c .[ ] data1.json | wc -l on obtient le nombre de ligne

1.2 Accès aux données selon des sélecteurs exo1

- $ jq .[ ].id immo.json
  - Demande la valeur de la propriété id dans les documents de immo.json
  - si id n'existe pas ou mal hiérarchisé retourne null

1.2 Accès aux données selon des sélecteurs exo1


```
[{
    "id": "600ab7774fe33cf22a658290",
    "isActive": false,
    "picture": "http://placehold.it/32x32",
    "title": "do pariatur officia esse nulla",
    "price": 742407,
    "owner": {
        "firstName": "Parsons",
        "lastName": "Hubbard",
        "email": "parsons.hubbard@test.me",
        "phone": "+1 (898) 482-2191"
    },
    "address": "242 Monitor Street, Floriston, Northern Mariana Islands, 2615",
    "informations": "Laborum esse do deserunt eu ipsum laborum et veniam dolor elit et aute. Cupidatat ad elit do
}
]
```


$ jq .[].id data1.json

Renvoie : "600ab7774fe33cf22a658290"

1.2 Accès aux données selon des sélecteurs exo1


```
[{
    "id": "600ab7774fe33cf22a658290",
    "isActive": false,
    "picture": "http://placehold.it/32x32",
    "title": "do pariatur officia esse nulla",
    "price": 742407,
    "owner": {
        "firstName": "Parsons",
        "lastName": "Hubbard",
        "email": "parsons.hubbard@test.me",
        "phone": "+1 (898) 482-2191"
    },
    "address": "242 Monitor Street, Floriston, Northern Mariana Islands, 2615",
    "informations": "Laborum esse do deserunt eu ipsum laborum et veniam dolor elit et aute. Cupidatat ad elit do r
}
]
```


$ jq .[].title data1.json

Renvoie : "do pariatur officia esse nulla"

1.2 Accès aux données selon des sélecteurs exo1


```
[{
    "id": "600ab7774fe33cf22a658290",
    "isActive": false,
    "picture": "http://placehold.it/32x32",
    "title": "do pariatur officia esse nulla",
    "price": 742407,
    "owner": {
        "firstName": "Parsons",
        "lastName": "Hubbard",
        "email": "parsons.hubbard@test.me",
        "phone": "+1 (898) 482-2191"
    },
    "address": "242 Monitor Street, Floriston, Northern Mariana Islands, 2615",
    "informations": "Laborum esse do deserunt eu ipsum laborum et veniam dolor elit et aute. Cupidatat ad elit do r
}
]
```


$ jq .[].firstName data1.json

- Renvoie : **null** (car mal hiérarchisé)

1.2 Accès aux données selon des sélecteurs exo1

- $ jq .[].owner immo.json
  - Demande la valeur de la propriété owner dans les documents de immo.json
  - Owner est lui même constitué de plusieurs propriétés
- $ jq -c .[].owner immo.json
  - La version compacté sur une ligne de la précédente commande
- $ jq .[].owner.firstName immo.json
  - Demande la valeur de la propriété owner.firstName dans les documents de immo.json
- Voir img_01

1.2 Accès aux données selon des sélecteurs + filtresexo1

- jq utilise le pipe | pour connecter plusieurs opérations ensemble
- jq répétera le filtre pour chaque objet JSON fourni par l'étape précédente.
- jq <filter> [files…]
  - Le filtre doit être entouré de guillemets simples

1.2 Accès aux données selon des sélecteurs + filtresexo1

- $ jq '.[] | select(.owner.firstName == "Kirk")' immo.json
  - Revoie tout le contenu des documents dont owner.firstName est "Kirk". S'il n'en existe pas la sortie est vide
- $ jq '.[] | select(.owner.firstName == "Kirk") | {id, price}' immo.json
  - Revoie la valeur des propriétés id et price des documents dont owner.firstName est "Kirk"
- Voir img_02

