Afpa
se former, avancer

TRAVAUX PRATIQUES

PERSISTER DES DONNÉES NON STRUCTURÉES (SEMI-STRUCTURÉES) DANS UN SGBD NOSQL

Afpa 2025 – Concepteur Développeur d'Applications

Révision 2025 - Version 1.0

Jérôme BOEBION

Table des matières

Objectifs : 3
Pré requis : 3
Méthodologie : 3
Mode d'emploi : 3
Ressources 3
1. INTRODUCTION 4
2. DECOUVRIR NOSQL 5
2.1 DEFINITION DU BIG DATA : LES « TROIS V » 5
2.1.1 Le volume 6
2.1.2 La variété 6
2.1.3 La vitesse 7
2.1.4 Un critère supplémentaire : la véracité 7
2.2 THEOREME DE CAP (CONSISTENCY, AVAILABILITY, PARTITION TOLERANCE) 7
2.2.1 Aucun système n'est parfait : sur un exemple 7
2.2.2 Aucun système n'est parfait : cas général 8
2.3 CLASSIFICATION DES SGBD SQL ET NOSQL SELON LE THEOREME DE CAP 9
2.3.1 AC : Availability, Consistency 9
2.3.2 PC : Partition Tolerance, Consistency 10
2.3.3 AP : Availability, Partition Tolerance 10
2.4 LES FAMILLES DE NOSQL 11
2.4.1 Couple clé/valeur 11
2.4.2 Base documentaire 12
2.5 QUE CHOISIR POUR SES DONNEES : SQL OU NOSQL ? 12
3. METTRE EN PRATIQUE NOSQL AVEC MONGODB 14
3.1 INSTALLER LE SERVEUR MONGODB 14
3.2 INSTALLER LE CLIENT Studio3T (anciennement Robot3T) 15
3.3 CREER ET REMPLIR UNE COLLECTION DE TEST 18
3.4 INTERROGER *MONGODB* EN *JAVASCRIPT* DEPUIS LE CLIENT Studio3T 22
3.4.1 Interrogations simples 23
3.4.2 Projection 25
3.4.3 Filtre 27

3.4.4 Filtre et Projection 29
3.4.5 Filtrer avec des expressions régulières 31
3.4.6 Filtrer avec des opérateurs 34
3.4.7 Le pipeline d'aggrégation : aggregate 42
3.4.8 Utiliser des variables JavaScript pour simplifier le code des requêtes 44
3.4.9 Le pipeline avec group by et fonctions d'agrégation 45
3.4.10 Utiliser d'autres fonctions d'agrégation : moyenne, min, max 48
3.4.11 Modifier la base : ajouter un champ, modifier un champ, supprimer un champ sur un objet existant 50
3.4.12 Modifications multiples (sur plusieurs objets d'une collection) 53
3.4.13 Renommer un attribut 54
3.4.14 Manipuler des attributs de type tableau 55
3.4.15 Insérer de nouveaux enregistrements dans une collection 56
3.4.16 Supprimer des enregistrements, selon un critère 57
3.4.17 Utiliser JavaScript pour des opérations plus complexes. 58
3.4.18 Faire une "jointure" entre plusieurs collections 60
3.5 APPELER MONGODB DEPUIS UN CLIENT JAVA 62
3.5.1 Création du projet Maven 62
3.5.2 Les premiers pas : valider l'environnement 63
3.5.3 Une première requête simple 64
3.5.4 Une requête paramétrée 65
3.5.5 Insérer de nouveaux objets et les afficher 65
3.5.6 Supprimer des objets dans une collection 66
Code complet Java : 68
Reproduction interdite 70

Objectifs :

A l'issue de cette séance, le stagiaire sera capable de :

- Comparer SQL et NoSQL et faire le meilleur choix pour ses données
- Comprendre les différences entre les familles de NoSQL
- Découvrir NoSQL par la pratique avec *MongoDB*

Pré requis :

Cette séance intervient après l'apprentissage des bases de données relationnelles et du langage SQL, qui demeurent des repères pour l'apprentissage.

Méthodologie :

Ce document peut être utilisé en présentiel ou à distance.

Il précise la situation professionnelle visée par la séance, la resitue dans la formation, et guide le stagiaire dans son apprentissage et ses recherches complémentaires.

Mode d'emploi :

Symboles utilisés :

📖 Renvoie à des supports de cours, des livres ou à la documentation en ligne constructeur.

💻 Propose des exercices ou des mises en situation pratiques.

💣 Point important qui mérite d'être souligné !

Ressources

- Documentation en ligne de MongoDB : https://docs.mongodb.com/manual/
- Présentation en français : https://www.mongodb.com/fr
- Introduction au NoSQL : https://fr.wikipedia.org/wiki/NoSQL

1. INTRODUCTION

NoSQL (« Not Only SQL ») désigne une nouvelle approche de la persistance de données, qui ne s'appuie plus sur le modèle relationnel.

- Les bases de données NoSQL, telles que MongoDB, ne sont donc pas des SGBDR. Elles peuvent si nécessaire être interrogées et mises à jour en SQL, mais ce n'est pas leur langage natif.

- L'approche NoSQL répond au problème des Big Data : par exemple, des moteurs de recherche comme Google, ou des réseaux sociaux comme Facebook ou Twitter doivent stocker et manipuler d'énormes quantités de données très diversifiées (structurées, non structurées ou semi-structurées), qui doivent être distribuées sur de nombreux serveurs. Or les SGDBR et le langage SQL n'ont pas été prévus à l'origine pour :
  - Manipuler des données hétérogènes et évolutives : il faut prévoir à l'avance les entités et leurs attributs, les tables et leurs colonnes, les clés primaires et étrangères etc.
  - Distribuer de gros volumes de données sur différents serveurs synchronisés.

- Nous allons préciser les insuffisances du langage SQL face aux Big Data et présenter les différentes réponses de l'approche NoSQL. Il s'agit d'une approche et pas d'une norme commune :
  - Il n'est pas si simple de remplacer SQL : chaque SGBD NoSQL répond bien à un ou plusieurs problèmes posés par les Big Data mais pas à tous
  - Il faudra donc savoir choisir le bon SGBD NoSQL en fonction du type de données à manipuler, ou au moins comprendre le choix effectué par un expert.

- Nous allons présenter quelques-unes des familles NoSQL et leurs avantages respectifs, sans chercher à devenir nous-même expert dans ce domaine.

- Nous mettrons en pratique cette nouvelle approche, avec la base MongoDB :
  - Avec un requêteur JSON pour l'apprentissage
  - Puis en réalisant un client Java.

2. DECOUVRIR NOSQL

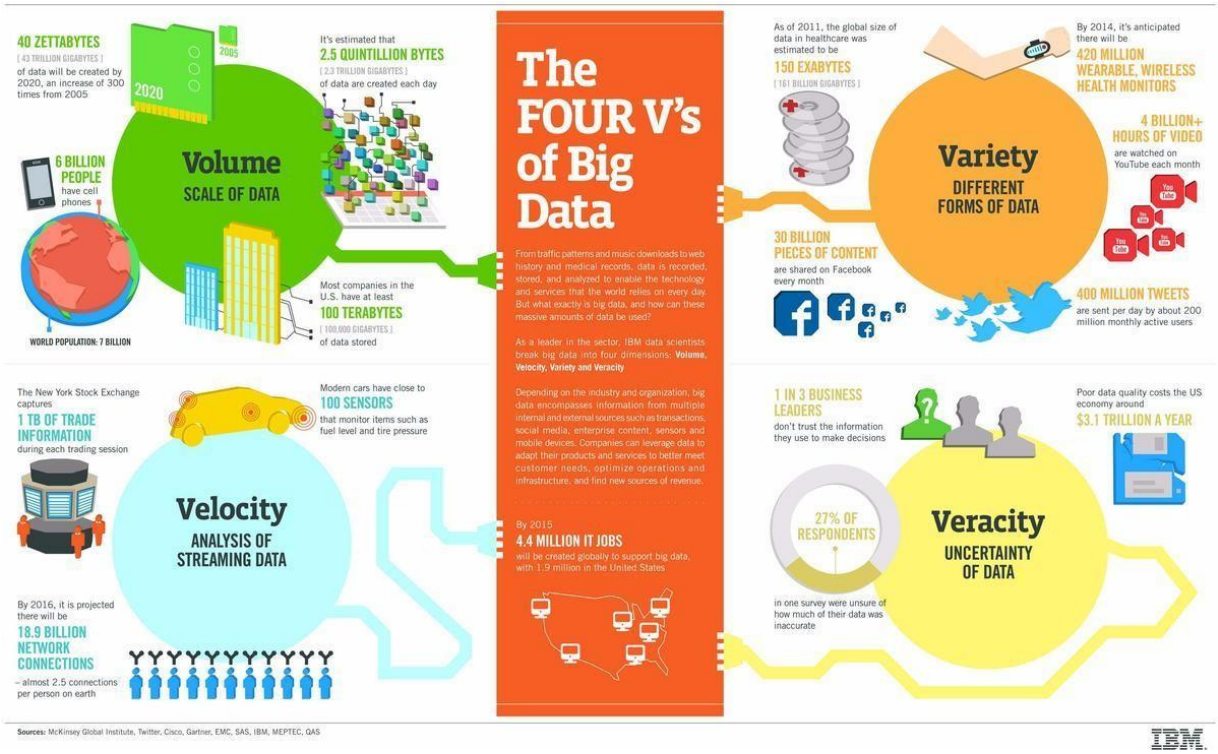
NoSQL est une approche qui répond aux problèmes techniques du *Big Data*.

2.1 DEFINITION DU BIG DATA : LES « TROIS V »

Le *Big Data* est apparu avec le développement des techniques numériques : réseaux sociaux, appareils mobiles, objets connectés, multiplication des recherches et des transactions sur Internet etc.

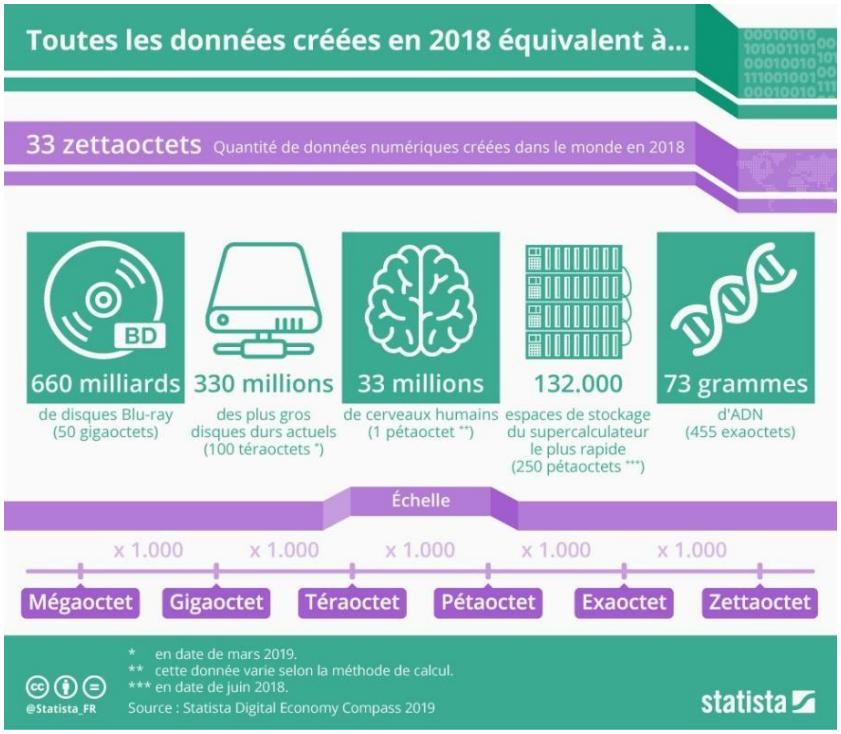
Il en a résulté une profonde évolution de la nature des données à traiter, que l'on peut résumer par le critère des « trois V » : **Volume**, **Vitesse**, **Variété**.

IBM ajoute un quatrième critère : la **Véracité** des données.



2.1.1 Le volume

C'est la principale caractéristique du *Big Data*.



33 Zettaoctet : (10^{21}) de données numériques ont été créées dans le monde en 2018, soit pour donner une idée concrète, 33 milliards de disque de 1 Tera. Avec une prévision de 40 Zettaoctet en 2020. Ce volume a été multiplié par 300 depuis 2005. Les données proviennent à la fois des entreprises et des particuliers.

Pour revenir à la technique, retenons un point précis : ces énormes volumes de données posent des problèmes de stockage et de temps de réponse dans les traitements.

Ceci exige de nombreux serveurs avec des traitements distribués, qui n'étaient pas prévus au départ par le modèle relationnel et sont difficiles à gérer.

2.1.2 La variété

Au-delà du volume, c'est aussi la diversité des données qui a augmenté : provenance, format, domaine.

Cette diversité exige des traitements différenciés selon les domaines, avec des niveaux de fiabilité et de sécurité variés. Deux cas extrêmes :

- Les données du domaine médical, sur des appareils connectés de « monitoring » : grande fiabilité et disponibilité, exigence de sécurité
- Les données publiques sur Facebook, les Tweets : peu d'exigence de sécurité, mais grande variété dans les formats à traiter.

Point technique : cette grande diversité des données du *Big Data* gène l'approche relationnelle, qui exige de prévoir à l'avance les entités et leurs attributs.

Dans un même domaine (par exemple le marketing), les données évoluent sans cesse, en introduisant de nouveaux indicateurs de tendance etc.

2.1.3 La vitesse

Les données doivent être collectées plus rapidement, souvent en temps réel, en provenance d'objets connectés par Internet (*IoT*).

Selon le premier schéma (IBM), une voiture récente possède environ 100 capteurs connectés. Chaque session de la bourse de New York réalise en moyenne 1 To de transactions, qui doivent bien sûr être traitées en temps réel.

2.1.4 Un critère supplémentaire : la véracité

Ces trois critères (volume, variété, vitesse) suffisent à définir le *Big Data*.

IBM leur ajoute la « véracité », qui est un critère classique pour les bases de données et constitue un enjeu majeur pour le *Big Data*.

Toujours selon le schéma d'IBM, 27% des entreprises reconnaissent qu'elles ne sont pas certaines de l'exactitude des données qu'elles ont collectées.

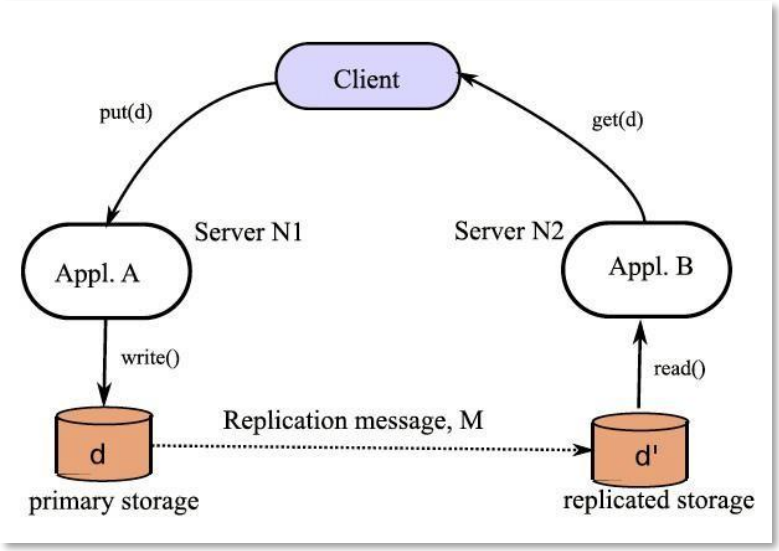
2.2 THEOREME DE CAP (CONSISTENCY, AVAILABILITY, PARTITION TOLERANCE)

Pour faire un choix, il faut connaître les limites des systèmes SQL et NoSQL, qui sont décrites selon trois critères par le théorème de CAP.

2.2.1 Aucun système n'est parfait : sur un exemple

Commençons par un cas particulier, avec deux serveurs N1 et N2 :

- Le serveur N1 est « maître », il est le seul à pouvoir écrire sur le disque, à la suite des demandes d'écriture du client
- Le serveur N2 est « esclave », il ne répond qu'aux demandes de lecture du client.



Dans cet exemple d'architecture distribuée, le serveur N1 duplique ses données sur le serveur N2 qui lui sert de miroir.

Pendant la duplication du message **M**, les deux serveurs ne sont pas synchronisés :

- Le disque **d** est différent de sa copie **d'**
- Cette architecture ne respecte pas la propriété de « Cohérence » (*Consistency*).

Si un serveur tombe en panne, l'ensemble ne fonctionne plus, car on perd soit la lecture, soit l'écriture : l'architecture ne respecte donc pas les propriétés de « Disponibilité » (*Availability*) et de « Tolérance au partitionnement » (*Partition Tolerance*).

