



UNIVERSITÉ DE ROUEN

DÉPARTEMENT DES SCIENCES ET TECHNIQUES

MASTER 2.2 BIOINFORMATIQUE MODELISATION ET STATISTIQUE (BIMS)

ANNÉE 2017-2018

---

# Base de données noSQL: MongoDB

---

Par

ANNTHOMY GILLES

KEVIN DURIMEL

MAXIME BODINIER

Sous la responsabilité de Dr.LINA SOUALMIA

# Summary

1	Présentation des différences entre les SGBD NoSQL . . . . .	1
2	MongoDB : exploration des différents concepts par la pratique . . . . .	6
2.1	Installation sur OS GNU/Linux (ici Ubuntu) . . . . .	7
2.2	Configuration et sécurisation de la base de données . . . . .	8
2.3	Langages de manipulations de données . . . . .	10
2.4	Création du jeu de données . . . . .	10
2.5	Optimisation de la base de données . . . . .	13
3	MongoDB : Exploitation de la base de données par la pratique (requêtes) - Cas d'usages . . . . .	20

# 1 Présentation des différences entre les SGBD NoSQL

Les bases de données NoSQL ("not only SQL") comprennent toutes les bases de données dont l'architecture n'est pas fondée sur le paradigme classique des bases de données relationnelles ("SQL"). Cette "nouvelle génération" de bases de données à émergé en synergie avec le développement du big data au cours des années 2010, avec la nécessité croissante de pouvoir exploiter des bases de données adaptée aux infrastructures matérielles massives exploitées par celui-ci [5] :

- Volume de données qui explose (double tous les 2 ans)
- Variété des types de données générées
- Vitesse avec laquelle ces données changent

Les bases de données NoSQL permettant la manipulation de volumes de données importants et une scalabilité horizontale, elles répondaient parfaitement aux besoins du big data. Toutefois, ses propriétés avantageuses ont été obtenues au prix de l'abandon des standards des SGBD relationnels. Ces concessions (pouvoir lire les données d'une manière qui n'était pas prévue lors de sa création, transactions pour les vérifications d'intégrité...) ont permis des traitements plus rapides pour ces types d'applications. Les bases de données reposent sur 3 principes issus du théorème de CAP d'Eric Brewer, et spécialement adaptées au big-data :

## **Cohérence (C : consistency)**

Tous les clients voient la même vue même lorsqu'il y a des mises à jour.

## **Haute disponibilité (A : availability)**

L'ensemble des clients peuvent trouver des données répliquées, même lorsqu'une avarie survient.

## **Tolérance au partitionnement (P : partition tolerance)**

Le système est tolérant au partitionnement et peut-être réparti sur plusieurs clusters

Cependant, seuls deux des trois principes peuvent être respectés en même temps. En général, la haute disponibilité et la tolérance à la partition sont priorisés car les besoins en disponibilité et en partitionnement sont souvent plus importants que la consistance dans un déploiement big-data. A l'heure actuelle, les structures des systèmes de gestion de bases de données restent très hétérogènes. De manière non exhaustive, on peut distinguer 4 structures types :

## Structure type 1 : les bases clé-valeur (Aerospike, Redis, Riak...)

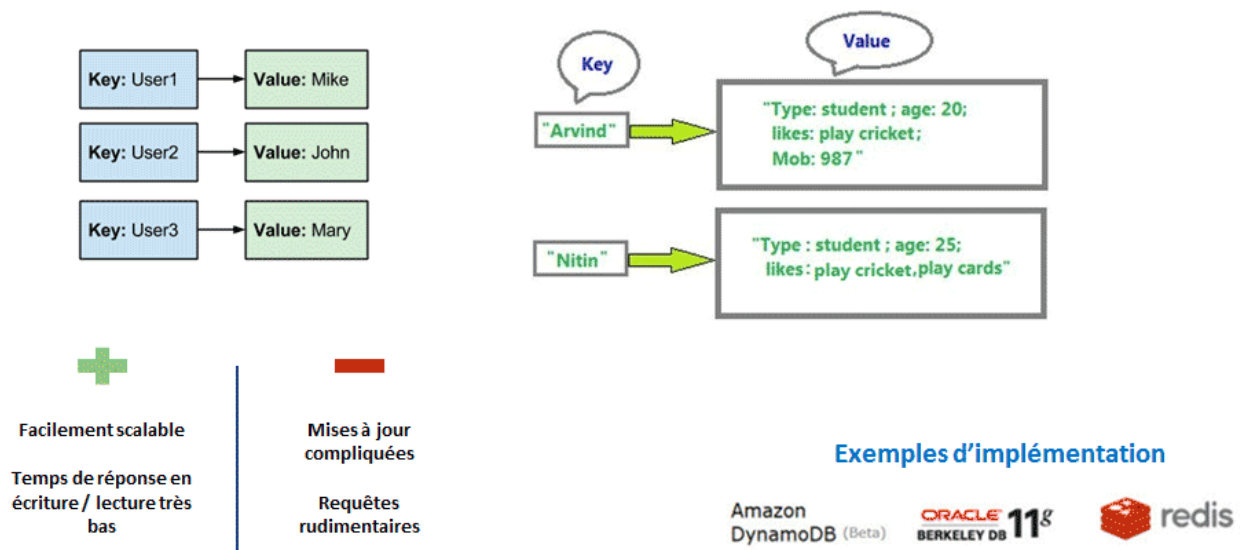


Figure 1: Représentation schématique d'une base clé-valeur, avantages, inconvénients et exemples d'implémentations SGBD

Crédits

C'est la forme la plus simple des structures de SGBD NoSQL. Elle associe des clés uniques à des valeurs (similaires à un dictionnaire en Python), avec pour objectif d'optimiser fortement les performances des applications reposant sur des jeux de données relativement simples, consultées et mises à jour en même temps (exemple : historique , informations d'un client...). Cette approche permet de résoudre le problème de montée en charge et en volume, mais sans tolérance aux pannes. Leur usage est pertinent pour la gestion de données ne nécessitant pas d'être structurées.

**Applications types\* :** Gestion d'historique, de fichier client...

**Avantages\* :**

- Légèreté : la plus légère des 4 structures NoSQL
- Evolutivité : forte évolutivité grâce à l'absence de structure ou de typage
- Très "scalable" : pas de jointures → donc pas de problème lorsque le nombre de données sera important et sera difficile à répartir entre plusieurs serveurs.
- Les meilleurs temps d'accès en lecture/écriture

**Inconvénients\* :**

- Requêtes rudimentaires : Utilisable uniquement sur des modèles d'interrogation (PUT, GET, DELETE...)
- Pas de tolérance aux pannes

- Très peu performant sur des requêtes SQL-like ( exemple: nombre total de clients : dans un SGBD relationnel cette information est directement retrouvable dans une table unique)

## Structure type 2 : les bases orientées colonnes (Cassandra, Accumulo, HBase...)

Leur structure est celle qui se rapproche le plus des bases des bases de données relationnelles, puisque les données sont sauvegardées sous forme de ligne avec les colonnes. Toutefois contrairement aux SGBD SQL, le nombre de colonnes peut varier d'une ligne à l'autre. Elles reposent sur le concept "BigTable" de Google [1]. Leur usage est pertinent pour le déploiement d'applications à grande échelle.