1.2 Accès aux données selon des sélecteurs +filtreexo1

- $ jq '.[] | select(.owner.firstName | test("^Jo")) | {owner}' immo.json
  - Revoie tout le contenu de owner des documents dont owner.firstName commence par un "Jo"
- $ jq '.[1] | keys' immo.json
  - Renvoie l'ensemble des clés du document d'index 1
- Voir img_03 & img_04

1.2 Accès aux données selon des sélecteurs +filtre exo1

- map(<filtre>) : Le filtre **map(x)** applique le filtre **x** à chaque élément et renverra les sorties dans un nouveau tableau
- $ jq 'map(has("**propriété**"))' nom.json
  - renvoie true/false selon que la propriété existe ou pas pour chacun des documents de la collection
  - $ jq 'map(has("isActive"))' data1.json renvoie 1 true
  - $ jq 'map(has("isActive"))' immo.json renvoie 11 true
  - $ jq 'map(has("foo"))' data1.json renvoie 1 false
- Voir img_05

1.2 Accès aux données selon des sélecteurs +filtre exo1

Le filtre **tonumber** transforme une chaine en nombre

Commande → Sortie

- $echo ' "03.14" ' | jq tonumber → 3.14
- $echo ' "0xa" ' | jq tonumber → Renvoie une erreur
- $echo ' "bonjour" ' | jq tonumber → Renvoie une erreur

$ jq -cn --args '$ARGS.positional[] | try [., tonumber] catch "n est pas un nbr"' 3.14 bonjour 1e-3 0xa

```
["3.14",3.14]
"n est pas un nbr"
["1e-3",0.001]
"n est pas un nbr"
```


1.2 Accès aux données selon des sélecteurs exo2

- Analysons la progression des instructions suivantes :

1) jq ' ' index.test

Équivalent à jq . index.test

Erreur si ce n'est pas du JSON (ce qui est le cas dans notre exemple)

2) jq --raw-input ' ' index.test

Équivalent à 1) l'option **--raw-input** considère que l'entrée est au format quelconque et pas nécessairement JSON. Chaque ligne de l'entrée est transmise au filtre sous forme de chaîne

3) Attention dans les commandes suivantes (2 à 10) le fichier manipulé est nécessairement NON JSON

Voir commandes exo2 à l'écran

2 Transformation des textes en JSON exo3

2.2.1 Commencer par transformer les textes en JSON

- Deux fichiers :
  - textes nommés par un identifiant par exemple 96
  - Des calcul de TF-iDF par exemple 96.tfidf

1) En posant pageId=96

2) On cherche à trouver la syntaxe qui :

- s'approche de : jq --arg pageId $pageId ' .... {"_id": $pageId …}
- et répond à la problématique : « Proposer dans un premier temps une transformation du texte de la page en un fragment JSON »

- Réponse : $ jq --raw-input --slurp --arg pageId $pageId '{"_id": $pageId, "text":.}' $pageId


```
{
  "_id": "96",
  "text": "Join the “Graphemics in the 21st century” conference in June 2018!\nhttp://conferences.telecom-bretagne.eu/grafematik/\nICBM address: 48°21'31.57\"N 4°34'16.76\"W\n\n"
}
```

2 Transformation des textes en JSONexo3-B
2.2.2 Transformons les tableaux de (mot, tfidf) en JSON

- head $pageId.tfidf | jq --raw-input --slurp ''
- head $pageId.tfidf | jq --raw-input --slurp '{"words": split("\n")}'
- head $pageId.tfidf | jq --raw-input --slurp '{"words": split("\n") | map(split(" ")) }'
- head $pageId.tfidf | jq --raw-input --slurp '{"words": split("\n") | map(split(" ")) | map({"word": .[0], "tfidf": .[1]})}'
- head $pageId.tfidf | jq --raw-input --slurp '{"words": split("\n") | map(split(" ")) | map({"word": .[0], "tfidf": .[1]})}'
- head $pageId.tfidf | head -c -1 | jq --raw-input --slurp '{"words": split("\n") | map(split(" ")) | map({"word": .[0], "tfidf": .[1]})}'