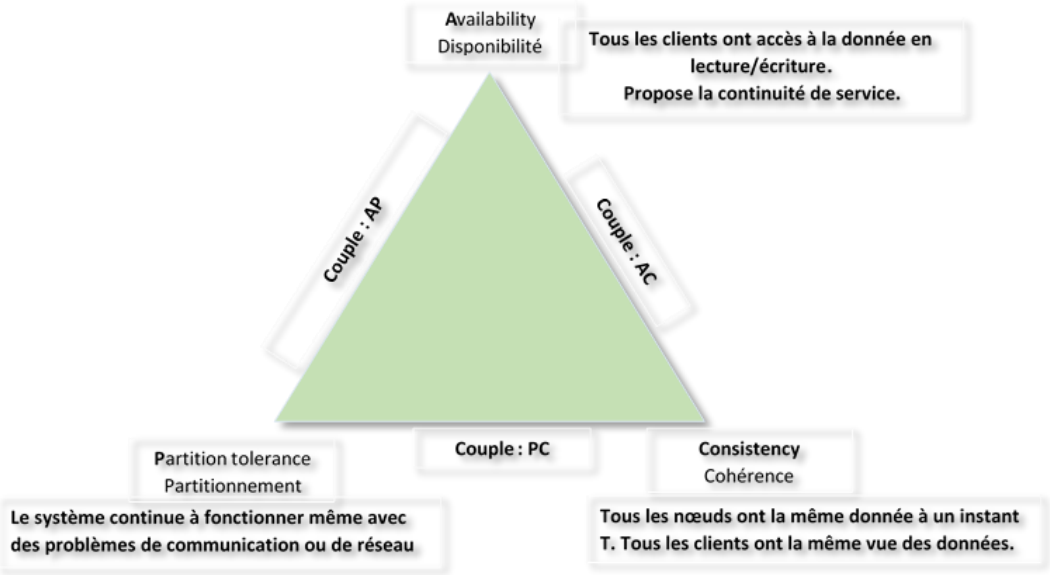
Ces deux propriétés seraient garanties par une duplication bidirectionnelle, avec un accès en lecture/écriture sur les deux serveurs, mais toujours pas la cohérence.

2.2.2 Aucun système n'est parfait : cas général

En prenant le cas général (théorème de CAP), un système distribué ne peut garantir à la fois que deux de ces 3 propriétés :

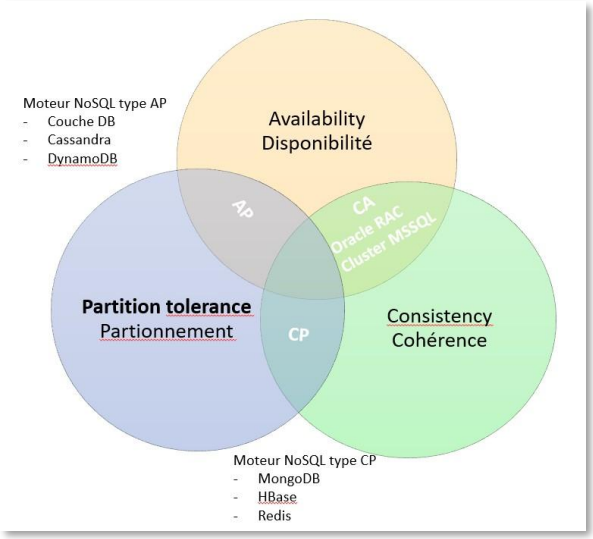
- ***Consistency*** (cohérence) : tous les nœuds voient exactement les mêmes données au même moment
- ***Availability*** (disponibilité) : garantie que toutes les requêtes reçoivent une réponse. La perte de nœuds n'empêche pas les survivants de continuer à fonctionner correctement
- ***Partition tolérance*** (partitionnement) : aucune panne autre que la coupure totale du réseau ne doit empêcher le système de répondre correctement ; en cas de morcellement en sous-réseaux, chacun doit pouvoir fonctionner de manière autonome.

Tout système distribué ne peut garantir qu'un couple d'exigences parmi trois : AP, AC ou PC



2.3 CLASSIFICATION DES SGBD SQL ET NOSQL SELON LE THEOREME DE CAP

Les SGBDR sont en général conformes à AC, les moteurs NoSQL à AP ou PC



2.3.1 AC : Availability, Consistency

Cluster de site unique, donc tous les nœuds sont toujours en contact :

- Le cluster est redondant, donc si un nœud tombe en panne, le cluster continue à fonctionner en lecture/écriture (*Availability*).
- Les nœuds sont synchronisés dans le cluster et renvoient tous la même donnée (*Consistency*).
- Mais lorsqu'une partition du système se produit, suite à un problème réseau, le système s'arrête.

Les SGBDR fournissent des versions en cluster (« grappe ») de serveurs, qui peuvent aussi traiter des *Big Data* (d'après leurs éditeurs !).

Ces clusters privilégient la disponibilité et la cohérence, au détriment de la résistance à la partition du système.

Une référence :

SQL Server 2019 en cluster :
https://docs.microsoft.com/fr-fr/sql/sql-server/what-s-new-in-sql-server-ver15?view=sqlallproducts-allversions

2.3.2 PC : Partition Tolerance, Consistency

Certaines données peuvent ne pas être accessibles, mais les données restent cohérentes, et le système supporte le partitionnement (coupure réseau…).

- **MongoDB :** https://www.mongodb.com/

C'est le SGDB NoSQL que nous allons utiliser dans ce document.

- **hbase :** https://www.lebigdata.fr/apache-hbase-tout-savoir
- **redis :** https://www.w3resource.com/slides/redis-nosql-database-an-introduction.php

2.3.3 AP : Availability, Partition Tolerance

Le système est toujours disponible même en cas de partitionnement, mais certaines données retournées peuvent être inexactes.

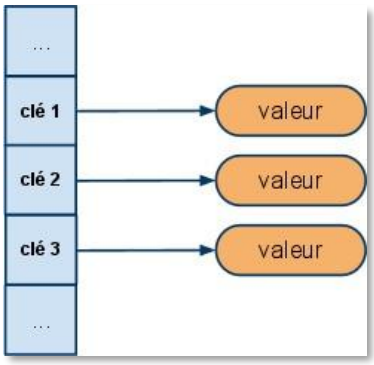
- **cassandra** *:* http://cassandra.apache.org/
- **dynamodb** *:* https://aws.amazon.com/fr/dynamodb/
- **couchdb** *:* http://couchdb.apache.org/

2.4 LES FAMILLES DE NOSQL

Les familles NoSQL ne remplissent pas toutes les mêmes critères, mais elles diffèrent aussi par leur structure et mode d'utilisation : stockage de couple (clé, valeur), base documentaire, base orientée colonnes, base de graphes.

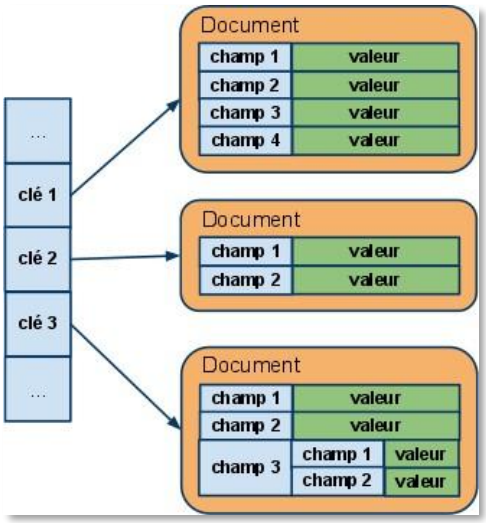
Nous allons décrire les deux premières familles, qui sont répandues et simples à mettre en œuvre.

2.4.1 Couple clé/valeur



- C'est la catégorie de base : chaque objet est identifié par une clé unique qui constitue la seule manière de le requêter.
- On dispose des quatre opérations CRUD (*Create, Read, Update, Delete*) qui sont toutes basées sur la clé de l'objet.
- La base utilise une table de hachage distribuée qui permet de retrouver les objets par leur clé.
- La structure de l'objet est libre et le plus souvent laissé à la charge du développeur (formats *XML, JSON*...).
- Exemples : **dynamodb**, **redis**

2.4.2 Base documentaire



- Les bases de données documentaires sont constituées de collections de document.
- Un document est composé de champs et de leurs valeurs associées : soit des types simples (entier, chaine de caractère, date...), soit des types composés de plusieurs couples clé/valeur.
- La structure des documents ne doit pas être définie à l'avance comme dans une base relationnelle. Elle peut évoluer en cours d'utilisation, ce qui la rend très pratique pour des données complexes et hétérogènes.
- Par rapport au modèle du couple (clé, valeur), le modèle documentaire a l'avantage de permettre des requêtes complexes sur toutes les valeurs (et pas seulement sur la clé de l'objet).
- Le modèle documentaire reste simple, et possède des meilleures performances que le modèle relationnel.
- Exemples : **mongodb**, **couchdb**

2.5 QUE CHOISIR POUR SES DONNEES : SQL OU NOSQL ?

- Il faut choisir un SGBDR avec SQL quand :
  - Les données sont connues à l'avance, liées entre elles, pas trop diversifiées et stables
  - L'intégrité des données est essentielle (gérée par le SGBDR) -
  - Les transactions sont essentielles (sites de vente, de réservation…)

- A l’inverse, NoSQL peut être une bonne solution quand :

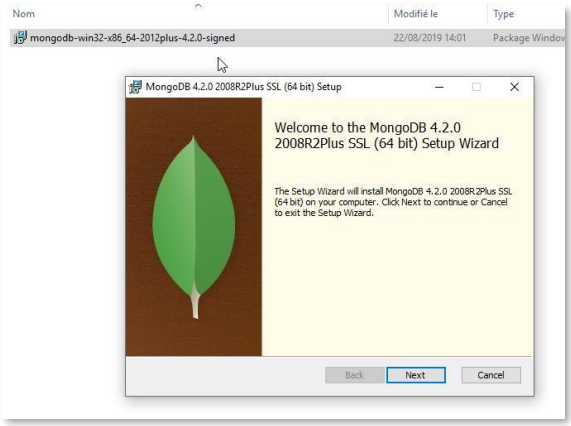
  - Les données sont indépendantes, difficiles à déterminer à l’avance et évolutives
  - La principale caractéristique du projet est l’agilité : il faudra intégrer de nombreux types de données, structurées ou non.
  - Il y a une forte exigence de performance : volume des données, rapidité d’accès, nécessité de distribuer la base sur différents serveurs.

3. METTRE EN PRATIQUE NOSQL AVEC MONGODB

3.1 INSTALLER LE SERVEUR MONGODB

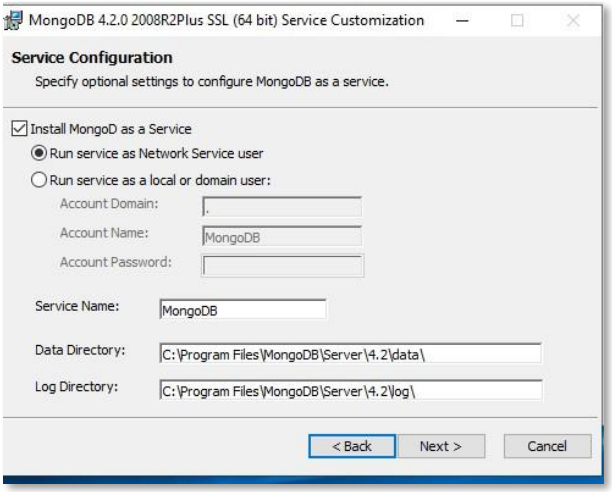
Téléchargez la version gratuite de *MongoDB* sur :
https://www.mongodb.com/download-center/community

- Lancez le fichier d'installation :



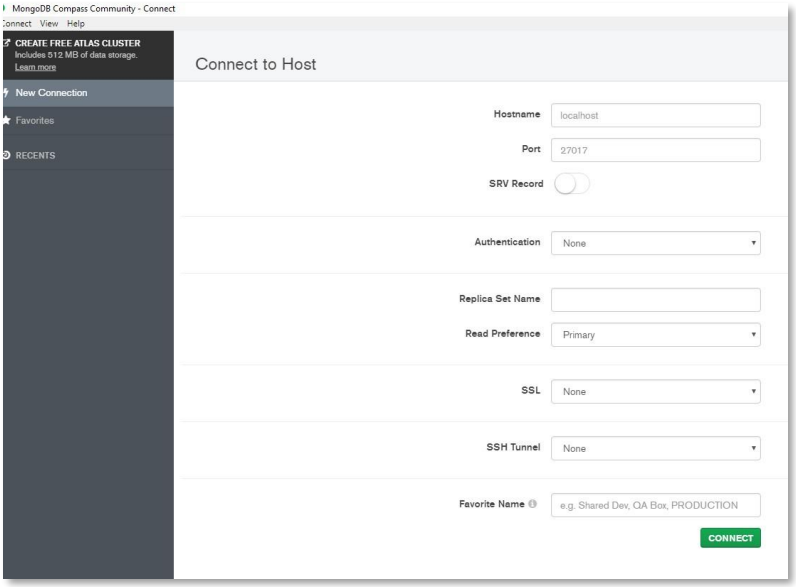
- Choisissez l'installation complète

- Installez *MongoDB* comme un service, avec la configuration standard :



- Installez *MongoDB Compass* qui est l'interface graphique standard de *MongoDB*

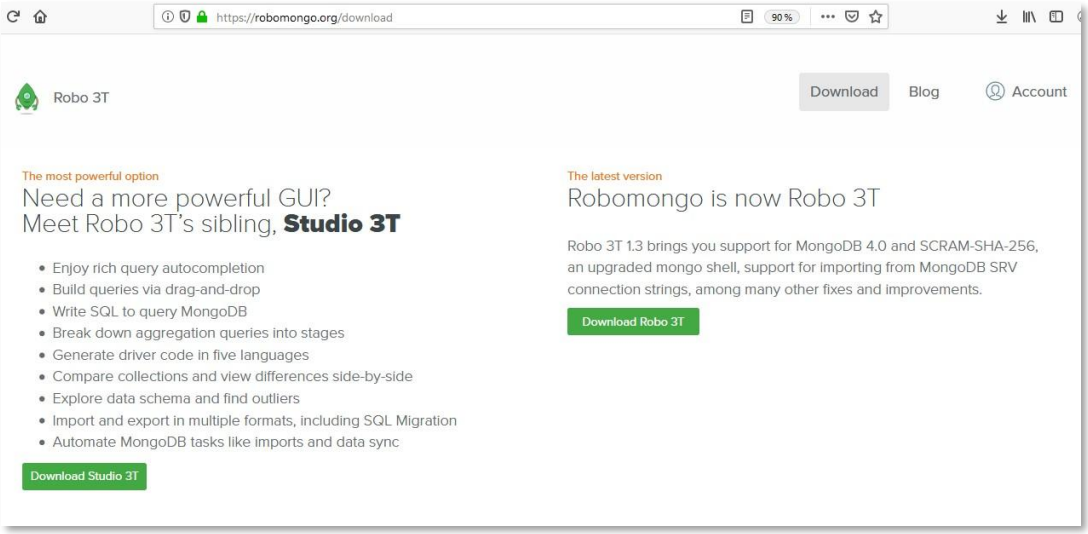
- A la fin de l'installation, *MongoDB Compass* vous propose de créer une connexion par défaut non sécurisée vers l'hôte. Confirmez en cliquant sur *Connect*.



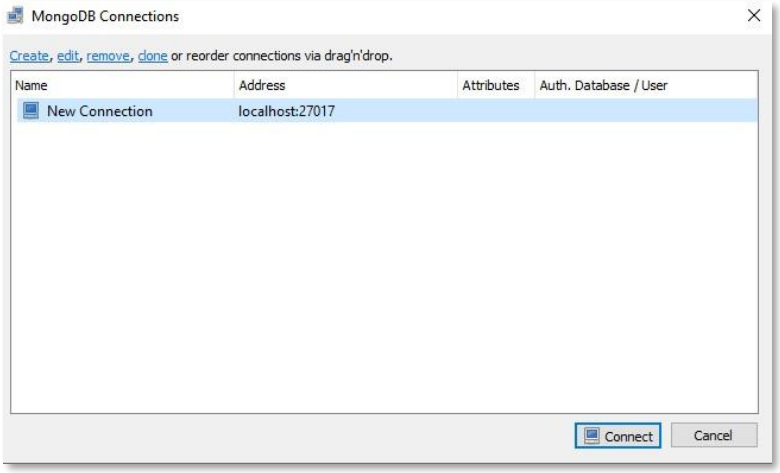
3.2 Installer le client Studio3T (anciennement Robot3T)

- Nous allons effectuer nos requêtes vers MongoDB avec le client.
- Téléchargez la distribution de ***Studio3T***, sur le site :

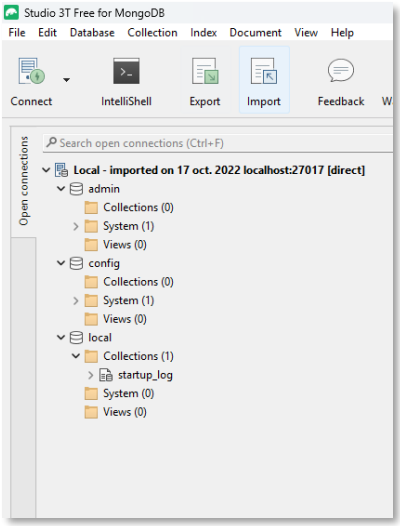
https://robomongo.org/download



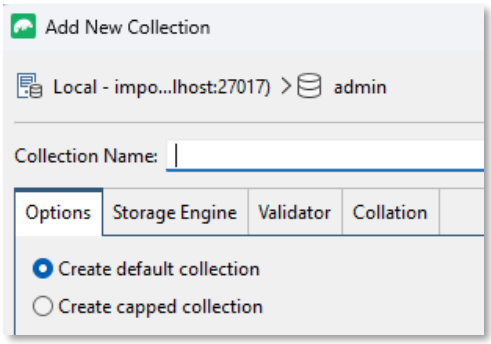
- Choisissez « ***Download Studio3T*** » pour obtenir la version gratuite, téléchargez l'installation pour Windows, et lancez l'exécutable de l'installateur
- A la fin de l'installation, ***Studio3T*** vous propose de vous connecter sur le serveur MongoDB.
- Cliquez sur ***Connect***



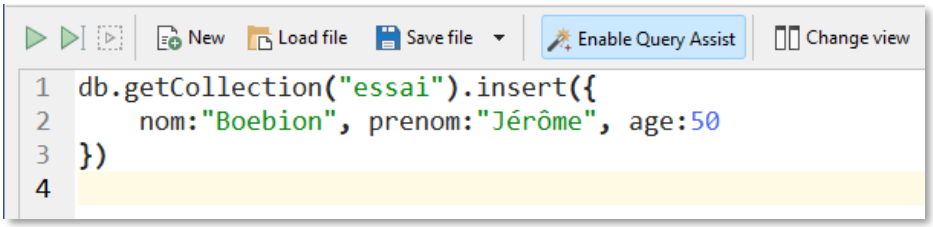
- ***Studio3T*** vous affiche une vue sur les trois bases de données par défaut de MongoDB : **admin**, **local** et **config** avec leurs collections, fonctions et utilisateurs.