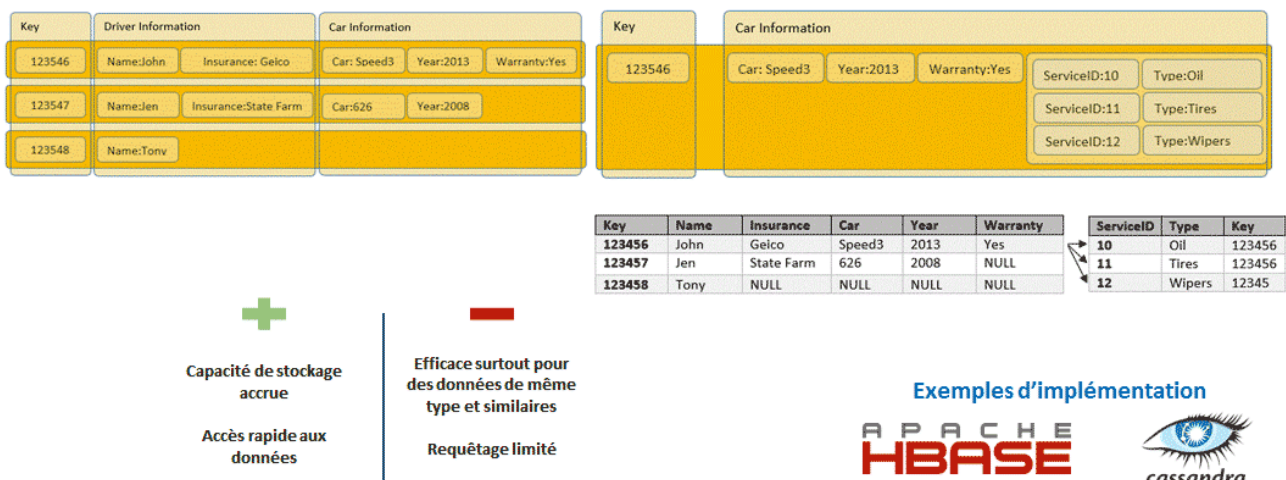


Figure 2: Représentation schématique d'une base orientée colonnes, avantages, inconvénients et exemples d'implémentations SGBD

### Crédits

Contrairement aux bases de données clé-valeur elles permettent de hauts niveaux de performance et de dimensionnement pour le parcours et le traitement d'important jeux de données dans leur intégralité.

**Applications types\*** : recherches internet, applications web à grande échelle, applications analytiques.

### Avantages\*

- Accès rapide aux données : structure "auto-indexée"
- Structure permettant un fort taux de compression des données (capacité de stockage accrue)
- Par conséquent, exécution rapide des opérations en colonnes (MIN, MAX, SUM, COUNT, AVG...)

### Inconvénients\*

- Très efficace en lecture mais beaucoup moins en écriture (une clé/index par colonne, les recherches sur les autres colonnes seront donc moins efficaces)
- Requêtage limité : nécessite l'utilisation de filtres pour des performances optimales [2] .

### Structure type 3 : les bases orientées graphe (InfiniteGraph, Titan, Neo4j...)

Cette structure se base sur la théorie des graphes pour représenter les données sous forme de noeuds et de relations, ce qui facilite la représentation d'ensembles de données issues du monde réel, c'est donc par définition aussi une base de données orientée objet. Leur usage est pertinent pour la gestion des données complexes et interconnectées (données spatiales, sociales, financières...).

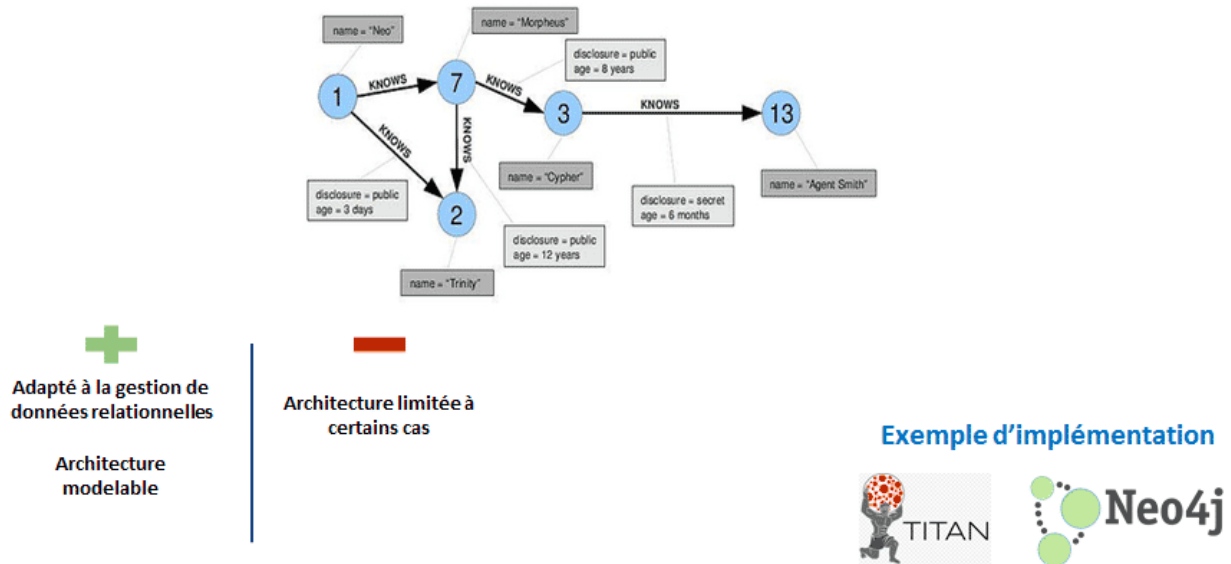


Figure 3: Représentation schématique d'une base orientée graphe, avantages, inconvénients et exemples d'implémentations SGBD

Crédits

Contrairement aux autres structures, les données (noeuds) sont organisées dans des graphes qui permettent de créer des associations (arrêtes) entre elles. Ces relations sont essentielles aux moteurs de recherche, afin d'associer des requêtes aux bons termes dans le bon contexte. L'intérêt de cette structure devient donc évident dans le cas de données complexes et interconnectées. C'est ainsi une structure idéale pour des recherches du type "partir d'un noeud et parcourir le graphe" plutôt que "trouver toutes les entités du type X", plus adaptées aux SGBD SQL. Il est cependant aussi possible d'effectuer des requêtes de ce dernier type, en utilisant un système d'indexation [7].

**Cas des triplestore :** Les triplestore, qui sont une forme particulière de base de données graphe, gèrent exclusivement des graphes binaires (non pondérés) de triplets RDF (Ressource description framework, donc centrés sur les relations) et sont destinés à proposer des inférences, contrairement aux autres bases de données orientées graphe qui fonctionnent avec différents types de graphes (pondérés, clusters, graphe et tables mixtes...).

**Applications types\* :** Réseaux sociaux, réseaux de transport, topologies, systèmes de recommandation de documents, inférences

**Avantages\* :**

- Traitement facilité de données fortement connectées (beaucoup de jointures auraient été

nécessaires en SQL)

- Architecture modelable.
- Très performant en lecture de type "parcours de graphe"

#### Inconvénients\* :

- Structure la moins tolérante aux partitionnements
- Pas performant pour le calcul de grandes agrégations de données (dénombrement de noeuds, filtrage par catégorie...) ou d'autres structures seront plus appropriées.

### Structure type 4 : les bases orientées document (MongoDB, CouchDB, RavenDB...)

Cette structure repose sur le paradigme clé-valeur, à la différence qu'ici la valeur est stockée au format JSON ou XML. Une seule clé permet ainsi de récupérer l'ensemble des informations hiérarchisées, opération qui nécessiterait plusieurs jointures dans un SGBD SQL. Leur usage est pertinent pour la gestion de documents que leur structures soient hétérogènes ou non.

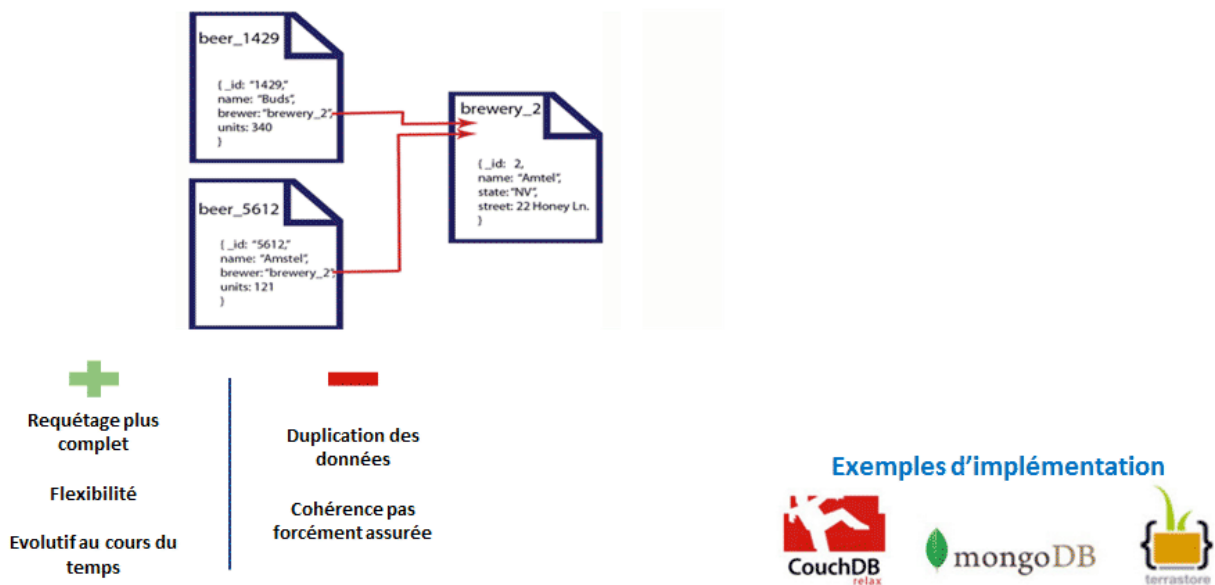


Figure 4: Représentation schématique d'une base orientée document, avantages, inconvénients et exemples d'implémentations SGBD