2 Transformation des textes en JSONexo3-C
2.2.3 Assemblage des deux transformations

- Additionner deux arbres JSON avec jq

  (req_jq_01 ; req_jq_02) | jq -c -s add
- Effectuer l'addition sur l'ensemble des textes

```
for pageId in $(seq 3655); do
    ( req_jq_01 ; req_jq_02 ) | jq -c -s add ;
done > ../pages.bson
```


Mongo DB

- Stocke les données sous un format binaire appelé BSON (Binary Json) permettant de sérialiser des objets JavaScriot en binaire (suivant un format dit clés/valeurs)
- Une approximation très grossière :
  - Tables → collections
  - Tuples → documents
- Sous MongoDB une BDD n'est pas censée être construite suivant un schéma prédéfini : donc, les tuples ne sont pas censés posséder une liste identiques d'attributs

Index

- Chaque collection est indexée par défaut sur l'attribut _id
- Vous pouvez à tout moment créer un autre index
- pour voir les index posés sur une collection : **db.<NomDeLaCollection>.getIndexes()**
- Pour créer un index sur une collection : **db.<NomDeLaCollection>.createIndexes(<attributes>,<options>)**

Index

>db.biens.getIndexes()

[{ "v" : 2, "key" : { "_id" : 1 }, "name" : "_id_" }]

- Chaque collection est indexée par défaut sur l'attribut _id

> db1.terms.getIndexes()

[{ "v" : 2, "key" : { "_id" : 1 }, "name" : "_id_" }]

> db1.pages.getIndexes()

[{ "v" : 2, "key" : { "_id" : 1 }, "name" : "_id_" }]

Peuplement

```
{
    nom,
    prenom,
    age
}
```

```
>db.personne.insert(
{"nom":"Durand","prenom":"Alice","age":25}
)
```

- Insert un document dans la collection personne. Renvoie la valeur de oid du document crée
- Mongo ajoute le champ _id de type ObjectID (int sur 12 octets) à chaque ligne du document

```
>db.personne.insertMany ([
{"nom":"Dupont","prenom":"Bob","age":27},
{"nom":"LeGrand","prenom":"Joyce","age":22,
"Fonction":"etudiant"},
{"nom":"LeJEune","prenom":"Aline","age":22,"
Fonction":"developeur"},
{"nom":"Donald","prenom":"Alice","age":20,
"sport" : [{"nom":"Fitness","nb":2},
   {"nom":"Running","nb":1},
   {"nom":"Judo","nb":1}] }
])
```


Peuplement

```
>db.personne.insert(
...
)
>db.personne.insertMany([
...
])
```

Depuis Linux

- $ mongoimport -d database_name -c collection_name --file collection_name.json
- TP 6.1

  $ jq -c .[] immo.json | mongoimport -d immo -c biens

Interrogation d'une collection

- Interrogation via un objet 'filtre'
  - La méthode find() sans paramètres

    db.<NomDeLaCollection>.find() Sélection de tous les documents de la collection

    (équivalent au SELECT * FROM NomDeLaCollection)
  - La méthode find() avec paramètres

    db.<NomDeLaCollection>.find(<ObjetFilter>)
    - Syntaxe du **<ObjetFilter> {** {critere} , {projection} **}**
      - critere : expression qui précise ce qu'on veut récupérer (équivalent de ce qu'on met dans un WHERE)
      - projection : indiquer quels sont les champs qu'on veut récupérer

Interrogation d'une collection

- Interrogation via un objet 'filtre' : find() sans paramètres
  - **db.<NomDeLaCollection>.find()** tous les doc de la collection
  - **db.<NomDeLaCollection>.find().pretty()** Affichage formaté
  - **db.<NomDeLaCollection>.find().count()** le nombre de doc de la collection
  - **db.<NomDeLaCollection>.find()[0]** le 1er document de la collection (s'il existe)
  - **db.<NomDeLaCollection>.find()[0].<nomPropriete>** renvoie le champs spécifié par nomPopriete du document 0. Ne s'applique pas sur l'ensemble des docs.