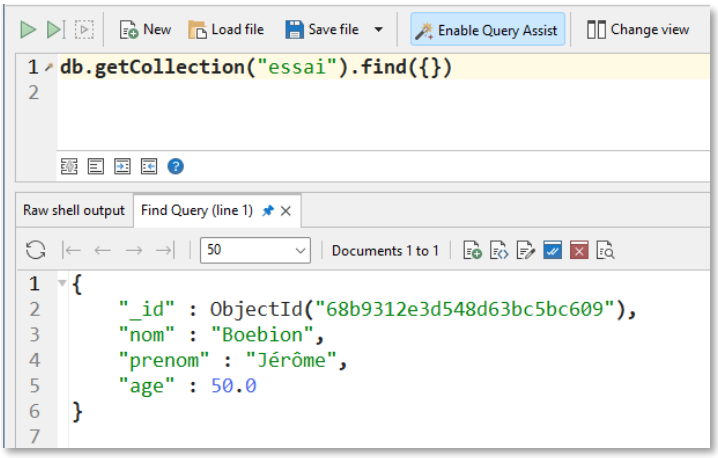
- Pour valider l'outil nous allons créer une collection « *essai* », dans la base *admin*, avec un premier document
- Cliquez avec le bouton droit sur *Collection* -> *Add Collection*



- Cliquez sur la collection « *essai* ». Nous allons insérer une donnée.
  - L'insertion, comme les interrogations, de *MongoBD* va se faire en JSON
- Par exemple, je crée un document avec mon nom, prénom et âge



- Puis exécutez la requête par
- Double-cliquez sur la collection « *essai* » puis pour visualiser ses documents :



- Le document est un objet avec une clé automatique *_id* du type *ObjectId*, qui contient trois autres champs : deux String *nom* et *prenom* et un Int32 *age*
- Votre serveur *MongoDB* est opérationnel

3.3 CREER ET REMPLIR UNE COLLECTION DE TEST

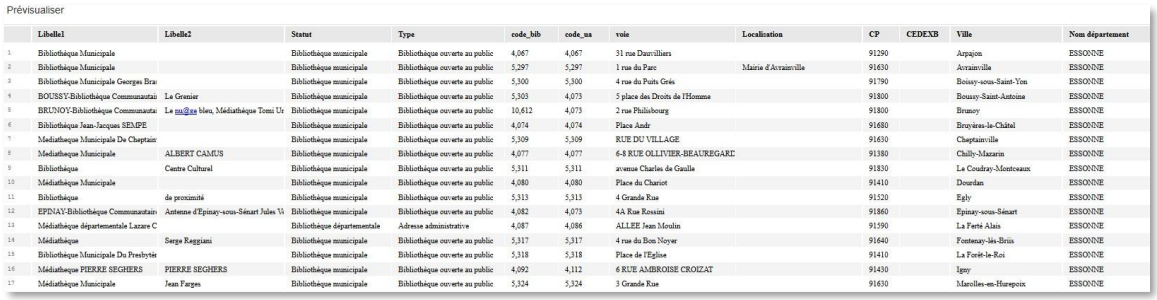
- Pour notre apprentissage de NoSQL, nous allons manipuler des données publiques, qui sont maintenant fournies par les administrations, sur les communes, le cadastre, les bibliothèques, musées etc. : https://www.data.gouv.fr/fr/

- Nous allons nous intéresser aux bibliothèques publiques :

  https://www.data.gouv.fr/fr/datasets/adresses-des-bibliotheques-publiques/

- Le site permet d'exporter les données aux formats CSV, JSON :

- On peut prévisualiser les données :



Prévisualiser

	Libelle1	Libelle2	Statut	Type	code_bib	code_na	voie	Localisation	CP	CEDEXB	Ville	Nom département
1	Bibliothèque Municipale		Bibliothèque municipale	Bibliothèque ouverte au public	4,067	4,067	31 rue Dauvilliers		91290		Arpajon	ESSONNE
2	Bibliothèque Municipale		Bibliothèque municipale	Bibliothèque ouverte au public	5,297	5,297	1 rue du Parc	Mairie d'Avrainville	91630		Avrainville	ESSONNE
3	Bibliothèque Municipale Georges Bra		Bibliothèque municipale	Bibliothèque ouverte au public	5,300	5,300	4 rue du Puits Grés		91790		Boissy-sous-Saint-Yon	ESSONNE
4	BOUSSY-Bibliothèque Communautai	Le Grenier	Bibliothèque municipale	Bibliothèque ouverte au public	5,303	4,073	5 place des Droits de l'Homme		91800		Boussy-Saint-Antoine	ESSONNE
5	BRUNOY-Bibliothèque Communautai	Le m@ze bleu, Médiathèque Tomi Ur	Bibliothèque municipale	Bibliothèque ouverte au public	10,612	4,073	2 rue Philisbourg		91800		Brunoy	ESSONNE
6	Bibliothèque Jean-Jacques SEMPE		Bibliothèque municipale	Bibliothèque ouverte au public	4,074	4,074	Place Andr		91680		Bruyères-le-Châtel	ESSONNE
7	Mediatheque Municipale Du Cheptain		Bibliothèque municipale	Bibliothèque ouverte au public	5,309	5,309	RUE DU VILLAGE		91630		Cheptainville	ESSONNE
8	Mediatheque Municipale	ALBERT CAMUS	Bibliothèque municipale	Bibliothèque ouverte au public	4,077	4,077	6-8 RUE OLLIVIER-BEAUREGARD		91380		Chilly-Mazarin	ESSONNE
9	Bibliothèque	Centre Culturel	Bibliothèque municipale	Bibliothèque ouverte au public	5,311	5,311	avenue Charles de Gaulle		91830		Le Coudray-Montceaux	ESSONNE
10	Médiathèque Municipale		Bibliothèque municipale	Bibliothèque ouverte au public	4,080	4,080	Place du Chariot		91410		Dourdan	ESSONNE
11	Bibliothèque	de proximité	Bibliothèque municipale	Bibliothèque ouverte au public	5,313	5,313	4 Grande Rue		91520		Egly	ESSONNE
12	EPINAY-Bibliothèque Communautair	Antenne d'Epinay-sous-Sénart Jules Vi	Bibliothèque municipale	Bibliothèque ouverte au public	4,082	4,073	4A Rue Rossini		91860		Epinay-sous-Sénart	ESSONNE
13	Médiathèque départementale Lazare C		Bibliothèque départementale	Adresse administrative	4,087	4,086	ALLEE Jean Moulin		91590		La Ferté-Alais	ESSONNE
14	Médiathèque	Serge Reggiani	Bibliothèque municipale	Bibliothèque ouverte au public	5,317	5,317	4 rue du Bon Noyer		91640		Fontenay-lès-Briis	ESSONNE
15	Bibliothèque Municipale Du Presbytèr		Bibliothèque municipale	Bibliothèque ouverte au public	5,318	5,318	Place de l'Eglise		91410		La Forêt-le-Roi	ESSONNE
16	Médiathèque PIERRE SEGHERS	PIERRE SEGHERS	Bibliothèque municipale	Bibliothèque ouverte au public	4,092	4,112	6 RUE AMBROISE CROIZAT		91430		Igny	ESSONNE
17	Médiathèque Municipale	Jean Farges	Bibliothèque municipale	Bibliothèque ouverte au public	5,324	5,324	3 Grande Rue		91630		Marolles-en-Hurepoix	ESSONNE

- Pour une importation dans une base NoSQL, le plus pratique sera le format JSON

Vous utiliserez un fichier *Biblio.json* qui provient de ces données publiques. Il a été adapté pour servir de support à nos exercices.

Chaque objet JSON commence par **{** et comprend :

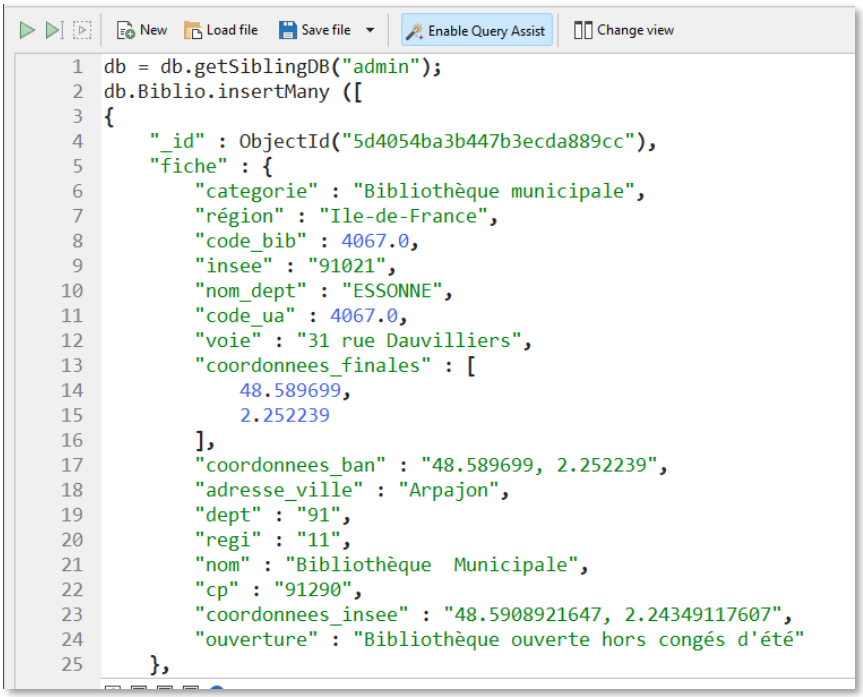
- un attribut ***_id*** qui est la clé de l'objet, du type *ObjectID*
- un attribut ***fiche***, de type objet, qui décrit la bibliothèque : ***catégorie***, ***nom***, ***nom_dept*** etc.

Ces données sont issues des données publiques. Certains noms d'attributs ont été modifiés pour clarifier. L'attribut ***ouverture*** a été ajouté.

- un attribut ***notation***, de type objet, qui a été ajouté pour les besoins de la série d'exercices. Il comprend une appréciation globale sur la bibliothèque (***appreciation***) donnée par l'administration, et un tableau ***notes*** : série de notes entre 1 et 5 donnée par le public.

```
1  db = db.getSiblingDB("admin");
2  db.Biblio.insertMany ([
3  {
4      "_id" : ObjectId("5d4054ba3b447b3ecda889cc"),
5      "fiche" : {
6          "categorie" : "Bibliothèque municipale",
7          "région" : "Ile-de-France",
8          "code_bib" : 4067.0,
9          "insee" : "91021",
10         "nom_dept" : "ESSONNE",
11         "code_ua" : 4067.0,
12         "voie" : "31 rue Dauvilliers",
13         "coordonnees_finales" : [
14             48.589699,
15             2.252239
16         ],
17         "coordonnees_ban" : "48.589699, 2.252239",
18         "adresse_ville" : "Arpajon",
19         "dept" : "91",
20         "regi" : "11",
21         "nom" : "Bibliothèque  Municipale",
22         "cp" : "91290",
23         "coordonnees_insee" : "48.5908921647, 2.24349117607",
24         "ouverture" : "Bibliothèque ouverte hors congés d'été"
25     },
26     "notation" : {
27         "appreciation" : "ok",
28         "notes" : [
29             5.0,
30             4.0,
31             5.0
32         ]
33     },
34     "record_timestamp" : "2018-06-08T15:12:13+02:00"
35 }
```

- Les données ne sont pas homogènes : beaucoup de bibliothèques n'ont pas encore de notation. Certaines ont des données supplémentaires : l'attribut ***local*** qui précise le lieu.
- Créez comme dans l'essai précédent une collection « *biblio* »
- Puis charger le fichier en cliquant sur Load file
- Toutes les requêtes vers MongoDB utilisent des paramètres en JSON : ici on passe une collection d'objets (commençant par **[**) à la méthode *insertMany* sur la collection *biblio.*



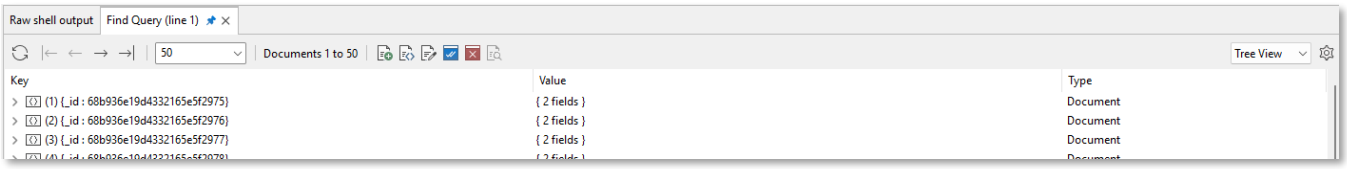
- Lancez la requête en cliquant sur la flèche verte puis attendre la fin de l'opération

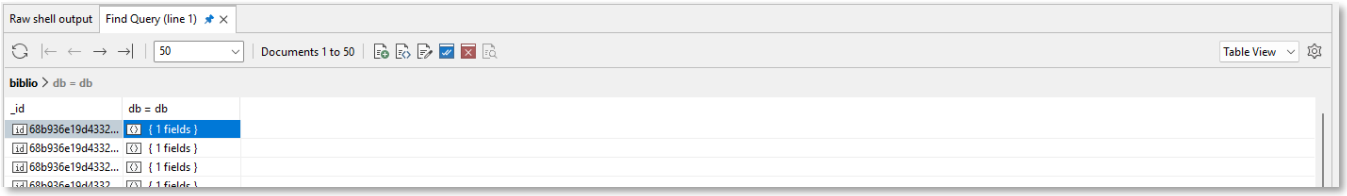
Note du formateur : lors de l'essai avec Studio3T, j'ai constaté une lenteur d'exécution. Si cela vous arrive, pour insérer les données, j'ai utilisé MongoDB Compass

- Rafraichir la base admin pour voir apparaitre biblio et sélectionner la
- Clliquer sur Add Data >> import JSON or CSV File
- Sélectionner le délimiter par Semicolon
- Puis import
- Retourner sur Studio3T

- Vérifiez que votre collection a été remplie, en cliquant à nouveau sur *biblio*

- Essayez les différents modes d'affichage qui seront pratiques pour parcourir les données :
  - en arbre montrant les objets imbriqués, en table et en JSON.





3.4 INTERROGER *MONGODB* EN *JAVASCRIPT* DEPUIS LE CLIENT Studio3T

Une correction sera donnée après la réalisation de ce TP.

- Nous allons utiliser cette base de test pour découvrir les possibilités de NoSQL, comme nous l'avions fait en SQL :
  - d'abord des interrogations simples
  - puis des requêtes avec regroupement (équivalent du GROUP BY en SQL)
  - des insertions, modifications, suppressions
  - des requêtes portant sur plusieurs collections (comme les jointures en SQL).

- La connaissance du langage SQL n'est en théorie pas nécessaire, mais cela aidera souvent comme point de comparaison !
- Toutes les requêtes vont manipuler des objets en JSON : pour les paramètres et pour les valeurs retournées.
- Les requêtes dans le client interactif ***Studio3T*** sont écrites en *JavaScript* : nous pourrons utiliser du code *JavaScript* pour enchaîner nos requêtes et organiser nos traitements. Pour chaque exercice, nous renverrons à la documentation en ligne de *MongoDB* qui contient de nombreux exemples :

  https://docs.mongodb.com/manual/reference/

- Pour chaque point :
  - Exécutez les codes d'exemple (en bleu)
  - Consultez la documentation, en suivant les liens
  - Réalisez et testez les exercices proposés

3.4.1 Interrogations simples

Lister toutes les bibliothèques avec toutes leurs caractéristiques (c'est-à-dire tous les objets de la collection biblio*)*

```
db.getCollection("Biblio").find({});
```

La méthode *find* sans paramètre retourne toute la collection.