[Crédits](#)

#### Applications type\* : Applications mobiles ou web à fort trafic

#### Avantages\* :

- Structure la plus adaptée à la gestion de documents
- Schémas dynamiques permettant de modéliser tout format de données
- Récupérer très simplement de l'information structurée de manière hiérarchique

### **Inconvénients\* :**

- Nécessité et difficultés à revoir le schéma de la BDD en cas de besoin d'effectuer des requêtes initialement non prévues (chaque modification apportée à l'application implique potentiellement une adaptation de la conception de celle-ci)
- Cohérence pas toujours assurée

\*Liste non exhaustive

La montée en puissance de l'utilisation des SGBD NoSQL peut s'expliquer par un "ticket d'entrée" peu élevé pour le développeur, puisque la plupart des solutions proposées sur le marché reposent sur un modèle open source et sont relativement plus faciles à prendre en main que les SGBD SQL (à première vue!). De plus leur utilisation peut également se justifier dans des modèles d'usage qui ne sont pas impactés par les volumes ou la charge associée à leur traitement. En effet, les bases NoSQL présentent l'avantage d'être plus agiles, plus durables, et d'offrir une maintenance plus intelligente, ce qui offre (lorsqu'elles sont maîtrisées!) un compromis intéressant entre performances et robustesse.

### **Choix du SGBD pour le projet**

Nous avons opté pour le SGBD orienté document MongoDB parce-que le format des données à exploiter sera au format JSON, qui se prête particulièrement à l'utilisation dans un SGBD orientés documents. Ces données pourront être ensuite exploitées par du Javascript, directement intégré dans MongoDB.

## **2 MongoDB : exploration des différents concepts par la pratique**

On propose dans ce chapitre de montrer en quoi MongoDB se distingue des autres SGBD en illustrant nos propos d'exemples que nous avons réalisé à la manière d'un tutoriel.

### **Historique**

MongoDB est un SGBD orientée documents développé depuis 2007 par MongoDB. Cette entreprise travaillait alors sur un système de Cloud computing à données largement distribuées. MongoDB est alors répartissable sur un nombre quelconque d'ordinateurs et ne nécessite pas de schéma prédéfini des données. Il est devenu depuis un des SGBD les plus connus et utilisés.

### **Détail**

Les données prennent la forme de documents enregistrés dans des collections au format XML ou JSON : le modèle de données est donc dit semi-structuré. Une collection peut contenir un nombre quelconque de documents. Les collections sont comparables aux tables, et les documents aux enregistrements des SGBDR. La limite de taille d'une BDD MongoDB est de 16mo,

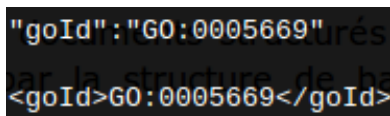


ce qui est considérable compte tenu du format exploité.

Contrairement aux SGBDR, les champs d'un enregistrement sont libres et peuvent être différents d'un enregistrement à un autre au sein d'une même collection. Le seul champ commun et obligatoire est le champ de clé principale ("id"). De plus, MongoDB ne permet ni les requêtes très complexes standardisées, ni les JOIN, mais permet de programmer des requêtes spécifiques en JavaScript.

## XML ou JSON?

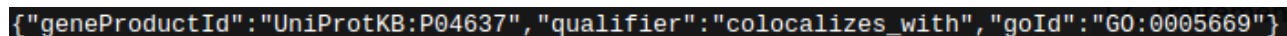
Le format XML très complet mais très lourd, le second a juste les qualités inverses. Ces formats sont, entre autres, conçus pour que le codage des documents soit adapté aux échanges dans un environnement distribué. Un document en JSON ou XML peut être transféré par réseau entre deux machines sans perte d'information et sans problème de codage/décodage.



```
"goId":"GO:0005669"
<goId>GO:0005669</goId>
```

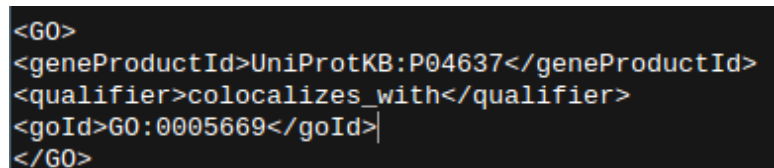
Figure 5: Une paire (clé-valeur) codée en JSON (haut) et codée en XML (bas)

La construction de paires (clé, valeur) permet de créer des agrégats. Voici un exemple JSON, dans lequel les agrégats sont codés avec des accolades ouvrante/fermante.



```
{"geneProductId":"UniProtKB:P04637","qualifier":"colocalizes_with","goId":"GO:0005669"}
```

En XML, il faut "nommer" l'agrégat (ici, "GO")



```
<GO>
<geneProductId>UniProtKB:P04637</geneProductId>
<qualifier>colocalizes_with</qualifier>
<goId>GO:0005669</goId>
</GO>
```

On constate tout de suite que le codage XML est beaucoup plus bavard que celui de JSON. Dans notre cas, n'ayant pas d'intérêt à utiliser le format XML et disposant de peu d'espace de stockage/ressources de calcul nous avons préféré opter pour le format JSON nativement plus léger que XML.

## 2.1 Installation sur OS GNU/Linux (ici Ubuntu)

Deux types de paquets sont disponibles: le paquet fourni par la communauté Ubuntu et le paquet fourni par la communauté MongoDB. Le deuxième comporte la version plus récente. pour plus de simplicité nous allons installer le paquet fourni par la communauté Ubuntu à l'aide d'apt :

---

```
1 sudo apt-get update | sudo apt-get install - mongodb-org
```

---

## 2.2 Configuration et sécurisation de la base de données

La configuration par défaut de MongoDB suffit à priori pour la plupart des utilisations en phase de développement. Cette configuration est disponible dans le fichier `/etc/mongodb.conf`.

Toutefois, la configuration par défaut de MongoDB est très permissive. Elle permet toutes les actions possibles sur la base de données sans besoin d'authentification, contrairement à la plupart des autres SGBD MySQL qui exigent une authentification par défaut.

Ainsi, de nombreuses bases de données MongoDB ont été impliquées dans plusieurs cas de fuites de données massives qui ont été très médiatisées. Pourtant, loin d'être causée par une véritable faille de sécurité, ces différentes fuites sont plutôt le résultat de la conjonction de plusieurs incompréhensions de la part des utilisateurs : d'une part la suppression/non-ajout de certains paramètres de sécurité d'autre part l'absence de mécanisme d'authentification par défaut.

Afin d'éviter les risques de compromission des données il est donc nécessaire de modifier cette configuration pour toute mise en production. MongoDB communique pourtant largement sur ces paramètres et sur les bonnes pratiques visant à sécuriser les bases de données de ce type text [8, 6, 4]. Les configurations à effectuer pour sécuriser la base de données sont, de manière non exhaustive, les suivantes :

### 1. Sécurité de base

- Filtrage par ip (limiter accès a la BDD uniquement depuis le serveur, par ex)
- Accès a la db par authentification
- Création d'un administrateur global
- Création d'un administrateur local
- Création d'un utilisateur standard
- Sauvegarde des données

### 2. Sécurité avancée

- Monitoring avec surveillance des évènements dans la BDD
- Cryptage des données
- Cryptage des communications

Nous allons appliquer les configurations permettant une sécurité de base, ce qui sera suffisant dans notre cas.

#### Filtrage par ip

Les adresses ip pouvant écouter la base de données sont inscrites dans le fichier de configuration (`.ini`) de MongoDB accessible à l'adresse `/etc/mongodb.conf` . Il suffit d'ajouter ou de décommenter la ligne "network interfaces" et y ajouter les adresses ip et port d'écoute autorisés.