Interrogation d'une collection

- Interrogation via un objet 'filtre'
  - La méthode find() avec paramètres **db.<NomDeLaCollection>.find(<ObjetFilter>)**
- Syntaxe du **<ObjetFilter>** { critere }, { projection }

SELECT age FROM personne WHERE nom="Gallois" ;

> db.personne.find({"nom": "Gallois"} , {age:1,_id:0})

Interrogation d'une collection

- SELECT nom FROM personne ;

  > db.personne.find({},{"nom":1})
- Dans la projection :
  - Le 1/true dans "nom":1 Afficher le champ
  - Le 0/false dans "nom":0 NE PAS afficher le champ
  - Par défaut Mongo affiche le _id → Mettre _id:0 pour ne pas l'afficher

  > db.personne.find({},{"nom":1,_id:0})

  "nom":0 permet de ne pas afficher le champ nom

```
{
    _id,
    nom,
    prenom,
    age
}
```


Interrogation d'une collection

- **db.<NomDeLaCollection>.find(<ObjetFilter** { critere }, { projection }**>)**
- SELECT * FROM personne ;

  > db.personne.find()

  > db.personne.find({}) → Affiche les documents vides
- SELECT nom FROM personne ;

  > db.personne.find({} , { "nom":1 , _id:0 })
- SELECT age FROM personne WHERE nom="Gallois" ;

  > db.personne.find({"nom": "Gallois"} , {age:1,_id:0})
- SELECT * FROM personne WHERE nom="Gallois" ;

  > db.personne.find({"nom": "Gallois"})

```
{
    _id,
    nom,
    prenom,
    age
}
```

Interrogation d'une collection

- La méthode find() avec paramètres

On peut utiliser dans critere de l'objet **<ObjetFilter>** :

- Les opérateurs de comparaison
- Les opérateurs logiques
- Les opérateurs : $exists ; ...
- Les opérateurs d'évaluation : $regex ; $text ; $where

Interrogation d'une collection

- Opérateurs de comparaison :
  - **$eq** : renvoie vrai si la valeur de la propriété est égale à la valeur spécifiée
  - **$ne** : renvoie vrai si la valeur de la propriété est différente de la valeur spécifiée
  - **$lt** : renvoie vrai si la valeur de la propriété est inférieure à la valeur spécifiée
  - **$lte** : renvoie vrai si la valeur de la propriété est ≤ à la valeur spécifiée
  - **$gt** : renvoie vrai si la valeur de la propriété est supérieure la valeur spécifiée
  - **$gte** : renvoie vrai si la valeur de la propriété est ≥ à la valeur spécifiée
  - **$in** : renvoie vrai si la valeur de la propriété est dans le tableau

Interrogation d'une collection

- Opérateurs logiques : $and, $or, $nor, $not
  - Opérateur **$and** :

    { $and: [{ Expression1 }, { Expression2 }, ..., { ExpressionN }] }

    ou { { Expression1 }, { Expression2 }, ..., { ExpressionN }}
    - Effectue une opération de ET logique sur le tableau d'une ou plusieurs expressions
    - Équivalent à l'opérateur **$and**
  - Opérateur **$or** :

    { $or: [{ Expression1 }, { Expression2 }, ..., { ExpressionN }] }
    - Effectue une opération de OU logique sur le tableau d'une ou plusieurs expressions
    - Équivalent à l'opérateur **$in**

• Interrogation d'une collection

Exemple : Opérateurs logiques : $and, $or

```
> db.biens.find(
    {$and :
      [
          { "price" : {$gt:500} } ,
        {"price":{$lt:800000} }
      ]
    } ,
    {"title":1}
  )
```

```
> db.biens.find(
    {$or :
    [
      {"price" : {$lt : 500000}  } ,
      {"price" : {$gt : 800000} }
    ]
    },
    {"title" : 1}
  )
```