Pour mémoire, en SQL : select * from biblio

Dans les exemples suivants, nous allons le plus souvent abandonner SQL et « penser en JSON ».

https://docs.mongodb.com/manual/reference/method/db.collection.find/

Récupérer le nombre de bibliothèques :

```
db.getCollection("Biblio").count();
db.getCollection("Biblio").find().count();
```

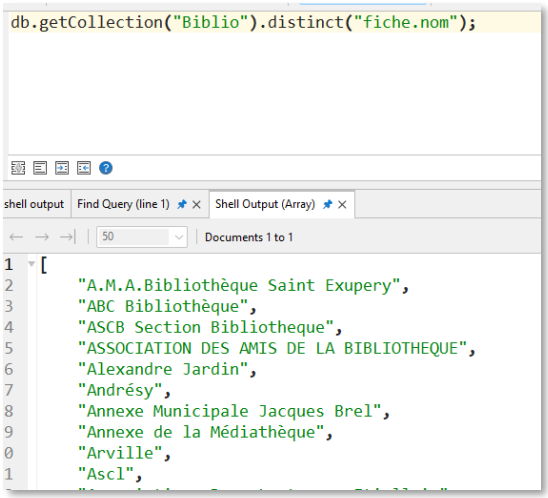
Le *find* retourne une collection sur laquelle on peut faire un *count*.

Lister les noms de toutes les bibliothèques, sans doublon

```
db.getCollection("Biblio").distinct("fiche.nom");
```

Remarquez la notation objet : le *nom* est un attribut de l'objet *fiche.*

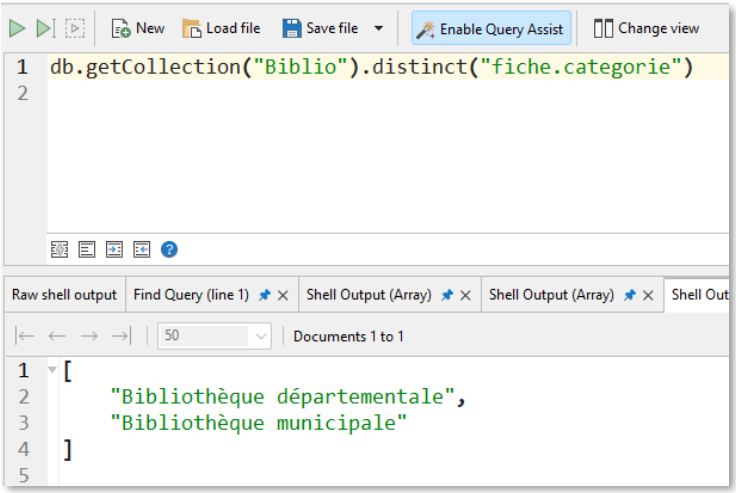
Comme son nom l'indique, le *distinct* n'affiche que des objets distincts, en les listant à « plat » dans un tableau JSON, sans les structures qui les contiennent :



Lister toutes les catégories, sans doublon

```
db.getCollection("Biblio").distinct("fiche.categorie");
```

La méthode ***distinct*** est pratique pour explorer une base NoSQL et visualiser des informations utiles, afin de bien construire ses requêtes.



`db.collection.find(query, projection, options)`

query : Optionnel - Spécifie le filtre de sélection en utilisant les opérateurs de requête.

- Pour retourner tous les documents d'une collection, omettez ce paramètre ou passez un document vide ({}).

projection Optionnel - Spécifie les champs à retourner dans les documents qui correspondent au filtre de requête.

- Pour retourner tous les champs des documents correspondants, omettre ce paramètre.

Options : Optionnel. Spécifie des options supplémentaires pour la requête.

- Ces options modifient le comportement de la requête et la façon dont les résultats sont retournés.

3.4.2 Projection

Lister toutes les bibliothèques, en affichant uniquement les champs : categorie, nom, région (sans l'id de l'objet) :

```
db.getCollection("Biblio").find(
    {},
    {"fiche.categorie":1, "fiche.nom":1, "fiche.région":1, "_id":0 }
)
```

La méthode *find* a deux paramètres, de type objet.

- Le deuxième paramètre liste les attributs que l'on veut afficher : c'est une « projection », l'équivalent des colonnes derrière un SELECT en SQL.
- Pour afficher un attribut, on le déclare à 1 (catégorie, nom et région).
- Pour exclure un attribut, on le déclare à 0 (_id)

```
db.getCollection("Biblio").find(
    {},
    {"fiche.categorie":1, "fiche.nom":1, "fiche.région":1, "_id":0 }
);
```

shell output | Find Query (line 1) | Shell Output (Array) | Shell Output (Array) | Shell Output (Array) | Find C

50 | Documents 1 to 50

```
1 {
2     "fiche" : {
3         "categorie" : "Bibliothèque municipale",
4         "région" : "Ile-de-France",
5         "nom" : "Bibliothèque  Municipale"
6     }
7 }
```

Exercice 1 : *Lister toutes les bibliothèques, en affichant uniquement leurs nom, ville et notation (appréciation et notes), sans l'id de l'objet*

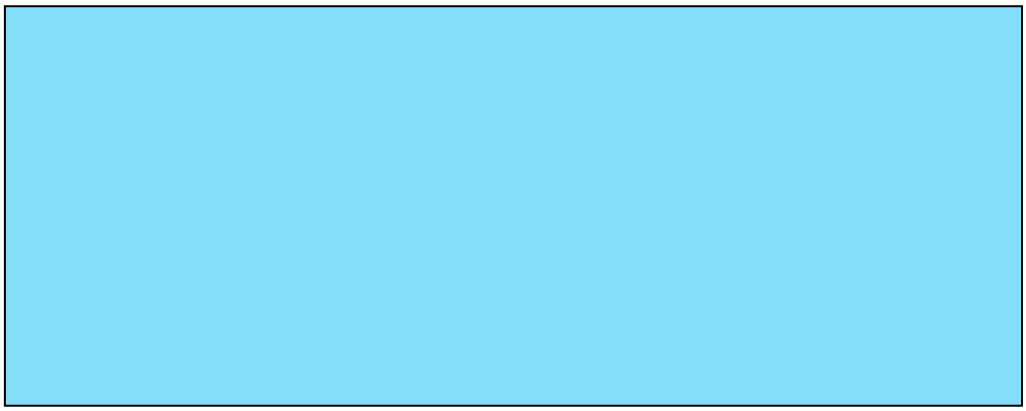
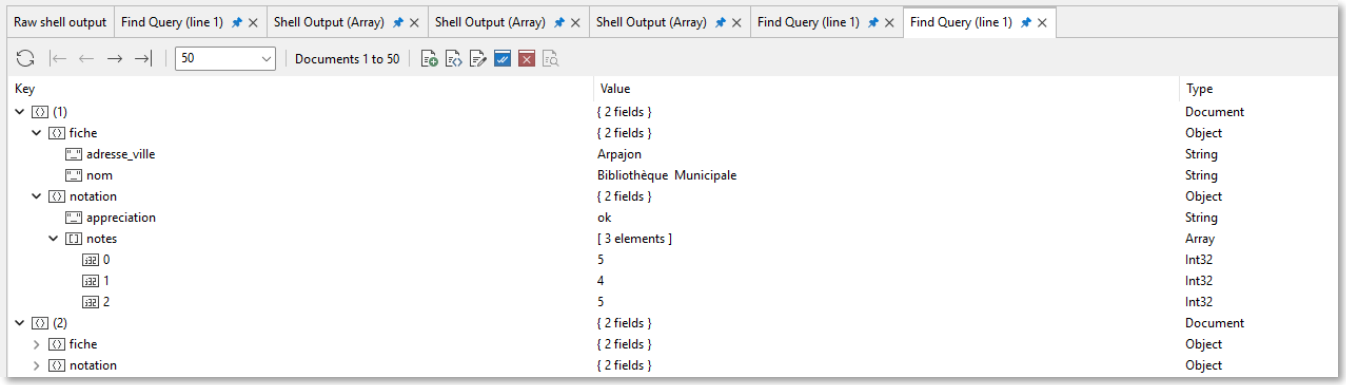


FIGURE 1 : REQUETE DE L'EXERCICE 1



Raw shell output | Find Query (line 1) | Shell Output (Array) | Shell Output (Array) | Shell Output (Array) | Find Query (line 1) | Find Query (line 1)

50 Documents 1 to 50

Key	Value	Type
(1)	{ 2 fields }	Document
fiche	{ 2 fields }	Object
adresse_ville	Arpajon	String
nom	Bibliothèque Municipale	String
notation	{ 2 fields }	Object
appreciation	ok	String
notes	[3 elements]	Array
0	5	Int32
1	4	Int32
2	5	Int32
(2)	{ 2 fields }	Document
fiche	{ 2 fields }	Object
notation	{ 2 fields }	Object

FIGURE 2 : CAPTURE DU RÉSULTAT

3.4.3 Filtre

Lister uniquement les bibliothèques du 91, qui sont ouvertes au public

```
db.biblio.find (
    {
        "fiche.dept":"91",
        "fiche.ouverture":"Bibliothèque ouverte au public"
    }
)
```

- La méthode *find* a deux paramètres, de type objet (ici, le deuxième est omis).
- Le premier paramètre liste les critères de sélection : c'est un « filtre », l'équivalent d'une clause WHERE en SQL.

Exercice 2 : *Lister uniquement les bibliothèques municipales, qui ont reçu une appréciation*

Pour tester l'existence d'un attribut, utilisez l'opérateur *$exists*

avec une valeur 1 (ou 0 pour l'inexistence) : *nom_attribut :{$exists :1}*

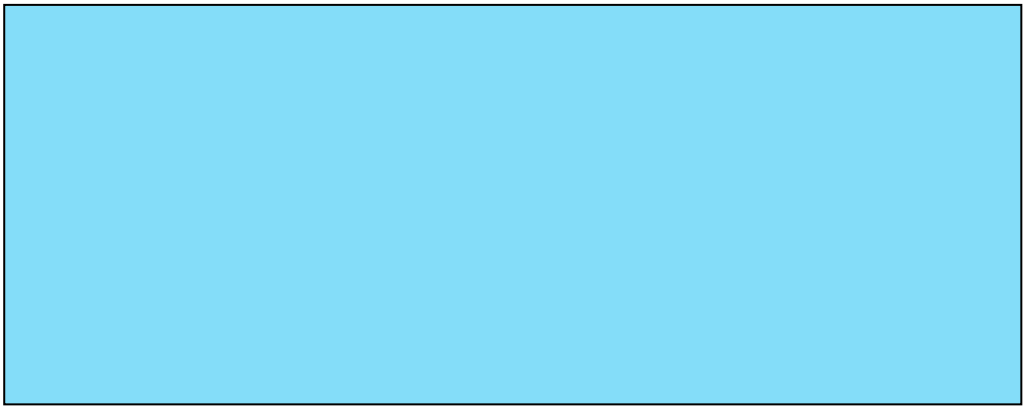


FIGURE 3 : REQUÊTE DE L'EXERCICE 2

FIGURE 4 : CAPTURE RÉSULTAT A OBTENIR [49 OCCURENCES]

3.4.4 Filtre et Projection

Filtrer les bibliothèques du 91 qui ne sont pas ouvertes au public, en affichant leur nom, ville, et le motif de non ouverture (sans l'id de l'objet) :

```
db.biblio.find(
  {
    "fiche.dept":"91",
    "fiche.ouverture":{$ne:"Bibliothèque ouverte au public"}
  },
  {
    "fiche.nom":1,
    "fiche.adresse_ville":1,
    "fiche.ouverture":1,
    "_id":0
  })
```

Notez l'utilisation de l'opérateur *not equal* : *$ne*

- Le premier paramètre du *find* est un objet JSON, qui contient une liste de critères (sous la forme clé : valeur), séparés par des virgules.
- Chaque valeur est soit une constante (**"91"**), soit elle-même un objet composé d'un opérateur et d'une valeur : **{$ne:"Bibliothèque ouverte au public"}**

Les opérateurs de MongoDB (« *Query Selectors* ») sont précédés d'un $, qui permet à l'interpréteur de les distinguer des constantes :

https://docs.mongodb.com/manual/reference/operator/query/#query-selectors

Raw shell output | Find Query (line 1) | Find Query (line 1) | Find Query (line 1) | Find Query (line 1) | Find Query (line 1) | Find Query (line 1) | Find Query

50 | Documents 1 to 1

Key	Value
adresse_ville	Arpajon
nom	Bibliothèque Municipale
ouverture	Bibliothèque ouverte hors congés d'été
1	{ 1 fields }
fiche	{ 3 fields }
adresse_ville	Avrainville
nom	Bibliothèque Municipale
ouverture	Accès réservé adhérents
2	{ 1 fields }
fiche	{ 3 fields }
adresse_ville	Boissy-sous-Saint-Yon
nom	Bibliothèque Municipale Georges Brassens
ouverture	Fermeture provisoire
3	{ 1 fields }

Exercice 3 : *Lister les bibliothèques départementales qui ont reçu une appréciation "ok", en affichant uniquement leur nom, catégorie, ville et appréciation, sans l'id de l'objet*

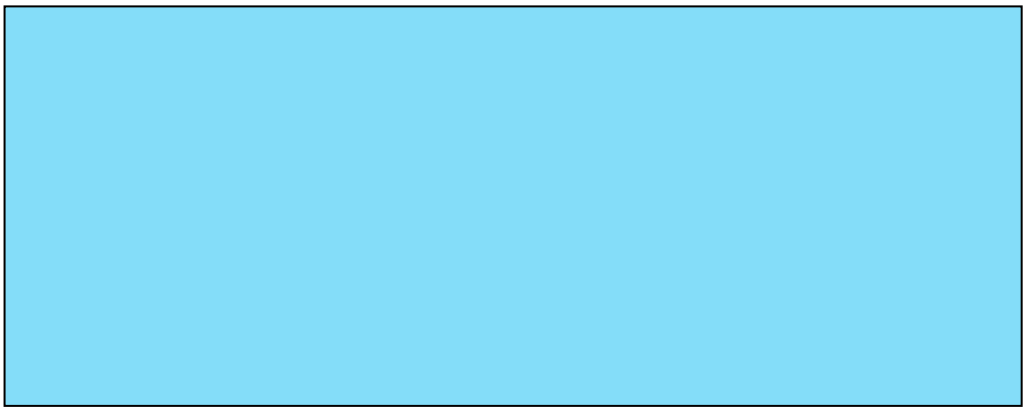


FIGURE 5 : REQUÊTE DE L'EXERCICE 3

```
[
    {
        "fiche" : {
            "categorie" : "Bibliothèque départementale",
            "adresse_ville" : "La Ferté Alais",
            "nom" : "Médiathèque départementale Lazare Carnot (Annexe De La Bdp)"
        },
        "notation" : {
            "appreciation" : "ok"
        }
    }
]
```

FIGURE 6 : (1 SEULE BIBLIOTHÈQUE)

3.4.5 Filtrer avec des expressions régulières

Rechercher les médiathèques (bibliothèques dont le nom contient le mot « Médiathèque » en majuscule ou minuscule)

```
db.getCollection("Biblio").find({"fiche.nom":/Médiathèque/i})
```

- Syntaxe JavaScript : les expressions régulières sont entourées par des /

Pour indiquer que le nom doit commencer par « Médiathèque » :

```
db.getCollection("Biblio").find({"fiche.nom":/^Médiathèque/i})
```

Le nom se termine par « Médiathèque » :

```
db.getCollection("Biblio").find({"fiche.nom":/Médiathèque$/i})
```

Le nom est égal à « Médiathèque » :

```
db.getCollection("Biblio").find({"fiche.nom":/^Médiathèque$/i})
```

Autre syntaxe avec l'opérateur *$regex* :

```
db.getCollection("Biblio").find({"fiche.nom":{$regex :/Médiathèque/i} })
```

https://docs.mongodb.com/manual/reference/operator/query/regex/#behavior

Exercice 4 : *Lister les bibliothèques du 95 dont la ville commence par « Arr », en affichant uniquement le département et la ville*