---

```
1 # network interfaces
2 net:
3   port: 27017
4   bindIp: 127.0.0.1 # écouter uniquement en local
```

---

## Accès a la db par authentification

Toujours dans le même fichier .ini, décommenter ou ajouter la ligne "security" :

---

```
1 security:
2   authorization: enabled
```

---

## Création d'un administrateur global(toutes les bases de données)

Il va tout d'abord falloir démarrer MongoDB (daemon) puis lancer son shell:

---

```
1 systemctl start mongod
2 mongo
```

---

L'administrateur global pourra administrer toutes les bases de données initialisées ainsi que les utilisateurs, mais ne pourra pas lire et écrire sur celles-ci, les privilèges de cet utilisateur correspondent au rôle "userAdminAnyDatabase" prédéfini dans MongoDB. De plus amples détails sur celui-ci sont disponibles dans la documentation officielle de MongoDB [\[3\]](#)

```
1 db.createUser({user: "globalAdmin", pwd: "mongopass1", roles: ["
                                userAdminAnyDatabase"]})
```

## Création d'un administrateur local (à une base de donnée, celle ou il à été crée)

L'administrateur local dispose des mêmes privilèges que l'administrateur global à la différence que ceux-ci s'appliquent uniquement à la base de données ou celui-ci à été initialisé. Nous allons créer un administrateur spécifique à la future base de données que nous allons créer.

Initialisons d'abord la base de données "goterms" , puis créons son administrateur spécifique:

```
1 use goterms
2 db.createUser({user: "GoAdmin", pwd: "mongopass2", roles: ["
                                userAdminAnyDatabase"]})
```

## Création d'un utilisateur standard

Nous créons un utilisateur standard qui aura accès uniquement à la base de données "goterms" et ceci en lecture et écriture (nécessaire pour les insertions à la suite du TP).

```
1 use goterms #se placer dans la bonne base de données
2 db.createUser({user: "GoUser", pwd: "mongopass3", roles: ["readWrite"]})
```

Il faudra redémarrer MongoDB pour appliquer les changements effectués dans le fichier .ini et donc activer l'authentification et le filtrage par ip.

---

```
1 systemctl restart mongod
```

---

Pour s'authentifier, il suffira de rentrer dans la bonne base de données si besoin puis de s'authentifier avec la commande `db.auth("utilisateur", "motDePasse")`. A présent, toute opération effectuée nécessitant des privilèges supérieurs à l'utilisateur en cours affichera un message d'erreur.

## Sauvegarde et restauration des données

La base de données peut-être sauvegardée l'aide de l'utilitaire `mongodump`, avec `-u` : nom d'utilisateur, `-p` mot de passe, `-d` base de données à sauvegarder, `-c` collections à sauvegarder (par défaut : toutes) `-o` chemin d'export. Cette sauvegarde peut être simplement automatisée en ajoutant la ligne de commande à un Crontab, par exemple. La base de donnée peut être restaurée à l'aide de l'utilitaire `mongorestore`, dans le même principe.

## 2.3 Langages de manipulations de données

MongoDB est livré avec des liaisons (pilotes) pour les principaux langages de programmation : C, C++, Go, Java, Javascript, Perl, **Python** [...]. Ces pilotes permettent de manipuler la base de données et ses données directement depuis ces langages. Toutefois Javascript est intégré nativement à MongoDB est constitue donc son langage de manipulation des données par défaut, qui peut être utilisé en ligne de commande. Nous avons utilisé l'utilitaire en ligne de commande pour une utilisation standard de la base de données et le module Python `pymongo` pour une exploitation plus avancée.

## 2.4 Création du jeu de données

### Schéma de la base de données

Il est tentant d'utiliser une base de données NoSQL pour sauvegarder rapidement des données sans avoir à réfléchir au préalable à la structure des données. Mais cette simplicité est à double tranchant, car si les données sont sauvegardées de manière trop anarchique il sera très difficile

de manipuler et organiser nos données par la suite. Par exemple si on décide de stocker tous les ids au sein d'un objet.

```
{"geneProductId":"UniProtKB:P04637","qualifier":"colocalizes_with","goId":"GO:0005669"}
```

Cela peut être pratique dans un premier temps, car ça permet d'obtenir et stocker simplement les ids, sans avoir à faire des liaisons complexes. Par contre, si dans le futur nous souhaitons trier les ids par date d'insertion nous serons obligés de repenser notre base (ajouter un clé "date" ou créer des requêtes plus complexes et ainsi moins performantes). Dans notre cas d'application, nous n'allons pas réfléchir au schéma de la base de données dans la mesure ou nous en allons récupérer une dont le format à déjà été pensé.

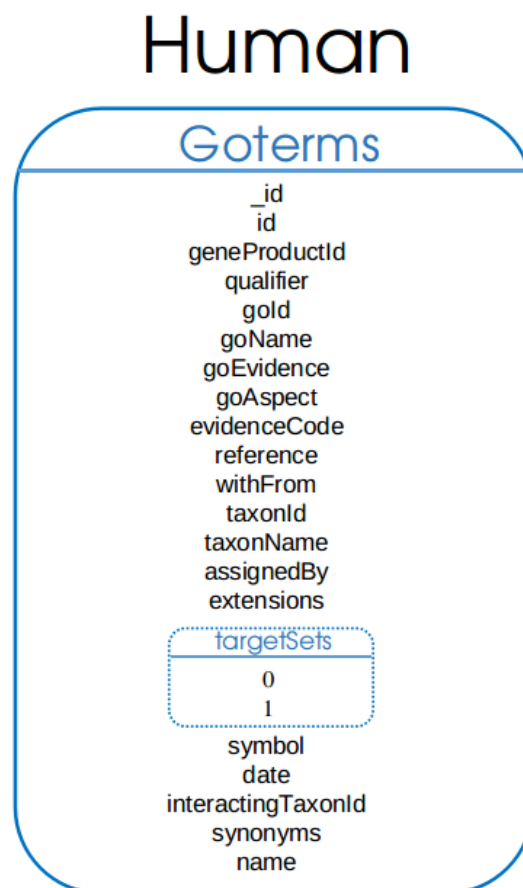


Figure 6: Schéma de la base de données MongoDB qui sera utilisée prochainement. Celle-ci est nommée "Human", contient une collection principale nommée "Goterms" qui contient l'ensemble des clés-valeurs de la base de données. La clé "targetSets" renvoie vers une sous collection (traits pointillés) contenant deux clés-valeurs.

Credits - BODINER Maxime, DURIMEL Kevin, GILLES Anthomy

## Provenance des données

La nouvelle version de [QuickGO](#) (publiée courant Juin 2017) fournit par rapport à ses précédentes versions un ensemble beaucoup plus riche et plus complet Services Web RESTful permettant d'accéder aux informations contenues dans la base de données [Uniprot-GOA](#) . La [documentation](#) de la nouvelle API permet de rapidement la prendre en main afin de constituer

aisément des requêtes sur différentes bases de données (Ontologies, annotations, genes products). La réponse modèle est la suivante :

```
1 QueryResultñAnnotationž {
2   numberOfHits (integer, optional),
3   pageInfo (PageInfo, optional),
4   results (Array[Annotation], optional)
5 }
6 PageInfo {
7   current (integer, optional),
8   resultsPerPage (integer, optional),
9   total (integer, optional)
10 }
11 Annotation {
12   assignedBy (string, optional),
13   date (string, optional),
14   evidenceCode (string, optional),
15   extensions (Array[ConnectedXRefsñQualifiedXrefž], optional),
16   geneProductId (string, optional),
17   goAspect (string, optional),
18   goEvidence (string, optional),
19   goId (string, optional),
20   goName (string, optional),
21   id (string, optional),
22   interactingTaxonId (integer, optional),
23   name (string, optional),
24   qualifier (string, optional),
25   reference (string, optional),
26   slimmedIds (Array[string], optional),
27   symbol (string, optional),
28   synonyms (string, optional),
29   targetSets (Array[string], optional),
30   taxonId (integer, optional),
31   taxonName (string, optional),
32   withFrom (Array[ConnectedXRefsñSimpleXRefž], optional)
33 }
34 ConnectedXRefsñQualifiedXrefž {
35   connectedXrefs (Array[QualifiedXref], optional)
36 }
37 ConnectedXRefsñSimpleXRefž {
38   connectedXrefs (Array[SimpleXRef], optional)
39 }
40 QualifiedXref {
41   db (string, optional),
42   id (string, optional),
43   qualifier (string, optional)
44 }
45 SimpleXRef {
46   db (string, optional),
47   id (string, optional)
```