Interrogation d'une collection

- Opérateur **$all**

  {<field>: { $all: [<value1> , <value2> ...] } }

  - Sélectionne les documents où la valeur d'un champ est un tableau qui contient tous les éléments spécifiés
  - Équivalent à **$and**

Interrogation d'une collection

- SELECT nom FROM personne WHERE age > 25 ;

```
> db.personne.find( {"age":{$gt:25}},{"nom":1,_id:0} )
```

- Afficher ceux qui ont le champ Fonction

```
> db.personne.find( {"Fonction":{$exists:true}},{"nom":1,_id:0} )
```

- Afficher tous les étudiants

```
> db.personne.find( {"Fonction":"étudiant"},{"nom":1,_id:0} )
```

- Afficher les noms des personnes de plus de 25 ans avec leur age

```
> db.personne.find( {"age":{$gt:25}},{"nom":1,"age":1 ,_id:0} )
```

- Trier les personnes selon l'age

```
{
    _id,
    nom,
    prenom,
    age
}
```


Agrégations

- Les agrégats :

  $group ; $limit ; $project ;

  $sort (où 1=asc et -1=dsc)

  $match ; $assFiels ;

  $count ; lookup ; $out ;

- Exemple : Trier les personnes selon l'age

  Interrogation :

  db.personne.find().sort({"age":-1})

  Agrégat :

```
db.personne.aggregate( [
 {
  $sort : { "age": -1 }
 }
 ,
 {
  $project : {
    "name" : 1,
    "prenom" : 1,
    "age" : 1
     }
 }
] )
```


Interrogation d'une collection

2.3 Requêter exo5 Exemple sur db.termes

```
> db.terms.find( )[0]
```

On en déduit le modèle du premier document

- les documents (tuples) ne sont pas censés posséder une liste identiques de champs

```
_id,
index [
    ...
    ]
```

Interrogation d'une collection

2.3 Requêter exo5 Exemple sur db.termes

Req01 : Lister les pages contenant le terme abcd :

> db.terms.find({_id:"abcd"})

Renvoie

{ "_id" : "abcd", "index" : [3127, 3136, 3578] }

Interrogation d'une collection

2.3 Requêter exo5 Exemple sur db.pages

- > db1.pages.find()[0]

On en déduit un premier modèle

- les documents ne sont pas censés posséder une liste identiques d'attributs

```
_id,
text,
words:[
    {   "word"
        "tfidf"
    },
    …
    {   "word"
        "tfidf"
    }
    ]
```


Interrogation d'une collection

2.3 Requêter exo5 Exemple sur db.pages

- La méthode find() **sans** paramètres

> db1.pages.find()

> db1.pages.find()[0]

> db1.pages.find()[0].words

> db1.pages.find().count()

3655

- La méthode find() **avec** paramètres

> db1.pages.find({"words.word":"abcd"})

> db1.pages.find({"words.word":"abcd"}, {_id:1})

> db1.pages.findOne({"words.word":"abcd"}, {_id:1})

{ "_id" : "3127" }

```
_id,
text,
words:[
    {   "word"
        "tfidf"
    },
    …
    {   "word"
        "tfidf"
    }
    ]
```


Interrogation d'une collection

2.3 Requêter exo5 Exemple sur db.pages

- La méthode find() **sans** paramètres

> db1.pages.find()

- La méthode find() **avec** paramètres

> db1.pages.find({"words.word":"abcd"})

```
_id,
text,
words:[
    {   "word"
        "tfidf"
    },
    …
    {   "word"
        "tfidf"
    }
    ]
```

Interrogation d'une collection

2.3 Requêter exo5 Exemple sur db.pages

Req02 : Lister les pages contenant le terme abcd :

> db.pages.find({"words.word":"abcd"} , {_id:1})

Renvoie

{ "_id" : "3127" }
{ "_id" : "3136" }
{ "_id" : "3578" }

```
_id,
text,
words:[
    {   "word"
        "tfidf"
    },
    …
    {   "word"
        "tfidf"
    }
    ]
```


Interrogation d'une collection

2.3 Requêter exo5

Req03 : Lister les pages qui contiennent les trois terms poste, machine, learning.