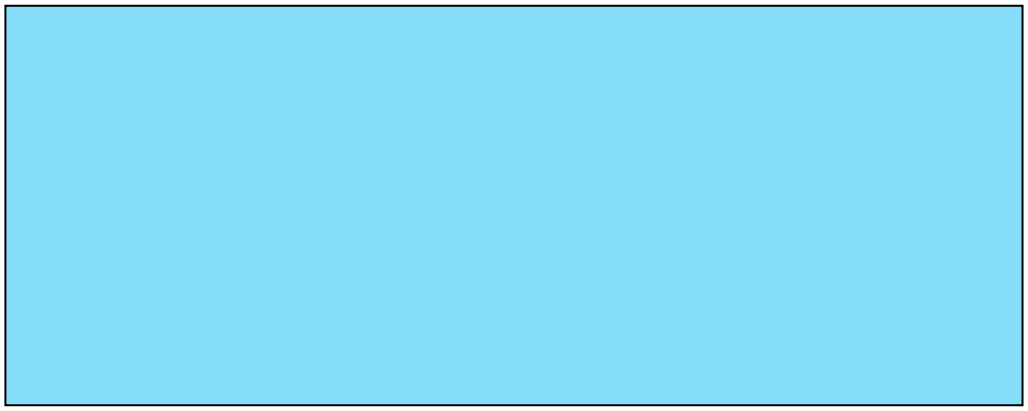


FIGURE 7 : REQUETE DE L'EXERCIE 4

```
[
    {
        "fiche" : {
            "adresse_ville" : "Arronville",
            "dept" : "95"
        }
    }
]
```

FIGURE 8 : 1 BIBLIOTHÈQUE

Exercice 5 : *Lister les bibliothèques dont la ville commence par "B", contient au moins un "o", et se termine par "le", en affichant uniquement la ville*

Dans une expression régulière, **.*** désigne une suite de caractères quelconques

FIGURE 9 : REQUÊTE DE L'EXERCICE 5

```
[
    {
        "fiche" : {
            "adresse_ville" : "Bondoufle"
        }
    }
]
```

FIGURE 10 : 1 SEULE RÉPONSE

3.4.6 Filtrer avec des opérateurs

Lister les bibliothèques qui ont reçu au moins une note inférieure ou égale à 1, avec leur nom, leur ville, leur appréciation et leurs notes

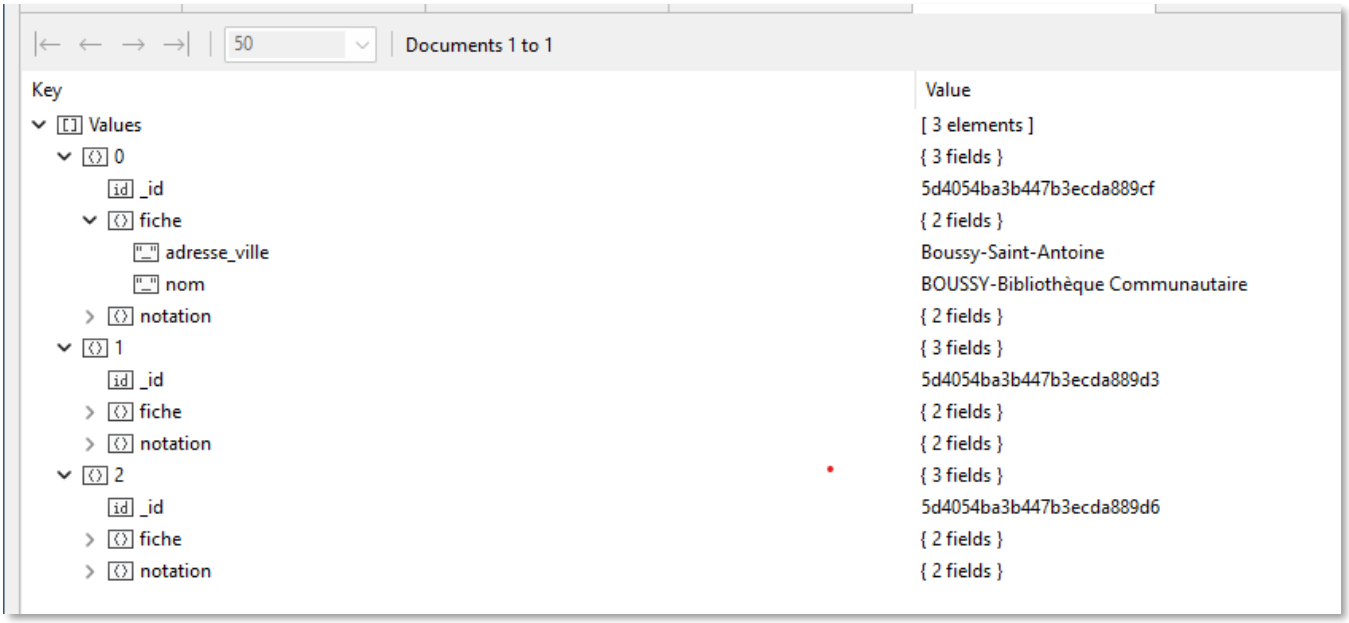
```
db.getCollection("Biblio").find(
  {
    "notation.notes": { $lte:1 }
  },
  {
    "fiche.adresse_ville":1,
    "fiche.nom":1,
    "notation":1
  })
```

MongoDB fournit des opérateurs de comparaison *($lte, $lt, $gt* etc.) et des opérateurs logiques *($or, $not, $and* etc.) qui permettent de combiner les tests. https://docs.mongodb.com/manual/reference/operator/query/#query-selectors

Ces opérateurs peuvent être utilisés sur des attributs simples ou composés, comme des tableaux.

- Dans le deuxième cas, ils fonctionnent comme des opérateurs ensemblistes en SQL (*EXISTS*, *ANY*, *ALL*, *IN*).

Dans l'exemple ci-dessus, *$lte* est appliqué au tableau des notes : ce qui signifie qu'il existe au moins une note inférieure ou égale à 1.



Exercice 6 : *Lister les bibliothèques qui n'ont aucune note strictement supérieure à 3 (y compris celles qui n'ont pas de notes), avec leur nom, ville, appréciation et notes*

FIGURE 11 : REQUÊTES DE L'EXERCICE 6

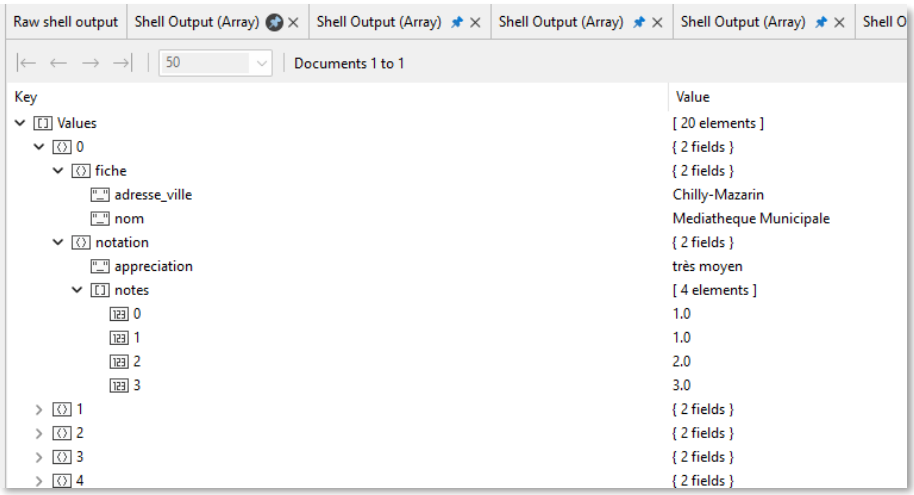


FIGURE 12 : 970 RÉSULTATS

Exercice 7 : ***Lister le nombre de bibliothèques qui ont été notées (50 bibliothèques) Et le nombre de bibliothèques sans aucune note (956 bibliothèques)***

FIGURE 13 : REQUÊTE 1

FIGURE 14 : REQUÊTE 2

Exercice 8 : *Lister les bibliothèques qui n'ont aucune note strictement supérieure à 3 avec leur nom, ville et notes, en limitant la recherche aux bibliothèques qui ont au moins une note*

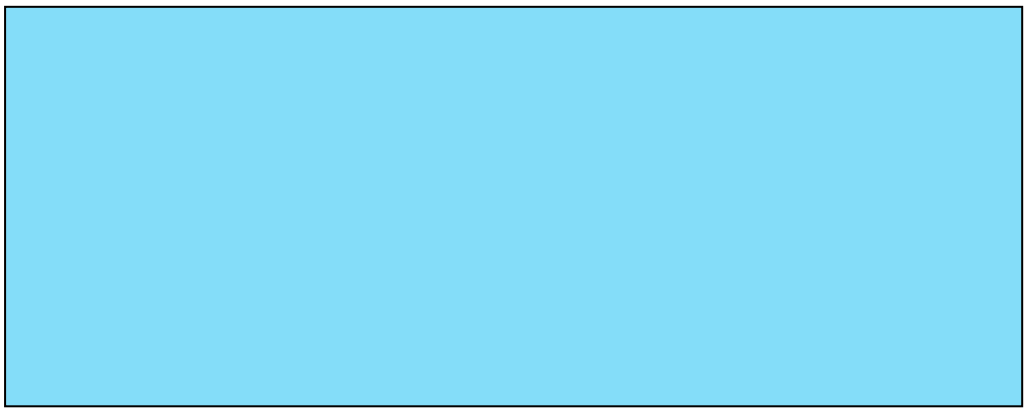


FIGURE 15 : REQUÊTE DE L'EXERCICE 8

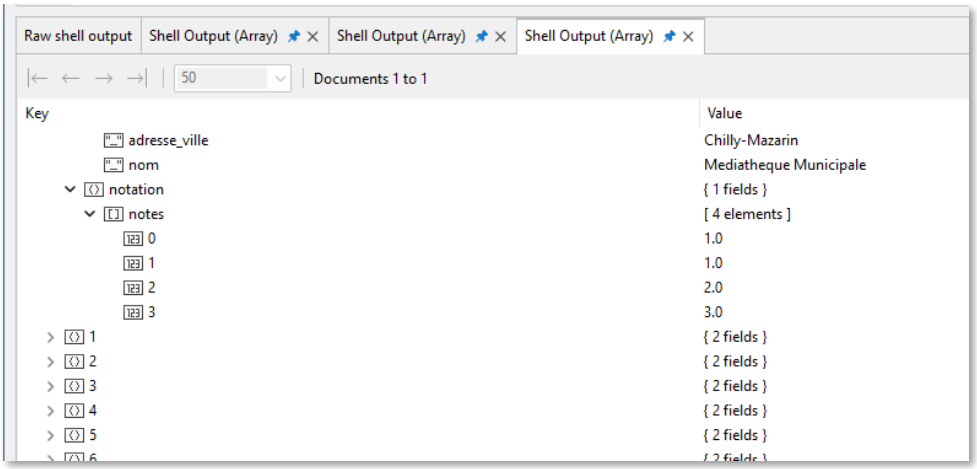


FIGURE 16 : 14 RÉSULTATS

Exercice 9 : *Lister les bibliothèques qui ont au moins une note strictement inférieure à 2 et aucune note strictement supérieure à 3*

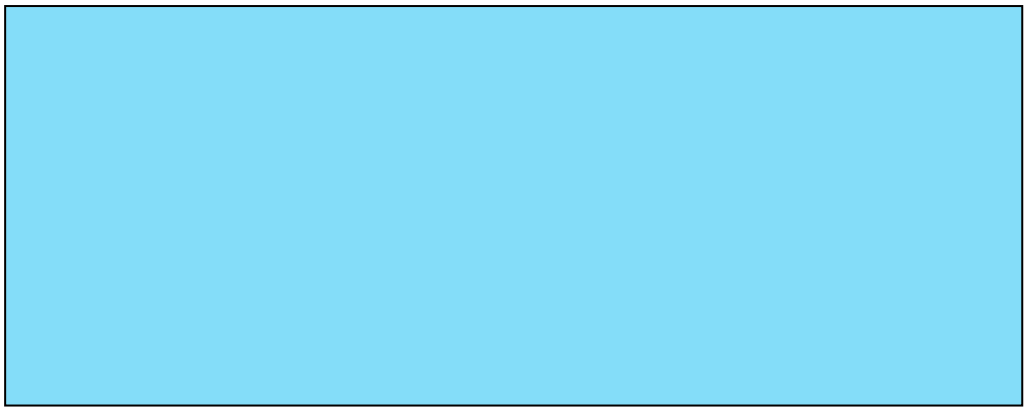


FIGURE 17 : REQUÊTE DE L'EXERCICE 9

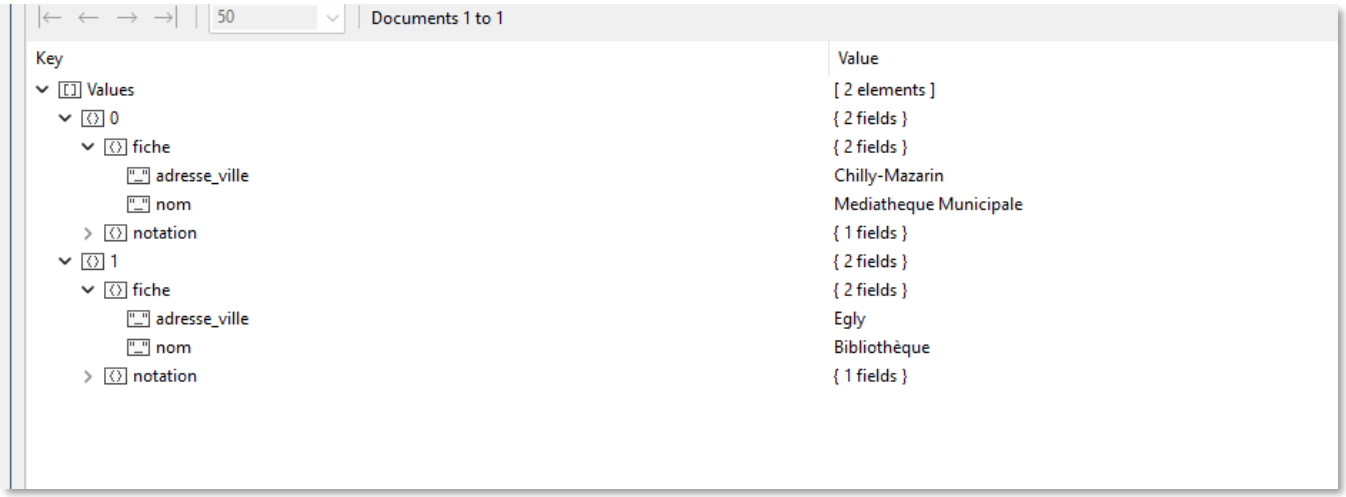


FIGURE 18 : 2 RÉSULTATS

Exercice 10 : *La ville, le nom, l'appréciation et les notes des bibliothèques qui ont une appréciation "moyen" ou "très moyen"*

(utilisez l'opérateur $in pour tester l'appartenance de l'appréciation à une de ces catégories)

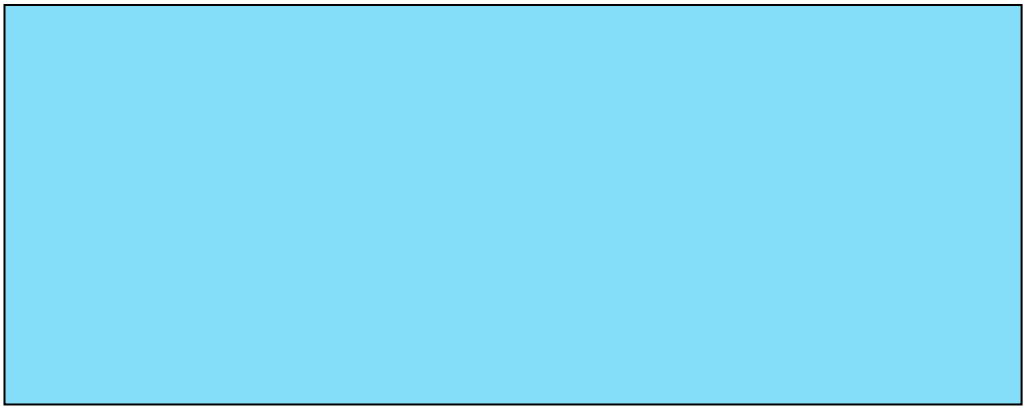


FIGURE 19 : REQUÊTE DE L'EXERCICE 10

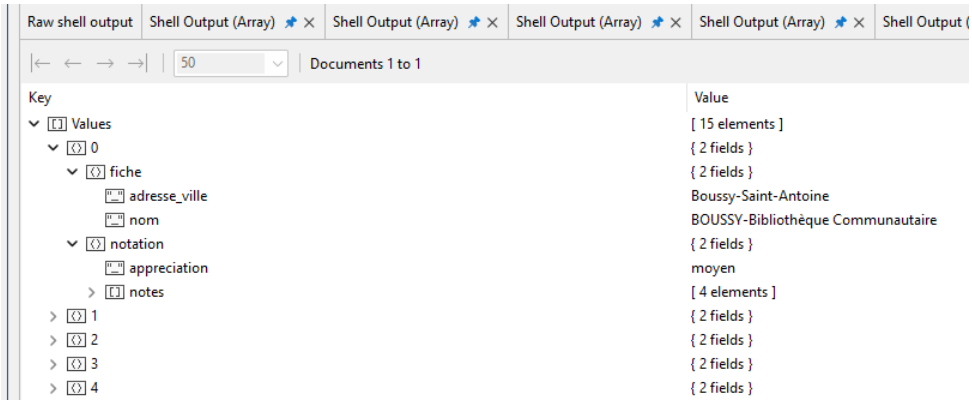


FIGURE 20 : 15 RÉSULTATS

Exercice 11 : *La ville, le nom et les notes des bibliothèques dont la plus ancienne note est 1*