Nous avons choisi de constituer une sous base de données d'annotations en GO-termes, extraite de la base de données "annotations", qui résume pour chaque GO-terme retrouvé les informations suivantes :

---

```

1 - id : identifiant du résultat
2 - geneProductId : identifiant Uniprot de l'élément annoté
3 - qualifieur : descriptif de la relation
4 - gold : identifiant de l'annotation GO
5 - goName : titre de l'annotation GO
6 - goEvidence : niveau de curation du GO-terme
7 - goAspect : catégorie du GO-terme (BP, MF, CC)
8 - evidenceCode : code d'évidence du GO-terme (=par quel moyen il à été inféré)
9 - reference : référence (littérature id, base de donnée id)
10 - withFrom : (null par défaut)
11 - taxonId : identifiant taxonomique de l'espèce
12 - taxonName : nom officiel de l'espèce
13 - assignedBy : base de donnée de provenance de l'annotation
14 - extensions : (null par défaut)
15 - targetSets : gene product set
16 - symbol : gene symbol officiel
17 - date : date de création
18 - interactingTaxonId : identifiant taxonomique de l'espèce interagissant
19 - synonyms : synonymes du gene symbol (symbol)
20 - name : nom complet officiel

```

---

Pour plus de légèreté nous avons réduit les données aux protéines P53 (identifiant Uniprot P04637) et BRCA2 (identifiant uniprot P51587). Pour ce faire nous avons construit la requête suivante :

---

```

1 https://www.ebi.ac.uk/QuickGO/services/annotation/search?includeFields=goName&includeFields=
  taxonName&includeFields=name&includeFields=synonyms&geneProductId=P04637,P51587

```

---

La base de données est au format décrit précédemment dans la figure 5. Les données reçues avaient un poids d'environ 350ko (Au format BSON celui-ci passe à 100ko soit un ratio de compression d'environ 1:3) :

Total character	Total word	Total lines	Size
353136	19285	8995	353.64Kb

On importe les données dans MongoDB avec la commande "mongoimport". L'import est fait dans la collection "human" de la base de données "goterms" depuis le fichier human.json :

```

1 mongoimport --db goterms --collection human --file human.json --jsonArray

```

## 2.5 Optimisation de la base de données

### Indexation interne

Les index permettent d'accélérer les requêtes en évitant au moteur de base de données de devoir scanner tous les documents de la collection afin de trouver ceux correspondant à la requête.

Afin de déterminer les index existant, on peut faire appel à la fonction getIndexes() qui va lister les index présents.

```

1 db.goterms8.getIndexes()
2 [
3   {
4     "v" : 1,
5     "key" : {
6       "_id" : 1
7     },
8     "name" : "_id_",
9     "ns" : "goterms.human"
10  }
11 ]

```

Ici, on voit que la collection contient un index sur le champ id. Celui-ci est toujours créé par défaut par le MongoDB sur l'identifiant des collections afin d'accélérer les recherches de la manière la plus globale possible (= sur toute la collection).

Pour créer un index spécifique, on utilise la fonction `createIndex()`. Ainsi, si nous voulons créer un index sur le champ "goAspect", nous procédons de la manière suivante :

```

1 db.human.createIndex({"goAspect" : 1})
2 {
3   "createdCollectionAutomatically" : false,
4   "numIndexesBefore" : 1,
5   "numIndexesAfter" : 2,
6   "ok" : 1
7 }

```

On vérifie que l'index a bien été créé :

```

1 db.human.getIndexes()
2 [
3   {
4     "v" : 1,
5     "key" : {
6       "_id" : 1
7     },
8     "name" : "_id_",
9     "ns" : "goterms.human"
10  },
11   {
12     "v" : 1,
13     "key" : {
14       "goAspect" : 1
15     },
16     "name" : "goAspect_1",
17     "ns" : "goterms.human"
18   }
19 ]

```

L'index va permettre, comme dans une base de données relationnelle, d'accélérer certaines requêtes (filtres ou tris sur le champ portant l'index). Pour supprimer un index, on utilise la



fonction dropIndex()

## Comparaison d'une requête sans index vs avec index :

```
1 db.human.explain(true).find({"goAspect" : "cellular_component"}).count()
```

Cette requête filtre les clés goAspect de la collection puis comptabilise le nombre de clés de la réponse. MongoDB nous permet d'évaluer la performance de cette requête avec la fonction explain(). Sans argument, la fonction renvoie le plan de requête qui va être exécuté. Avec true comme argument, la fonction renvoie également des informations sur l'exécution de la requête.

```
1 {
2   "queryPlanner" : {
3     "plannerVersion" : 1,
4     "namespace" : "goterms.human",
5     "indexFilterSet" : false,
6     "parsedQuery" : {
7       "goAspect" : {
8         "$eq" : "molecular_function"
9       }
10    },
11    "winningPlan" : {
12      "stage" : "COUNT",
13      "inputStage" : {
14        "stage" : "COLLSCAN",
15        "filter" : {
16          "goAspect" : {
17            "$eq" : "molecular_function"
18          }
19        },
20        "direction" : "forward"
21      }
22    },
23    "rejectedPlans" : [ ]
24  },
25  "executionStats" : {
26    "executionSuccess" : true,
27    "nReturned" : 0,
28    "executionTimeMillis" : 2,
29    "totalKeysExamined" : 0,
30    "totalDocsExamined" : 25,
31    "executionStages" : {
32      "stage" : "COUNT",
33      "nReturned" : 0,
34      "executionTimeMillisEstimate" : 2,
35      "works" : 27,
36      "advanced" : 0,
37      "needTime" : 26,
38      "needYield" : 0,
39      "saveState" : 0,
```

```

40     "restoreState" : 0,
41     "isEOF" : 1,
42     "invalidates" : 0,
43     "nCounted" : 23,
44     "nSkipped" : 0,
45     "inputStage" : {
46         "stage" : "COLLSCAN",
47         "filter" : {
48             "goAspect" : {
49                 "$eq" : "molecular_function"
50             }
51         },
52         "nReturned" : 23,
53         "executionTimeMillisEstimate" : 0,
54         "works" : 27,
55         "advanced" : 23,
56         "needTime" : 3,
57         "needYield" : 0,
58         "saveState" : 0,
59         "restoreState" : 0,
60         "isEOF" : 1,
61         "invalidates" : 0,
62         "direction" : "forward",
63         "docsExamined" : 25
64     },
65 },
66 "allPlansExecution" : [ ]
67 },
68 "serverInfo" : {
69     "host" : "kevin-Aspire-7741",
70     "port" : 27017,
71     "version" : "3.2.16",
72     "gitVersion" : "055562as28114e44c5385c8a8776fb582363e094"
73 },
74 "ok" : 1
75 }

```

La sortie obtenue nous résume les étapes de déroulement de la requête et ses performances. En première partie, le winning plan nous indique que la requête est composée de 2 étapes (stages). La première étape fait un scan de la collection pour filtrer les documents satisfaisant le prédicat. La deuxième étape fait un "collscan" des documents. Dans la deuxième partie du explain, on a des informations plus précises sur l'exécution de la requête. On peut ainsi constater que 25 documents sont examinés et que la durée de requête est estimée à 2 ms. Relançons la même commande sur la même collection indexée :

```

1 {
2     "queryPlanner" : {
3         "plannerVersion" : 1,
4         "namespace" : "goterms.human",