> db.pages.find({},{"words.word":1, _id:0}).pretty().count()
3655
> db.pages.find({"words.word": { $regex:"poste"} }).count()
809

- Proposez une solution pour la requête 3 ….
  1) On utilise UNE seule collection, la quelle ?
  2) Quel(s) opérateur(s) ?

```
_id,
text,
words:[
    {   "word"
        "tfidf"
    },
    …
    {   "word"
        "tfidf"
    }
    ]
```


Interrogation d'une collection

2.3 Requêter exo5

> db.pages.find({},{"words.word":1, _id:0}).pretty().count()
3655
> db.pages.find({"words.word": { $regex:"poste"} }).count()
809
> db.pages.find({"words.word": { $regex:"machine"} }).count()
382
> db.pages.find({"words.word": { $regex:"learning"} }).count()
307

- Quel sera le nombre de page au vu de ce qui précède ?
- Quel(s) opérateur(s) choisir ?

```
_id,
text,
words:[
    {   "word"
        "tfidf"
    },
    …
    {   "word"
        "tfidf"
    }
    ]
```


Interrogation d'une collection

2.3 Requêter exo5

- Réponse à Req03-Opérateur **$and** :

> db.pages.find(**{** **$and**:[{"words.word" : "poste"} , {"words.word" : "machine"}, {"words.word" : "learning"}] **}**)

- Réponse à Req03-Opérateur **$all** :

> db.pages.find(**{** "words.word" : { **$all** : ["poste", "machine", "learning"] } **}**)

- Pour cet exemple, le count renvoie 34

```
_id,
text,
words:[
    {   "word"
        "tfidf"
    },
    …
    {   "word"
        "tfidf"
    }
    ]
```

Préparation au TP Partie MongoDB

1)Lancer la machine virtuelle utilisée lors du TP04

$ ssh -p 2222 tp@127.0.0.1

2)Se connecter au serveur :

$ mongo

3)Obtenir la liste des bases

> show databases (show dbs)

> help sur le système

- Avant de requêter il faut accéder à la DB avec

  > use un_nomDB

- Lister les collections

  > show collections

- > db.help() sur la DB
- Créer une collection

  > db.client.insert({nom: "toto"})

- Pour quitter mongo > quit() ou exit ou ^C

Préparation au TP 6.1
Premier contact avec MongoDB

- Partie 1
  - Création et manipulation de la DB client sous MangoDB

    > db.client.insert({nom: "toto"})
- Partie 2
  - Téléchargement des données sur la VM

    1) Téléchargement de immo.json depuis Ecampus

    2) Depuis la machine réelle $ scp -P 2222 immo.json tp@127.0.0.1:
  - Initiation jq
  - Quelques requêtes pour s'entraîner (tout en bas celles de MangoDB)
- Partie 3
  - Depuis la VM importation des données vers MongoDB
  - Quelques requêtes pour s'entraîner

2.3 Importation des données TD-exo4

TP6.1- Partie 3

1) Depuis la VM importation des données vers MongoDB

$ jq -c .[] immo.json | mongoimport -d immo -c biens

Réponse du serveur :

connected to: mongodb://localhost/

11 document(s) imported successfully. 0 document(s) failed to import.

2) Se connecter pour le requêtage

$ mongo

> show dbs Lister les DB disponibles

> use immo Accéder à la DB avec

> show collections Lister les collections

> db.help()

> db.biens.find().count()

Préparation au TP 6.2

Partie 1

Importation des données à faire lors du TP6.1

Partie 2

script JS dans Mongo (la semaine prochaine)