*(c'est-à-dire le premier élément du tableau notes, "*notation.notes.**0**")

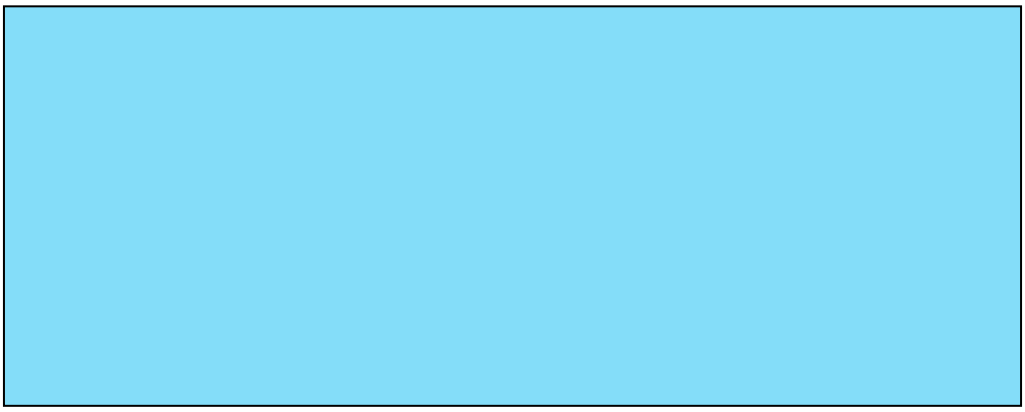


FIGURE 21 : REQUÊTE DE L'EXERCICE 11

Key	Value
Values	[3 elements]
0	{ 2 fields }
fiche	{ 2 fields }
adresse_ville	Boussy-Saint-Antoine
nom	BOUSSY-Bibliothèque Communautaire
notation	{ 1 fields }
1	{ 2 fields }
fiche	{ 2 fields }
adresse_ville	Chilly-Mazarin
nom	Mediatheque Municipale
notation	{ 1 fields }
2	{ 2 fields }
fiche	{ 2 fields }
adresse_ville	Egly
nom	Bibliothèque
notation	{ 1 fields }

FIGURE 22 : 3 RÉSULTATS

Exercice 12 : *La ville, le nom et les notes des bibliothèques qui ont une appréciation "ok" et exactement trois notes, dans l'ordre : 5, 5, 4*

(on peut comparer un tableau à une liste de constantes : notes:[5,5,4])

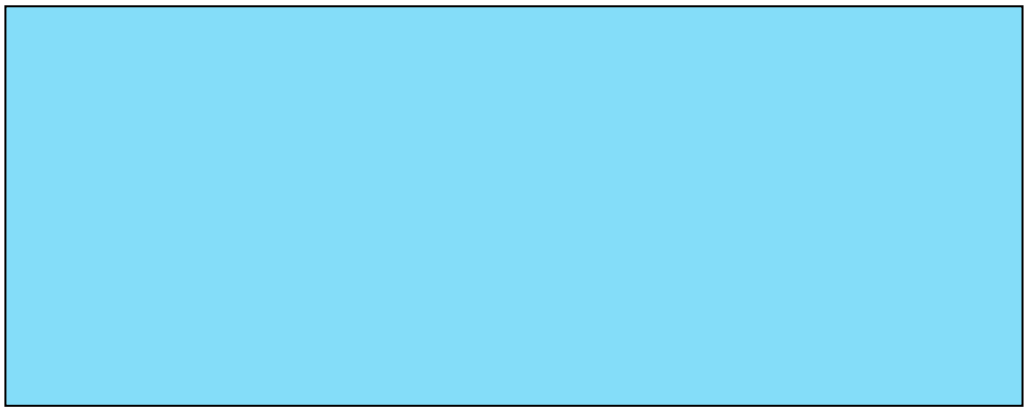


FIGURE 23 : REQUÊTE DE L'EXERCICE 12

Key	Value
Values	[1 elements]
0	{ 2 fields }
fiche	{ 2 fields }
adresse_ville	Boissy-sous-Saint-Yon
nom	Bibliothèque Municipale Georges Brassens
notation	{ 1 fields }
notes	[3 elements]

FIGURE 24 : 1 RÉSULTAT

3.4.7 Le pipeline d'aggrégation : aggregate

La syntaxe de MongoDB permet d'enchaîner plusieurs opérations, dans un « pipeline » avec la méthode *aggregate* :

- des filtres (par *$match*),
- des tris (par *$sort*),
- des projections en choisissant les colonnes à afficher (par *$project*),
- des groupements (équivalents du GROUP BY en SQL, par un $*group*).

https://docs.mongodb.com/manual/reference/method/db.collection.aggregate/

Par exemple, l'exercice précédent « *Chercher les bibliothèques dont la plus ancienne note est 1, en n'affichant que la ville, le nom et les notes* ») peut se décomposer en un filtre (comme le

WHERE en SQL) et une projection (comme le SELECT du SQL)

```
db.biblio.aggregate (
[
   {$match : {"notation.notes.0":1} },
   {
    $project : {
    "fiche.adresse_ville":1,
    "fiche.nom":1,
    "notation.notes":1, _id : 0 }
   }
]
)
```

A noter : la liste des opérations à effectuer dans l'ordre, est passée à la méthode *aggregate* dans un tableau.

Key	Value
Values	[3 elements]
0	{ 2 fields }
fiche	{ 2 fields }
notation	{ 1 fields }
1	{ 2 fields }
fiche	{ 2 fields }
notation	{ 1 fields }
2	{ 2 fields }
fiche	{ 2 fields }
notation	{ 1 fields }

Exercice 13 : *Réalisez le même filtre sur la première note, en triant les bibliothèques par ordre croissant des villes, et en n'affichant que les noms de la ville et de la bibliothèque.*

Pour trier, ajouter un $sort dans le tableau, après le $match : { $sort : {"fiche.adresse_ville":1} }

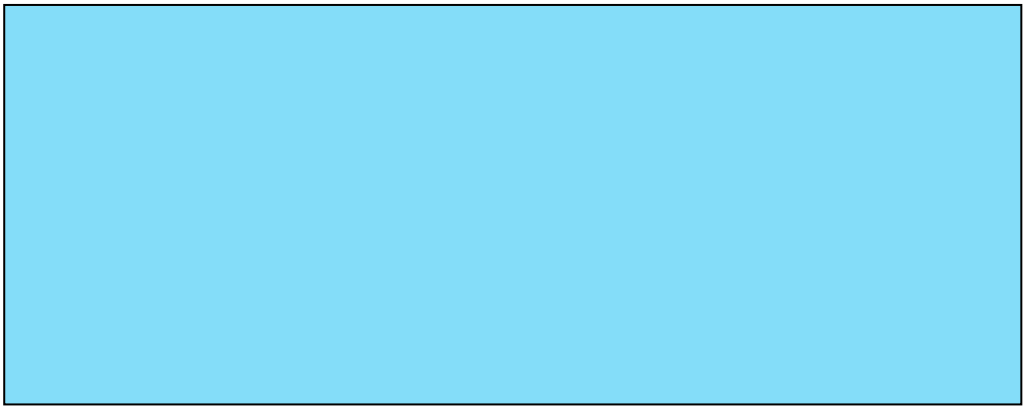


FIGURE 25 : REQUÊTE DE L'EXERCICE 13

Key	Value
Values	[3 elements]
0	{ 1 fields }
fiche	{ 2 fields }
adresse_ville	Boussy-Saint-Antoine
nom	BOUSSY-Bibliothèque Communautaire
1	{ 1 fields }
fiche	{ 2 fields }
adresse_ville	Chilly-Mazarin
nom	Mediatheque Municipale
2	{ 1 fields }
fiche	{ 2 fields }
adresse_ville	Egly
nom	Bibliothèque

FIGURE 26 : 3 RÉSULTATS

3.4.8 Utiliser des variables JavaScript pour simplifier le code des requêtes

La syntaxe des requêtes en JSON devient vite lourde, en particulier dans une méthode *aggregate*.

Comme les requêtes sont écrites en JavaScript, on peut utiliser des variables intermédiaires JavaScript pour simplifier la syntaxe.

Utilisation de variables en Javascript :

```
const var = {$match : { "notation.notes.0":1} };
db.Biblio.aggregate( [ varMatch, varProject, varSort ] )
```

Exercice 14 : *Refaites l'exercice précédent avec cette syntaxe simplifiée, et vérifiez que l'on obtient le même résultat.*

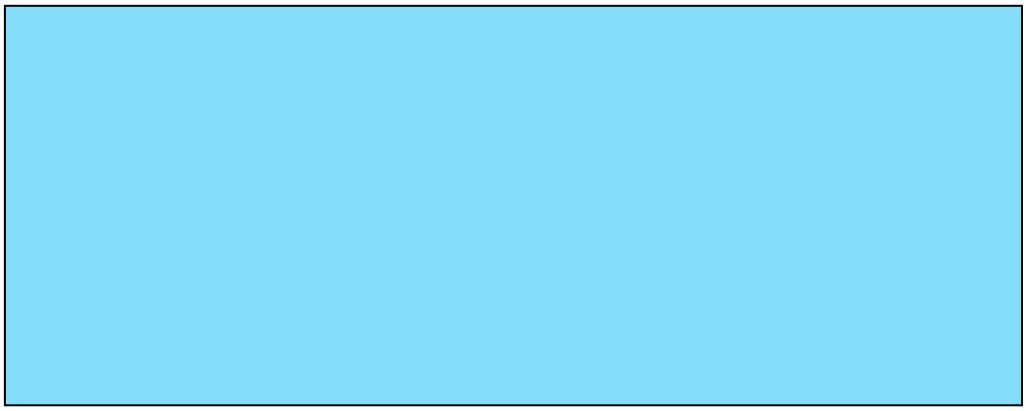


FIGURE 27 : REQUÊTE DE L'EXERCICE 14

3.4.9 Le pipeline avec group by et fonctions d'agrégation

La méthode *$group* dans le pipeline de *l'aggregate* est l'équivalent du GROUP BY en SQL avec la syntaxe :

{ $group : {_id :"critère de regroupement", "nom attribut" : {$fonctionAgregation :1}}

Le critère de regroupement est un des attributs de l'objet, qui va déterminer des sous-ensembles (par exemple *$fiche.dept* si l'on regroupe par département) Ou *null* si l'on veut travailler sur tout l'ensemble, sans le décomposer.

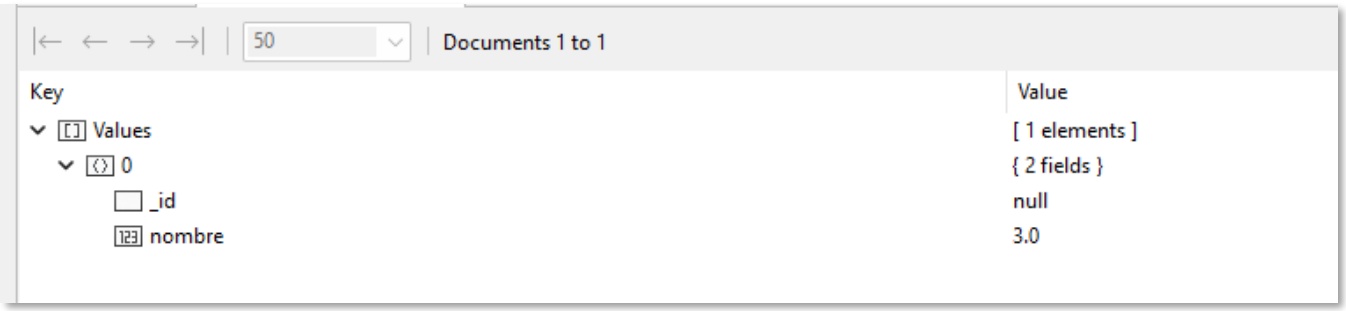
Les fonctions d'agrégation sont sensiblement les mêmes qu'en SQL, mais il faut tout de même faire attention :

- *$sum* :
  - avec la syntaxe {$sum :1} compte le nombre de documents, comme *count* en SQL
  - avec la syntaxe {$sum :attribut} fait la somme d'un attribut, comme *sum* en SQL

    https://docs.mongodb.com/manual/reference/operator/aggregation/sum/#grp._S_sum
- *$min* : valeur minimale de l'attribut
- *$max* : valeur maximale de l'attribut
- *$avg* : moyenne de l'attribut etc.

```
// on utilise un $group sur tout l'ensemble,
// sans critère de regroupement (_id = null) pour  compter les
//bibliothèques dont la première note est 1
const varMatch = {$match : { "notation.notes.0":1} };
const varGroup = {$group :  {_id:null, "nombre" : {$sum:1} } };

db.biblio.aggregate( [ varMatch, varGroup ] )
```



Exercice 15 : *Le résultat obtenu est correct mais peu lisible ("id" : null). Retirez les affichages inutiles*

(en ajoutant une projection dans le tableau, après le regroupement) :

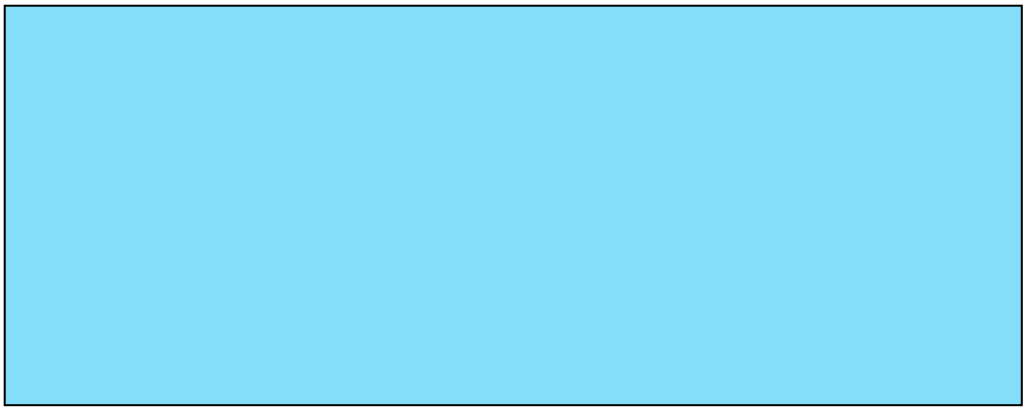


FIGURE 28 : REQUÊTE DE L'EXERCICE 15

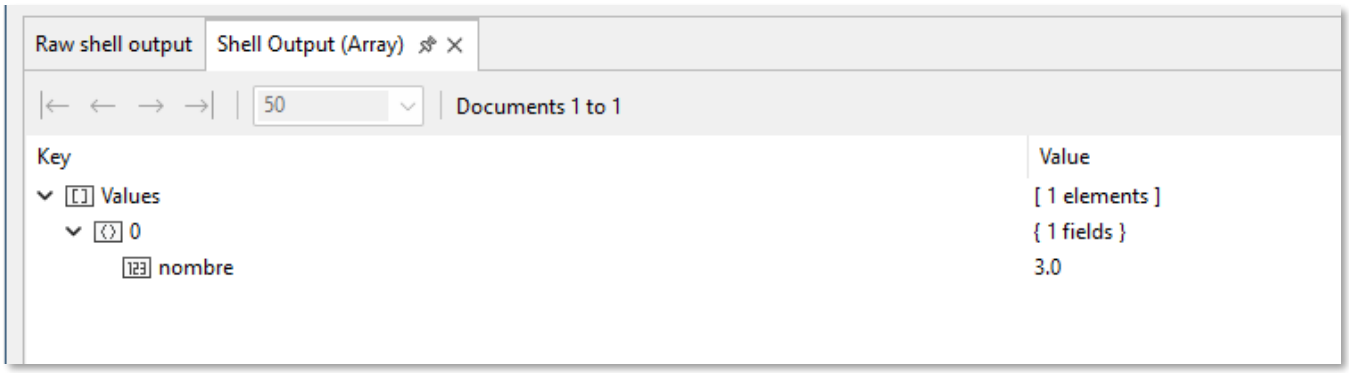


FIGURE 29 : CAPTURE DU RÉSULTAT

Exercice 16 : *Regroupez les bibliothèques par département, en comptant leur nombre par département. Affichez chaque numéro de département et le nombre de bibliothèques du département.*