```

```

5      "indexFilterSet" : false,
6      "parsedQuery" : {
7          "goAspect" : {
8              "$eq" : "molecular_function"
9          }
10     },
11     "winningPlan" : {
12         "stage" : "COUNT",
13         "inputStage" : {
14             "stage" : "COUNT_SCAN",
15             "keyPattern" : {
16                 "goAspect" : 1
17             },
18             "indexName" : "goAspect_1",
19             "isMultiKey" : false,
20             "isUnique" : false,
21             "isSparse" : false,
22             "isPartial" : false,
23             "indexVersion" : 1
24         }
25     },
26     "rejectedPlans" : [ ]
27 },
28 "executionStats" : {
29     "executionSuccess" : true,
30     "nReturned" : 0,
31     "executionTimeMillis" : 1,
32     "totalKeysExamined" : 24,
33     "totalDocsExamined" : 0,
34     "executionStages" : {
35         "stage" : "COUNT",
36         "nReturned" : 0,
37         "executionTimeMillisEstimate" : 0,
38         "works" : 24,
39         "advanced" : 0,
40         "needTime" : 23,
41         "needYield" : 0,
42         "saveState" : 0,
43         "restoreState" : 0,
44         "isEOF" : 1,
45         "invalidates" : 0,
46         "nCounted" : 23,
47         "nSkipped" : 0,
48         "inputStage" : {
49             "stage" : "COUNT_SCAN",
50             "nReturned" : 23,
51             "executionTimeMillisEstimate" : 0,
52             "works" : 24,
53             "advanced" : 23,
54             "needTime" : 0,

```

```

55     "needYield" : 0,
56     "saveState" : 0,
57     "restoreState" : 0,
58     "isEOF" : 1,
59     "invalidates" : 0,
60     "keysExamined" : 24,
61     "keyPattern" : {
62         "goAspect" : 1
63     },
64     "indexName" : "goAspect_1",
65     "isMultiKey" : false,
66     "isUnique" : false,
67     "isSparse" : false,
68     "isPartial" : false,
69     "indexVersion" : 1
70 }
71 },
72 "allPlansExecution" : [ ]
73 },
74 "serverInfo" : {
75     "host" : "kevin-Aspire-7741",
76     "port" : 27017,
77     "version" : "3.2.16",
78     "gitVersion" : "055562as28114e44c5385c8a8776fb582363e094"
79 },
80 "ok" : 1
81 }

```

Cette fois-ci, la requête n'a plus besoin de scanner toute la collection : les documents ne sont plus scannés et toutes les clés n'ont pas été scannées. La recherche a été accélérée puisque la durée d'exécution de la requête est estimée à 0 ms.

## Théorème CAP

### Consistance : par support de la concurrence d'accès (C)

Contrairement aux SGBDR, la gestion de la concurrence d'accès dans MongoDB n'est pas assurée par un système de transactions. MongoDB utilise le verrouillage et d'autres mesures de contrôle de concurrence pour prévenir plusieurs clients de modifier le même fragment de données simultanément. Ensemble, ces mécanismes garantissent que toutes les écritures dans un document unique se produisent soit en totalité, soit pas du tout et que les clients n'obtiennent jamais une vue incohérente des données. Pour les rapports sur l'information concernant l'utilisation des verrous, on peut utiliser un des nombreux outils de monitoring fournis avec MongoDB (mongotop, mongostat, commandes du shell MongoDB). Par exemple, le document locks dans la sortie de db.serverStatus(), ou le champ locks dans les rapports sur l'opération courante donne un aperçu du type de verrou et la quantité de conflits de verrouillage dans votre instance mongod. Dans notre cas, depuis le démarrage de la dernière instance ( 3 heures)

```

1 "lock" : {

```

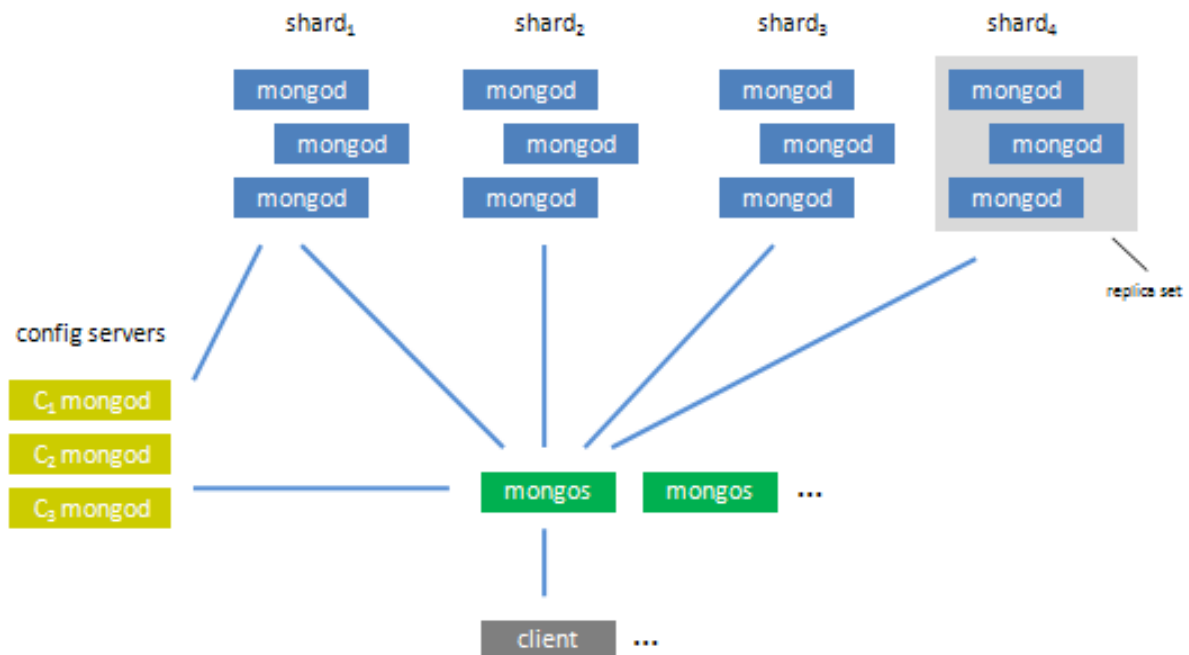
```

2      "checkpoint lock acquisitions" : 45,
3      "checkpoint lock application thread wait time (usecs)" : 0,
4      "checkpoint lock internal thread wait time (usecs)" : 0,
5      "handle-list lock eviction thread wait time (usecs)" : 1217,
6      "metadata lock acquisitions" : 45,
7      "metadata lock application thread wait time (usecs)" : 0,
8      "metadata lock internal thread wait time (usecs)" : 0,
9      "schema lock acquisitions" : 104,
10     "schema lock application thread wait time (usecs)" : 0,
11     "schema lock internal thread wait time (usecs)" : 0,
12     "table lock acquisitions" : 0,
13     "table lock application thread time waiting for the table lock (usecs
14         )" : 0,
15     "table lock internal thread time waiting for the table lock (usecs)"
        : 0
    }

```

### Tolérance au partitionnement (P) et haute disponibilité (A) : Replicaset

MongoDB met à disposition un concept de réplication des données nommé Replica Set. Sa mise en place est simple. Il permet d'améliorer la disponibilité des données et dans certains cas d'augmenter les capacités en lecture. En fonction de la configuration choisie on sacrifie la haute disponibilité (A) ou la résistance au partitionnement (P).



Chaque replicat set est aussi appelé shard. Le sharding consiste en une distribution du stockage des données sur les différentes instances Mongo au sein de ce shard. Chaque machine stocke donc un sous ensemble des données. La réplication consiste à écrire les données sur plusieurs serveurs. Il existe 5 types de shards :

**Primary** Accès en écriture seulement.

**Secondary** Réplication du shard Primary mais en lecture seulement. Il peut à son tour devenir Primary dans le cas où le vrai Primary est indisponible.

**Priority 0** Fonctionnement similaire au shard Secondary mais ne peut devenir shard Primary.

**Hidden** Fonctionnement similaire au shard Secondary mais n'est pas visible aux clients. Ce type de shard sert généralement de backup.

**Arbiter** Il ne s'agit pas d'une réplication du shard Primary.

Nous allons simuler la mise en place d'une architecture standard: 1 membre Primary et 2 membres Secondary (en production, il est recommandé d'initialiser 3 mongod au minimum dans un Replica Set).

Démarrage de chaque shard sur chacun des serveurs

```
1 mongod --replSet "rs0"
2 # Connexion au premier shard
3 mongo
4 # Initialisation du replica set
5 rs.initiate({
6     _id:"rs0",
7     members:[{
8         _id:1,
9         host: "rs.0-1.0.0.0.1:27017"
10    }]
11 })
12 # Ajout des autres shards
13 rs.add("rs0-2.0-1.0.0.0.1:27017")
14 rs.add("rs0-3.0-1.0.0.0.1:27017")
15 # Verification
16 rs.status()
17 { "ok" : 0, "errmsg" : "not running with --replSet", "code" : 76 }
```

Nous obtenons bien sur un message d'erreur puisque nous avons tenté de lancer les 3 shards en local. Dans la mesure où nous n'avons pas eu accès à des clusters, nous n'avons pas pu tester cette fonctionnalité de MongoDB.