{$group : {_id: "$fiche.dept", "nombre" : {$sum:1} } }

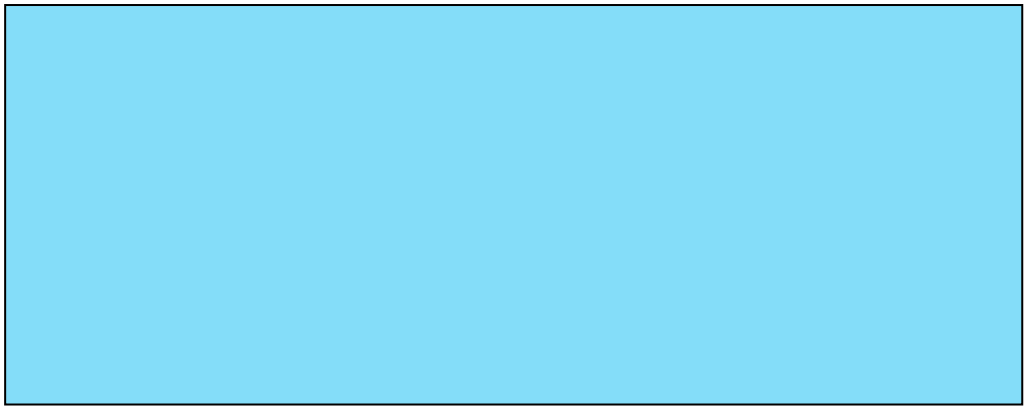
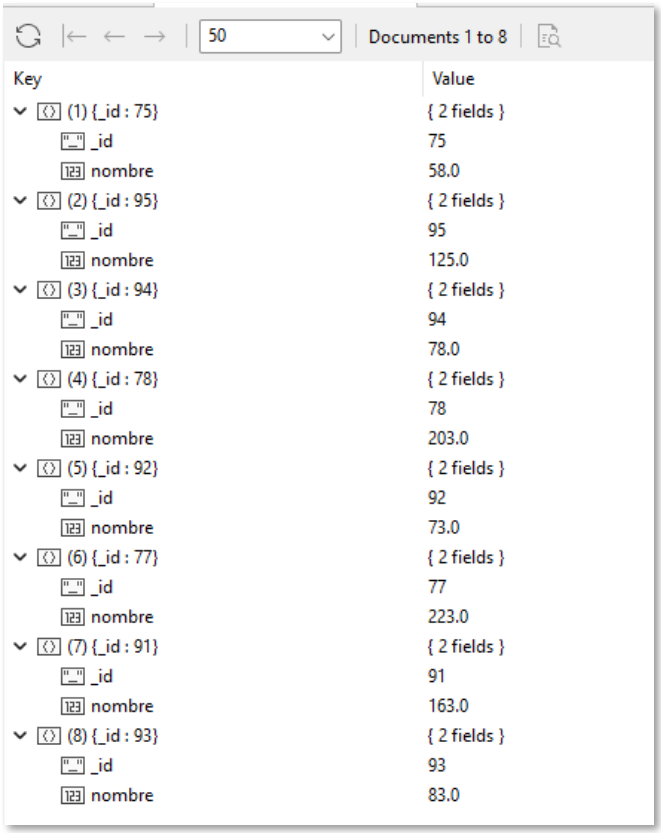


FIGURE 30 : RÉSULTAT DE LA REQUÊTE



50 | Documents 1 to 8

Key	Value
(1) {_id : 75}	{ 2 fields }
_id	75
nombre	58.0
(2) {_id : 95}	{ 2 fields }
_id	95
nombre	125.0
(3) {_id : 94}	{ 2 fields }
_id	94
nombre	78.0
(4) {_id : 78}	{ 2 fields }
_id	78
nombre	203.0
(5) {_id : 92}	{ 2 fields }
_id	92
nombre	73.0
(6) {_id : 77}	{ 2 fields }
_id	77
nombre	223.0
(7) {_id : 91}	{ 2 fields }
_id	91
nombre	163.0
(8) {_id : 93}	{ 2 fields }
_id	93
nombre	83.0

FIGURE 31 : CAPTURE DU RÉSULTAT [8]

3.4.10 Utiliser d'autres fonctions d'agrégation : moyenne, min, max

```
// Calculer la moyenne des notes des bibliothèques,
// regroupées par appréciation
varMatch = {$match : { notation:{$exists:1}} };
varUnwind = { $unwind :"$notation.notes" };
varGroup = {$group :  {
    _id:"$notation.appreciation",
    "moyenne" : {$avg:"$notation.notes" }
 }};
 db.biblio.aggregate( [ varMatch, varUnwind,  varGroup ] )
```

Key	Value	Type
(1) {_id : moyen}	{ 2 fields }	Document
_id	moyen	String
moyenne	2.3333333333333335	Double
(2) {_id : très moyen}	{ 2 fields }	Document
_id	très moyen	String
moyenne	1.8571428571428572	Double
(3) {_id : ok}	{ 2 fields }	Document
_id	ok	String
moyenne	4.121212121212121	Double

A noter :

- Il faut vérifier qu'un objet (*notation*) existe par $*exists*, avant de prendre un de ses attributs (*appreciation*) comme critère de regroupement
- On ne peut pas appliquer directement les fonctions d'agrégation sur des collections imbriquées dans un objet : ici sur la collection *notes* qui appartient à l'objet *notation*
- Il faut d'abord recréer une collection au premier niveau de l'objet principal en appliquant la méthode $*unwind* (sur *$notation.notes*)

Exercice 17 : *Regroupez les bibliothèques par département, en affichant la note la plus haute et la note la plus basse par département.*

Attention : les min et les max portent sur la collection notes qui est imbriquée dans l'objet notation. Pour accéder à ses éléments, il faut d'abord utiliser l'opérateur $unwind { $unwind :"$notation.notes" }

FIGURE 32 : REQUÊTE DE L'EXERCICE 17

50 | Documents 1 to 2 | Tree View

Key	Value	Type
(1)	{ 3 fields }	Document
note_max	5	Int32
note_min	1	Int32
departement	91	String
(2)	{ 3 fields }	Document
note_max	5	Int32
note_min	2	Int32
departement	92	String

FIGURE 33 : RÉSULTAT

3.4.11 Modifier la base : ajouter un champ, modifier un champ, supprimer un champ sur un objet existant

- Comme en SQL, on va pouvoir modifier la base, par la méthode *update*.
- Nous n'allons pas raisonner en tables et en lignes mais sur des objets et des attributs :
  - Une base NoSQL contient des collections d'objets JSON que l'on peut modifier directement, ou modifier en JavaScript puis sauvegarder.
  - Avantage du NoSQL : on peut modifier non seulement le contenu de l'objet (les valeurs des attributs) mais également sa structure, en ajoutant, supprimant des attributs ou en modifiant leur nom.
- Pour commencer, nous allons ajouter grâce à la méthode *update* et l'opérateur *$set,* un commentaire avec la mention "*** *Coup de cœur* ***" à une bibliothèque avec une appréciation "ok" et des notes élevées (que vous choisirez vous-même dans votre base) : Le premier paramètre de la méthode *update* décrit le ou les objets à modifier (l'équivalent du WHERE)

Le deuxième paramètre décrit les différentes modifications à effectuer (par l'opérateur *$set*)

```
db.Biblio.update (
    { _id:ObjectId("5d4054ba3b447b3ecda889d0") },
    { $set : { "commentaire " : "*** Coup de coeur ***" } }
)
```

- Puis, vous vérifierez que le nouvel attribut a bien été créé, en faisant un *find* :

```
db.Biblio.find ( {_id:ObjectId("5d4054ba3b447b3ecda889d0")})
```

Key	Value	Type
(1) {_id : 5d4054ba3b447b3ecda889d0}	{ 5 fields }	Document
_id	5d4054ba3b447b3ecda889d0	ObjectId
fiche	{ 17 fields }	Object
notation	{ 2 fields }	Object
appreciation	ok	String
notes	[4 elements]	Array
record_timestamp	2018-06-08T15:12:13+02:00	String
commentaire	*** Coup de coeur ***	String

- On pourra aussi modifier la valeur d'un champ existant (équivalent d'un « setCommentaire » en objet)

```
db.Biblio.update (
    {_id:ObjectId("5d4054ba3b447b3ecda889d0")},
    { $set : { "commentaire " : " – En fait, assez moyen !!" } }
)
```

(1) {_id : 5d4054ba3b447b3ecda889d0}	{ 5 fields }
_id	5d4054ba3b447b3ecda889d0
fiche	{ 17 fields }
notation	{ 2 fields }
appreciation	ok
notes	[4 elements]
record_timestamp	2018-06-08T15:12:13+02:00
commentaire	– En fait, assez moyen !!

- Ou supprimer un champ existant :

```
db.biblio.update (
    {_id:ObjectId("5d4054ba3b447b3ecda889d0")},
    { $unset : { "commentaire " : 1 } }
)
db.Biblio.find ( {_id:ObjectId("5d4054ba3b447b3ecda889d0")})
```

Key	Value
(1) {_id : 5d4054ba3b447b3ecda889d0}	{ 4 fields }
_id	5d4054ba3b447b3ecda889d0
fiche	{ 17 fields }
notation	{ 2 fields }
appreciation	ok
notes	[4 elements]
record_timestamp	2018-06-08T15:12:13+02:00

Exercice 18 : *Faites toutes les opérations précédentes sur un attribut de votre choix, en jouant à la fois sur le contenu et la structure des objets. Pour chaque manipulation, vérifiez le résultat par un find*

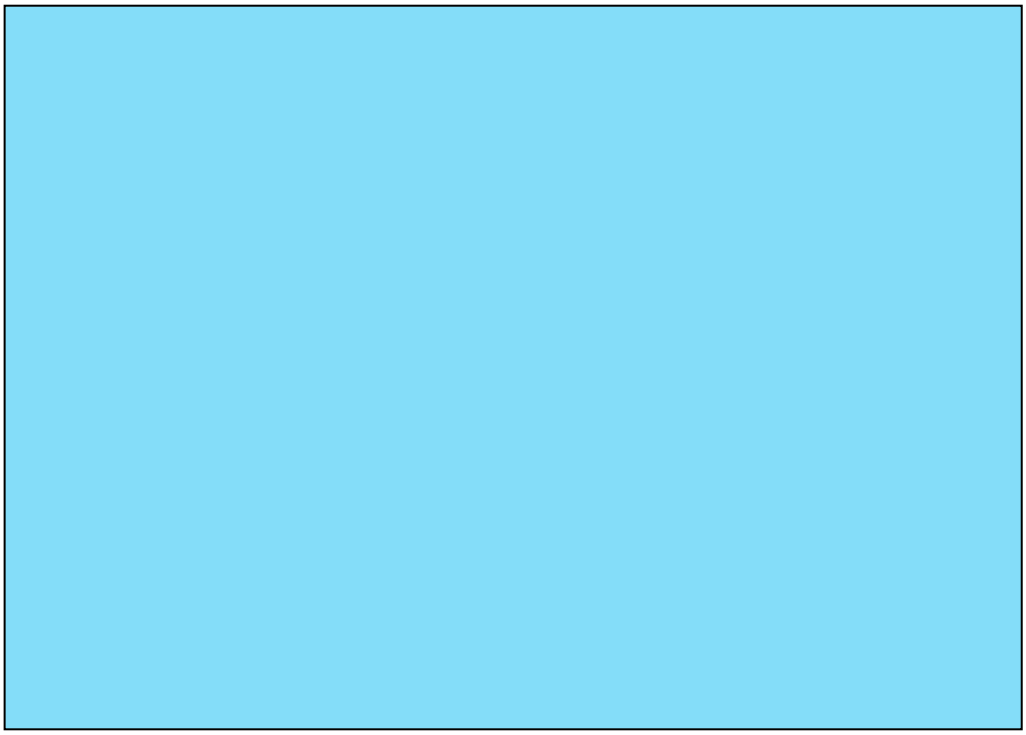


FIGURE 34 : VOS REQUÊTES

3.4.12 Modifications multiples (sur plusieurs objets d'une collection)

Les manipulations précédentes sont utiles, mais elles ne s'appliquent qu'à un seul objet désigné par son identifiant _id

- On a aussi besoin de traitements génériques : augmenter de 10% le prix de tous les ours en peluche…
- La méthode ***update*** doit alors comporter une clause « ***multi*** » qui autorise les modifications multiples dans une collection (par défaut, elles sont interdites, par mesure de sécurité) : {"multi" : true}

Exercice 19 : *En généralisant le cas précédent, ajoutez un commentaire « A visiter … » à toutes les bibliothèques qui ont reçu une appréciation « ok » et une première note à 5.*

(35 enregistrements modifiés). Vérifiez le traitement par un find, en n'affichant que les noms, l'appréciation et les notes, et le commentaire

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 35,
  modifiedCount: 35,
  upsertedCount: 0
}
```

FIGURE 35 : VOTRE REQUÊTE

Key	Value
(1)	{ 3 fields }
fiche	{ 1 fields }
nom	Bibliothèque Municipale
notation	{ 2 fields }
appreciation	ok
notes	[3 elements]
commentaire	A visiter ...
(2)	{ 3 fields }
(3)	{ 3 fields }

On voit qu'on peut enrichir notre base, sans « tout casser », de manière plus souple qu'avec une base relationnelle.

3.4.13 Renommer un attribut

Exercice 20 : *Dans toute la collection, renommer l'attribut* record_timestamp *en* date_maj*, pour faciliter la compréhension de la base.*

Utilisez l'opérateur *$rename*, en n'oubliant pas la clause *multi* : { $rename: { "xxxx": "yyyy" } }, { multi: true }

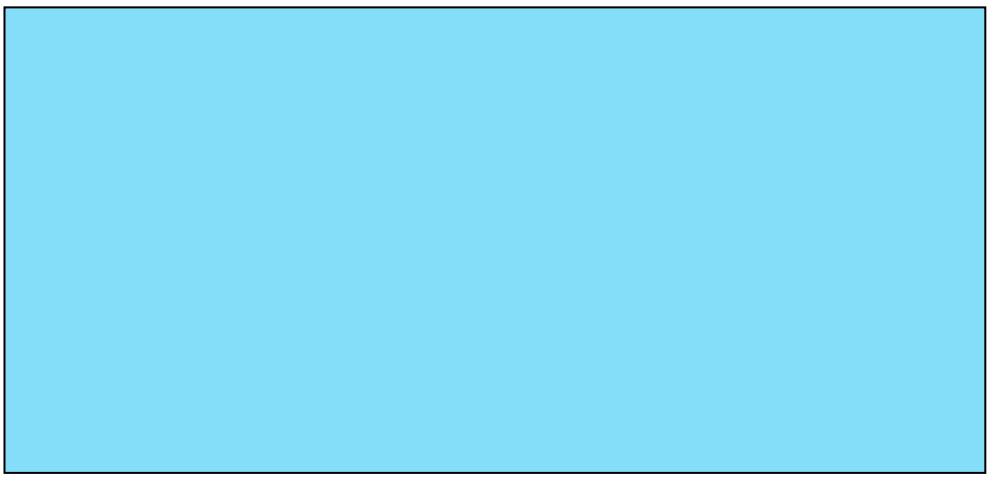


FIGURE 36 : VOTRE REQUÊTE

FIGURE 37 : 1006 MISES À JOUR

3.4.14 Manipuler des attributs de type tableau

Les tableaux peuvent être manipulés comme des listes, où l'on peut ajouter ou supprimer le premier ou le dernier élément

```
// On considère que les premières notes sont trop anciennes
// et doivent être retirées de la liste des notes.
db.biblio.update(
    {},
    { $pop: { "notation.notes" : -1 } },
    { multi: true }
)
```

L'opérateur ***$pop*** permet de supprimer un élément en tête de tableau (avec -1) ou en fin de tableau (avec 1)
https://docs.mongodb.com/manual/reference/operator/update/pop/#definition

Exercice 21 : *Faites l'essai ci-dessus et vérifiez le fonctionnement par un find.*

FIGURE 38 : PLUS QUE 2 NOTES

3.4.15 Insérer de nouveaux enregistrements dans une collection

L'insertion se fait par la méthode *insertOne* :

```
// directement
db.Biblio.insertOne (
    {
        categorie:"Bibliothèque Nationale",
        nom:"BNF Richelieu", ville:"Paris"
    })
```

Ou en passant par un objet de travail :

```
// ou en passant par un objet de travail
nouvBibli = {
    categorie:"Bibliothèque Nationale",
    nom:"BNF François Mitterrand", ville:"Paris"
}
db.biblio.insertOne (nouvBibli)
```

NOTE : Pour la suite de la pratique, une connaissance de JAVASCRIPT est nécessaire. Je vous propose de reprendre ces derniers exemples après avoir pratiqué le langage.

Mais on peut aussi créer un enregistrement à partir d'un enregistrement existant, en utilisant JavaScript.