### 3 MongoDB : Exploitation de la base de données par la pratique (requêtes) - Cas d'usages

On se propose dans ce chapitre de montrer les commandes basiques de MongoDB en illustrant nos propos d'exemples réalisés à l'aide de Python et de la librairie "pymongo". Nous avons choisi de faire les requêtes avec pymongo car la syntaxe est identique à celle du Shell MongoDB et l'utilisation de Python nous permettait de faire des analyses à la suite de nos requêtes, reflétant des cas d'utilisations typées "fouilles de données" pertinentes en bioinformatique.

Pour des raisons de lisibilité et d'ergonomie visuelle, nous vous conseillons de visualiser ce chapitre au format HTML (fichier MongoDB.html)

# MongoDB

January 23, 2018

## 1 MongoDB et Python

```
In [6]: from pymongo import MongoClient
import pandas as pd

import datetime
import pprint

from bioservices import QuickGO

from datetime import datetime
from bson.objectid import ObjectId
from collections import defaultdict
```

```
In [7]: s = QuickGO(verbose=False)
```

```
res = s.Annotation(geneProductId="P04637,P51587",includeFields="goName,taxo
```

```
In [8]: res=res["results"]
for d in res:
    d.update((k, datetime.strptime(v,"%Y%m%d")) for k, v in d.items() if k
```

Un aperçu des données.

```
In [97]: res[:2]
```

```
Out[97]: [{'_id': ObjectId('5a66e40979f75dd082aa61bb'),
  'assignedBy': 'BHF-UCL',
  'date': datetime.datetime(2009, 6, 17, 0, 0),
  'evidenceCode': 'ECO:0000314',
  'extensions': None,
  'geneProductId': 'UniProtKB:P04637',
  'goAspect': 'cellular_component',
  'goEvidence': 'IDA',
  'goId': 'GO:0005669',
  'goName': 'transcription factor TFIID complex',
  'id': 'UniProtKB:P04637!415151352',
```

```

'interactingTaxonId': 0,
'name': 'Cellular tumor antigen p53',
'qualifier': 'colocalizes_with',
'reference': 'PMID:15053879',
'symbol': 'TP53',
'synonyms': 'P53_HUMAN,TP53,P53',
'targetSets': ['BHF-UCL', 'KRUK'],
'taxonId': 9606,
'taxonName': 'Homo sapiens',
'withFrom': None},
{'_id': ObjectId('5a66e40979f75dd082aa61bc'),
'assignedBy': 'BHF-UCL',
'date': datetime.datetime(2009, 12, 11, 0, 0),
'evidenceCode': 'ECO:0000314',
'extensions': None,
'geneProductId': 'UniProtKB:P04637',
'goAspect': 'cellular_component',
'goEvidence': 'IDA',
'goId': 'GO:0016604',
'goName': 'nuclear body',
'id': 'UniProtKB:P04637!415151353',
'interactingTaxonId': 0,
'name': 'Cellular tumor antigen p53',
'qualifier': 'colocalizes_with',
'reference': 'PMID:10360174',
'symbol': 'TP53',
'synonyms': 'P53_HUMAN,TP53,P53',
'targetSets': ['BHF-UCL', 'KRUK'],
'taxonId': 9606,
'taxonName': 'Homo sapiens',
'withFrom': None}]

```

### 1.0.1 MongoDB 1,2

**Connexion au serveur** Tout d’abord, exécuter le client mongo depuis votre terminal. Par défaut, il se connecte au serveur mongod en localhost sur le port 27017.

```
In [9]: client = MongoClient('localhost:27017')
        client
```

```
Out [9]: MongoClient(host=['localhost:27017'], document_class=dict, tz_aware=False,
```

**Création d’une base de donnée** Une fois connecté au serveur, il est possible d’accéder aux bases de données. MongoDB crée des bases et des collections automatiquement si elles n’existent pas déjà. Une seule instance MongoDB peut gérer de multiples bases de données indépendantes. En utilisant PyMongo, on accède ou accède au bases de données de la façon suivante:

```
In [10]: db = client['GoTerm_database']
         db
```



```
Out[10]: Database(MongoClient(host=['localhost:27017'], document_class=dict, tz_aware=False), namespace='test')
```

Il est possible de connaître les bases de données disponibles en utilisant la méthode `database_names()`. NB: les bases de données sans collections ou contenant des collections vides ne seront pas affichées.

```
In [11]: client.database_names()
```

```
Out[11]: ['GoTerm_database', 'admin', 'config', 'local']
```

**Les collections** Une collection est un ensemble de documents et peut être vue comme une table dans une base de données relationnelle. Les collections et les bases sont créées de façon “lazy” c’est-à-dire quand le premier document est inséré. Enfin, MongoDB intègre des index notés `_id` unique pour chaque document.

```
In [12]: collection = db['ontology']
         db.collection_names()
```

```
Out[12]: ['ontology']
```

**Insertion de documents** Afin d’insérer un document dans une collection, nous pouvons utiliser la méthode `insert_one()`:

```
In [99]: post = res[0]
         collection.insert_one(post)
         collection
```

De plus, afin d’insérer une série de documents il est possible d’utiliser la méthode `insert_many()` qui prend en argument une liste. Cela insérera chaque document contenu dans la liste.

```
In [179]: collection.insert_many(res)
```

```
Out[179]: <pymongo.results.InsertManyResult at 0xb2ab308>
```

## 1.0.2 Requête les données

**Lecture** L’une des principales opérations qu’il est nécessaire de connaître est de retrouver les données contenues dans les bases. L’objet connexion fournit alors 2 méthodes `find_one()` et `find()`:  
- La méthode `find_one()` extrait et retourne un unique document de la collection. Elle est utile quand nous souhaitons n’extraire qu’un résultat, ou le premier résultat matchant la condition passée en argument:

```
In [18]: pprint.pprint(collection.find_one())

{'_id': ObjectId('5a66e40979f75dd082aa61bb'),
 'assignedBy': 'BHF-UCL',
 'date': datetime.datetime(2009, 6, 17, 0, 0),
 'evidenceCode': 'ECO:0000314',
 'extensions': None,
```

```

'geneProductId': 'UniProtKB:P04637',
'goAspect': 'cellular_component',
'goEvidence': 'IDA',
'goId': 'GO:0005669',
'goName': 'transcription factor TFIID complex',
'id': 'UniProtKB:P04637!415151352',
'interactingTaxonId': 0,
'name': 'Cellular tumor antigen p53',
'qualifier': 'colocalizes_with',
'reference': 'PMID:15053879',
'symbol': 'TP53',
'synonyms': 'P53_HUMAN, TP53, P53',
'targetSets': ['BHF-UCL', 'KRUK'],
'taxonId': 9606,
'taxonName': 'Homo sapiens',
'withFrom': None}

```

- La méthode `find()` fonctionne comme la méthode `find_one()` tandis qu'elle retourner tous les documents matchant la condition passée en argument. La commande ci-dessous renverrait tous les documents (et itérerait sur chacun d'entre-eux) si nous n'avions pas commandé l'extraction des deux premiers.

```

In [19]: for post in collection.find()[:2]:
          pprint.pprint(post)

{'_id': ObjectId('5a66e40979f75dd082aa61bb'),
 'assignedBy': 'BHF-UCL',
 'date': datetime.datetime(2009, 6, 17, 0, 0),
 'evidenceCode': 'ECO:0000314',
 'extensions': None,
 'geneProductId': 'UniProtKB:P04637',
 'goAspect': 'cellular_component',
 'goEvidence': 'IDA',
 'goId': 'GO:0005669',
 'goName': 'transcription factor TFIID complex',
 'id': 'UniProtKB:P04637!415151352',
 'interactingTaxonId': 0,
 'name': 'Cellular tumor antigen p53',
 'qualifier': 'colocalizes_with',
 'reference': 'PMID:15053879',
 'symbol': 'TP53',
 'synonyms': 'P53_HUMAN, TP53, P53',
 'targetSets': ['BHF-UCL', 'KRUK'],
 'taxonId': 9606,
 'taxonName': 'Homo sapiens',
 'withFrom': None}
{'_id': ObjectId('5a66e40979f75dd082aa61bc'),