```
// rechercher l'enregistrement : ici "BNF Richelieu"
// qui va servir de modèle
// on crée "Beaubourg" sur ce modèle
db.biblio.find (
  {
    nom:"BNF Richelieu"}).forEach (function (bib) {
      // print (bib.categorie, bib.ville)
      nouvBibli = {
        categorie:bib.categorie,
        nom:"Beaubourg",
        "ville":bib.ville
    }
      db.biblio.insertOne (nouvBibli)
    }
)
```

La méthode *forEach* utilisée sur la collection renvoyée par le *find permet de parcourir cette collection et d'exécuter sur chaque élément une fonction JavaScript passée en paramètre. Le paramètre « bib » passé à la fonction est l'élément courant dans la collection.*

3.4.16 Supprimer des enregistrements, selon un critère

```
// toutes les bibliothèques
// de la catégorie "Bibliothèque Nationale"
db.biblio.deleteMany( {categorie:"Bibliothèque Nationale"} )
```

Exercice 22 : *Faites différents essais d'insertion, avec des constantes et en se basant sur des enregistrements existants. Puis supprimez tous les nouveaux enregistrements, pour remettre la base dans son état initial.*

3.4.17 Utiliser JavaScript pour des opérations plus complexes.

Le parcours de collection par *forEach* permet de faire des traitements complexes en JavaScript sur les éléments.

Exercice 23 : *Ajouter une moyenne des notations à chaque bibliothèque :*

Attention : toutes les bibliothèques n'ont pas de notation

Dans un objet JavaScript « bib », pour créer un nouvel attribut « moyenne », il suffit de l'initialiser : *bib.moyenne = laMoyenne ;* et de sauver l'objet « bib » dans la base de données : *db.biblio.save (bib)*

Exercice 24 : *Faire une extraction de données pour peupler une autre collection.*

(Cette exercice suit le précédent : consultez d'abord le corrigé précédent, si vous avez un doute)

- *Créez une collection « appreciation » dont les objets comprendront 4 attributs :*

  - idbibli : id de la bibliothèque,

  - appreciation (tirée de notation.appreciation),

  - moyenne (tirée de la moyenne calculée précédemment)

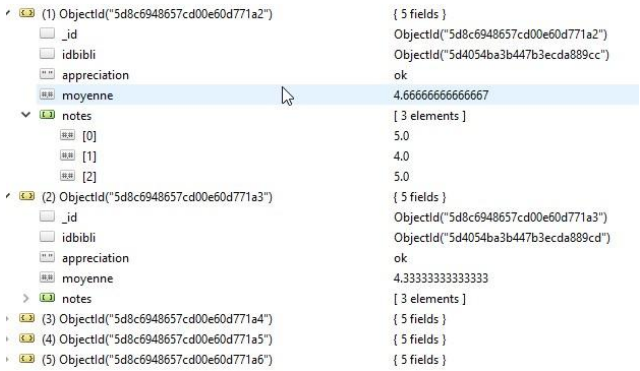
  - notes (tirées de notation.notes)

(50 objets en tout)

- *Supprimez ensuite les attributs notation et moyenne de la collection d'origine, pour éviter la redondance.*

Pour peupler la nouvelle collection, on utilisera la méthode *insertOne* avec des objets définis en JSON, à partir des attributs de l'objet courant « bib » dans la liste des bibliothèques :

{idbibli : bib._id, appreciation:bib.notation.appreciation, etc. }



Pour supprimer les champs *notation* et *moyenne dans la collection « biblio »*, faire un *$unset* sur ces champs, avec l'option « *multi* »

3.4.18 Faire une "jointure" entre plusieurs collections

On peut déjà faire beaucoup de choses avec une seule collection, en NoSQL.

Mais il restera des cas où une jointure ou son équivalent sera nécessaire entre deux collections, pour éviter ou limiter la redondance de données.

L'exercice précédent a préparé le travail : la collection "*appreciation*" contient maintenant les notes, moyenne et appréciation des bibliothèques, qui sont référencées par leur id (champ *idbibli*, qui tient lieu de clé étrangère).

Ces attributs ne sont plus dans la collection d'origine.

Nous allons maintenant croiser les données de ces deux collections par un *$lookup* (équivalent du *join* sur MongoDB) :

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

https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/

```
db.appreciation.aggregate([{
        $lookup:
         {
           from: "biblio",
           localField: "idbibli",
           foreignField: "_id",
           as: "bibliotheque"
         }
}])
```

Exercice 25 : *Faites le traitement ci-dessus et observez le résultat*


```
/* 1 */
{
    "_id" : ObjectId("5d8c6948657cd00e60d771a2"),
    "idbibli" : ObjectId("5d4054ba3b447b3ecda889cc"),
    "appreciation" : "ok",
    "moyenne" : 4.6666666666667,
    "notes" : [
        5.0,
        4.0,
        5.0
    ],
    "bibliotheque" : [
        {
            "_id" : ObjectId("5d4054ba3b447b3ecda889cc"),
            "fiche" : {
                "categorie" : "Bibliothèque municipale",
                "région" : "Ile-de-France",
                "code_bib" : 4067.0,
                "insee" : "91021",
                "nom_dept" : "ESSONNE",
                "code_ua" : 4067.0,
                "voie" : "31 rue Dauvilliers",
```


(50 objets en tout)

Les attributs de la bibliothèque « *idbibli* » sont stockés dans un tableau déclaré par la clause « as » du lookup : « *bibliothèque* »

3.5 APPELER MONGODB DEPUIS UN CLIENT JAVA

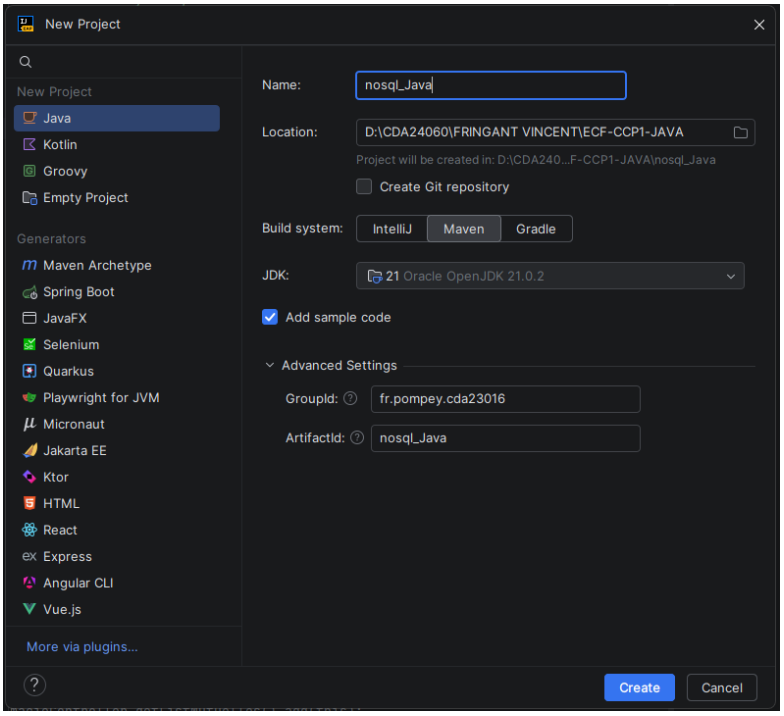
Nous allons maintenant laisser de côté l'interpréteur JavaScript et écrire quelques requêtes vers MongoDB avec une bibliothèque Java.

NOTE : Pour la suite de la pratique, on utilise Maven, outil de gestion de projet Java. Cet outil travaille avec un fichier de gestion appelé pom.xml. C'est dans ce fichier en autres qu'on configure les librairies utilisées dans le projet.

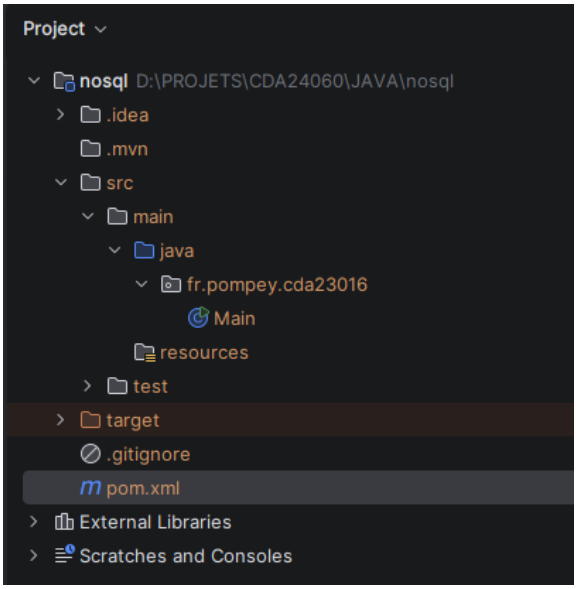
On n'a pas encore découvert cet outil. Je vous propose pour ceux qui le souhaite de reprendre ce point après avoir pratiqué l'outil.

3.5.1 Création du projet Maven

- Nous allons vous guider dans la création d'un premier client lourd Java vers MongoDB
- Vous allez vous approprier le code commenté, réaliser et tester chaque étape.
- Créez un projet Java « maven »



À la suite de la création, Maven vous a créé une structure de projet complète avec notre fichier pom.xml :



- avec le fichier pom suivant, qui permet de charger le driver de mongo-db :

```
<properties>
    <maven.compiler.source>21</maven.compiler.source>
    <maven.compiler.target>21</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
    <!-- https://mvnrepository.com/artifact/org.mongodb/mongodb-driver-sync -->
    <dependency>
        <groupId>org.mongodb</groupId>
        <artifactId>mongodb-driver-sync</artifactId>
        <version>5.5.1</version>
    </dependency>
</dependencies>
```

3.5.2 Les premiers pas : valider l'environnement

Pour vérifier que notre client parvient à interroger le serveur MongoDB local, créez une classe *Main* dans votre projet, en vous inspirant de l'extrait de code qui suit :

```java
public class Main {
    public static void main(String[] args) {

        Main obj = new Main();
        obj.start();
    }

    private void start() {  1 usage

        System.out.println("Demo client java MongoDB");

        final String uriString = "mongodb://localhost:27017";
        MongoIterable<String> liste;
        try (MongoClient mongoClient = MongoClients.create(uriString)) {


            // liste les bases de données disponibles sur le serveur
            liste = mongoClient.listDatabaseNames();

            System.out.println("Liste database: ");
            for (String s : liste) {
                System.out.println(s);
            }
        } catch (Exception ex) {
            System.err.println(ex.getMessage());
        }

    }
}
```

- Le programme se connecte au serveur MongoDB local grâce à la bibliothèque *MongoClient*
- Puis il liste les bases de données présentes sur ce serveur : *admin, config, local* etc.

```
C:\Users\jboeb\.jdks\openjdk-21.0.2\bin\java.
Demo client java MongoDB
sept. 04, 2025 2:50:58 PM com.mongodb.interna
WARNING: SLF4J not found on the classpath.  L
Liste database:
admin
config
local

Process finished with exit code 0
```

3.5.3 Une première requête simple

Nous allons commencer classiquement par compter le nombre d'objets dans notre collection « biblio »

```java
// selectionne la base de donnée admin
MongoDatabase database = mongoClient.getDatabase( s: "admin");
System.out.println("Liste database: " + database.getName());

// affiche le nombre de documents de la collection biblio
MongoCollection<Document> collection = database.getCollection( s: "Biblio");
System.out.println("Nombre de bibliothèques : " + collection.countDocuments());
```

Le programme sélectionne la base de données *admin* et charge la collection *biblio* sur laquelle il fait un *count* :

```
Liste database: admin
Nombre de bibliothèques : 1006
```

3.5.4 Une requête paramétrée

Nous allons faire un *find* avec un paramètre en JSON pour ne récupérer que les bibliothèques du département de l'Essonne.

```
// affiche les bibliothèques de l'ESONNE
Bson bson = Filters.eq( fieldName: "fiche.nom_dept",  value: "ESSONNE");
System.out.println("**** Bibliothèques de l'Essonne : ");
FindIterable<Document> listeEssonne = collection.find(bson);

for (Document doc : listeEssonne) {
    Document fiche = (Document) doc.get("fiche");
    System.out.println("Département : " + fiche.get("nom_dept")
        + "\tNom :" + fiche.get("nom"));
}
```

Le paramètre pour filtrer est un objet JSON, comme dans les appels en JavaScript : {fiche.nom_dept :'ESSONNE'}

Pour l'API de MongoDB en Java, le format JSON doit au préalable être converti en format Bson (*Binary Json*) qui est aussi le format de stockage de MongoDB.

Dans chaque objet de la liste (de type *org.bson.Document*), on extrait d'abord l'objet "*fiche*" par un *get* sur le nom de l'attribut, puis dans cet objet, les attributs "*nom_dept*" et "*nom*" de type String.

```
**** Bibliothèques de l'Essonne :
Département : ESSONNE   Nom :Bibliothèque  Municipale
Département : ESSONNE   Nom :Bibliothèque  Municipale
Département : ESSONNE   Nom :Bibliothèque  Municipale Georges Brassens
Département : ESSONNE   Nom :BOUSSY-Bibliothèque Communautaire
Département : ESSONNE   Nom :BRUNOY-Bibliothèque Communautaire
Département : ESSONNE   Nom :Bibliothèque Jean-Jacques SEMPE
Département : ESSONNE   Nom :Mediatheque Municipale De Cheptainville
```

https://howtodoinjava.com/mongodb/mongodb-find-documents/

3.5.5 Insérer de nouveaux objets et les afficher

- En NoSQL, nous ne sommes pas limités par un modèle de données préalable.
- Pour illustrer ce point, nous allons insérer dans la collection « *biblio* » de nouveaux objets qui ne seront pas des bibliothèques.
- Dans la vie réelle, on n'insère pas n'importe quoi dans une collection existante. Rappelons la règle d'or de l'objet qui reste valable en NoSQL : pas de fourre-tout ; la souplesse ne doit pas conduire à une absence de structuration.

- Mais l'on pourra ajouter de nouveaux attributs à l'objet bibliothèque, d'autres types d'objet proche de la bibliothèque ou en lien fonctionnel avec elle (bibliobus, points d'échange de livres, administration en charge des bibliothèques etc.)
- Pour la démonstration technique, nous allons ajouter deux objets simples, avec un seul attribut « *essai* » de type entier, et les réafficher :

```
System.out.println("**** Création de documents : ");
for (int i=1; i <=2; i++) {
    Document d = new Document().append("essai", i);
    collection.insertOne(d);
}

System.out.println("*** Documents créés : ");
bson = Filters.exists( fieldName: "essai",  exists: true);

for (Document doc : collection.find(bson) ) {
    System.out.println(doc);
}
```

- La méthode *append* de la classe Document permet d'ajouter un attribut (clé, valeur) à un document.
- La méthode *insertOne* permet d'ajouter un nouveau document dans une collection de documents existante.
- On filtre la collection (par un *find*) sur les objets qui possèdent un attribut « *essai* » et on affiche la collection filtrée.

```
**** Création de documents :
*** Documents créés :
Document{{_id=68b9950b13423916250fa66e, essai=1}}
Document{{_id=68b9950b13423916250fa66f, essai=2}}
```

3.5.6 Supprimer des objets dans une collection

On supprimera les objets dans une collection « *col* » par la méthode *deleteMany* avec le même filtre BSON que pour un *find*

- On réaffichera les objets avec le même filtre pour vérifier leur suppression.

```
// Suppression
collection.deleteMany(bson);
System.out.println("*** Documents delete : ");

for(Document doc : collection.find(bson)) {
    System.out.println(doc);
}
```

Code complet Java :

```java
package fr.pompey.cda23016;

import com.mongodb.client.*;
import com.mongodb.client.model.Filters;
import org.bson.Document;
import org.bson.conversions.Bson;

//TIP To <b>Run</b> code, press <shortcut actionId="Run"/> or
// click the <icon src="AllIcons.Actions.Execute"/> icon in the gutter.
public class Main {
    public static void main(String[] args) {

        Main obj = new Main();
        obj.start();
    }

    private void start() {  1 usage

        System.out.println("Demo client java MongoDB");

        final String uriString = "mongodb://localhost:27017";
        MongoIterable<String> liste;
        try (MongoClient mongoClient = MongoClients.create(uriString)) {


            // liste les bases de données disponibles sur le serveur
            liste = mongoClient.listDatabaseNames();

            System.out.println("Liste database: ");
            for (String s : liste) {
                System.out.println(s);
            }

            // selectionne la base de donnée admin
            MongoDatabase database = mongoClient.getDatabase( s: "admin");
            System.out.println("Liste database: " + database.getName());

            // affiche le nombre de documents de la collection biblio
            MongoCollection<Document> collection = database.getCollection( s: "Biblio");
            System.out.println("Nombre de bibliothèques : " + collection.countDocuments());
```

```java
        // affiche les bibliothèques de l'ESONNE
        Bson bson = Filters.eq( fieldName: "fiche.nom_dept",  value: "ESSONNE");
        System.out.println("**** Bibliothèques de l'Essonne : ");
        FindIterable<Document> listeEssonne = collection.find(bson);

        for (Document doc : listeEssonne) {
            Document fiche = (Document) doc.get("fiche");
            System.out.println("Département : " + fiche.get("nom_dept")
                + "\tNom :" + fiche.get("nom"));
        }

        System.out.println("**** Création de documents : ");
        for (int i=1; i <=2; i++) {
            Document d = new Document().append("essai", i);
            collection.insertOne(d);
        }

        System.out.println("*** Documents créés : ");
        bson = Filters.exists( fieldName: "essai",  exists: true);

        for (Document doc : collection.find(bson) ) {
            System.out.println(doc);
        }

        // Suppression
        collection.deleteMany(bson);
        System.out.println("*** Documents delete : ");

        for(Document doc : collection.find(bson)) {
            System.out.println(doc);
        }
    } catch (Exception ex) {
        System.err.println(ex.getMessage());
    }
}
```

CRÉDITS

OEUVRE COLLECTIVE DE L'AFPA

Document tiré du document original de

Régis Lécu – Formateur AFPA Pont de Claix

Reproduction interdite

Article L 122-4 du code de la propriété intellectuelle.

« Toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droits ou ayants cause est illicite. Il en est de même pour la traduction, l'adaptation ou la reproduction par un art ou un procédé quelconque. »