```

```

'assignedBy': 'BHF-UCL',
'date': datetime.datetime(2009, 12, 11, 0, 0),
'evidenceCode': 'ECO:0000314',
'extensions': None,
'geneProductId': 'UniProtKB:P04637',
'goAspect': 'cellular_component',
'goEvidence': 'IDA',
'goId': 'GO:0016604',
'goName': 'nuclear body',
'id': 'UniProtKB:P04637!415151353',
'interactingTaxonId': 0,
'name': 'Cellular tumor antigen p53',
'qualifier': 'colocalizes_with',
'reference': 'PMID:10360174',
'symbol': 'TP53',
'synonyms': 'P53_HUMAN, TP53, P53',
'targetSets': ['BHF-UCL', 'KRUK'],
'taxonId': 9606,
'taxonName': 'Homo sapiens',
'withFrom': None}

```

**Comptage** Combien de document il y a-t-il dans la collection? Combien de document correspondent à la requête? La méthode `count()` réponds à ces questions. Nous comptons ci-dessous combien de documents contient la collection.

```
In [188]: collection.count()
```

```
Out[188]: 100
```

**Conditions** Les requêtes MongoDB sont représentées comme une structure JSON. Pour construire une requête, il suffit juste de spécifier un dictionnaire contenant les propriétés que nous souhaitons retrouver.

```
In [194]: mf=collection.find({"goAspect": "molecular_function"}).count()
          cc=collection.find({"goAspect": "cellular_component"}).count()
          bp=collection.find({"goAspect": "biological_process"}).count()
          print(" Il y a {} GO term annotés  Molecular Function, {} Cellular Component et {} Biological Process")
```

```
Il y a 98 GO term annotés  Molecular Function, 2 Cellular Component et 0 Biological Process
```

**Les opérateurs de requêtes** Les requêtes peuvent également faire appel aux opérateurs. Ces opérateurs sont notamment `gt`, `gte`, `lt`, `lte`, `ne`, `nin`, `regex`, `exists`, `not`, `or`, et bien d'autres . La requête suivante ira extraire tous les GO term antérieur au 11 Juillet 2007.

*NB: Nous avons converti les éléments dates du json afin d'en faire des objets `datetime`, aisément manipulable en python*

```
In [197]: date = datetime.strptime("11/07/2007", "%d/%m/%Y")
         _=collection.find({"date": {"$lt":date}}).count()
         print("Il y a {} GO term antérieurs au 11 Juillet 2007".format(_))
```

Il y a 16 GO term antérieurs au 11 Juillet 2007

Extraction des GO term dont le “qualifier” n’est pas “enables”

```
In [200]: for post in collection.find({"qualifier":{"$nin":["enables"]} }):
         pprint.pprint(post)
```

```
{'_id': ObjectId('5a6618bd79f75da856687485'),
 'assignedBy': 'BHF-UCL',
 'date': datetime.datetime(2009, 6, 17, 0, 0),
 'evidenceCode': 'ECO:0000314',
 'extensions': None,
 'geneProductId': 'UniProtKB:P04637',
 'goAspect': 'cellular_component',
 'goEvidence': 'IDA',
 'goId': 'GO:0005669',
 'goName': 'transcription factor TFIID complex',
 'id': 'UniProtKB:P04637!415151352',
 'interactingTaxonId': 0,
 'name': 'Cellular tumor antigen p53',
 'qualifier': 'colocalizes_with',
 'reference': 'PMID:15053879',
 'symbol': 'TP53',
 'synonyms': 'P53_HUMAN,TP53,P53',
 'targetSets': ['BHF-UCL', 'KRUK'],
 'taxonId': 9606,
 'taxonName': 'Homo sapiens',
 'withFrom': None}
{'_id': ObjectId('5a6618bd79f75da856687486'),
 'assignedBy': 'BHF-UCL',
 'date': datetime.datetime(2009, 12, 11, 0, 0),
 'evidenceCode': 'ECO:0000314',
 'extensions': None,
 'geneProductId': 'UniProtKB:P04637',
 'goAspect': 'cellular_component',
 'goEvidence': 'IDA',
 'goId': 'GO:0016604',
 'goName': 'nuclear body',
 'id': 'UniProtKB:P04637!415151353',
 'interactingTaxonId': 0,
 'name': 'Cellular tumor antigen p53',
 'qualifier': 'colocalizes_with',
 'reference': 'PMID:10360174',
 'symbol': 'TP53',
```

```
'synonyms': 'P53_HUMAN,TP53,P53',
'targetSets': ['BHF-UCL', 'KRUK'],
'taxonId': 9606,
'taxonName': 'Homo sapiens',
'withFrom': None}
```

### 1.0.3 Aggregation 1,2,3

Les opérations d’aggrégations s’exécutent directement sur les données et retourne un résultat de type collection. Ces opérations “regroupent” ensembles les valeurs de multiples documents, et peuvent également procéder à une large variété d’opérations. Les plus pertinentes sont présentées ici. ##### \$group Permet de faire un group by SQL et ainsi de retrouver le nombre de X disponibles pour chaque Y.

```
In [71]: pipeline = [{"$group": {"_id": "$goAspect", "count": {"$sum": 1}}}]
pprint.pprint(list(collection.aggregate(pipeline)))

[{'_id': 'molecular_function', 'count': 98},
 {'_id': 'cellular_component', 'count': 2}]
```

**\$match** Recherche dans la collection. Dans ce cas nous extrayons les GO term “Molecular Function” puis nous les regroupons par goName afin de procéder à un comptage.

```
In [84]: pipeline = [{
    "$match": {
        "goAspect": "molecular_function"
    }
}, {"$group": {"_id": "$goName", "count": {"$sum": 1}}}]
pprint.pprint(list(collection.aggregate(pipeline)))

[{'_id': 'protein binding', 'count': 69},
 {'_id': 'mRNA 3'-UTR binding', 'count': 1},
 {'_id': 'DNA binding transcription factor activity', 'count': 4},
 {'_id': 'chromatin binding', 'count': 2},
 {'_id': 'transcriptional activator activity, RNA polymerase II transcription '
    'regulatory region sequence-specific DNA binding',
    'count': 1},
 {'_id': 'transcription factor activity, TFIID-class binding', 'count': 1},
 {'_id': 'core promoter sequence-specific DNA binding', 'count': 2},
 {'_id': 'TFIID-class transcription factor binding', 'count': 1},
 {'_id': 'transcription cofactor binding', 'count': 1},
 {'_id': 'RNA polymerase II transcription factor activity, sequence-specific '
    'DNA binding',
    'count': 2},
 {'_id': 'p53 binding', 'count': 2},
 {'_id': 'damaged DNA binding', 'count': 1},
```

```
{'_id': 'copper ion binding', 'count': 1},
{'_id': 'RNA polymerase II transcription factor binding', 'count': 1},
{'_id': 'transcription factor activity, core RNA polymerase binding',
 'count': 1},
{'_id': 'RNA polymerase II regulatory region sequence-specific DNA binding',
 'count': 1},
{'_id': 'DNA binding', 'count': 5},
{'_id': 'transcription factor activity, RNA polymerase II proximal promoter '
 'sequence-specific DNA binding involved in preinitiation complex '
 'assembly',
 'count': 1},
{'_id': 'protease binding', 'count': 1}]
```

**\$project** Permet de remodeler une collection. Dans le cas ci-dessous nous nous contentons de mettre en majuscule tous les identifiants “qualifier” dont le goAspect est *cellular\_component*, mais n’importe qu’elle fonction peut être appliquée sur les données.

```
In [95]: pipeline=[
        {
            "$match": {
                "goAspect": "cellular_component"
            }
        },
        { "$project":
            {
                "_id": "$goAspect",
                "qualifier": {"$toUpper": "$qualifier" },
            }
        }
    ]
    pprint.pprint(list(collection.aggregate(pipeline)))

[{'_id': 'cellular_component', 'qualifier': 'COLOCALIZES_WITH'},
 {'_id': 'cellular_component', 'qualifier': 'COLOCALIZES_WITH'}]
```

```
In [72]: collection.find().count()
```

```
Out[72]: 100
```

#### 1.0.4 Tri

MongoDB peut trier les requêtes.

```
In [30]: results=collection.find().sort([("date", -1)])
        for post in results[:2]:
            print(post,end="\n\n")
```

```
{'_id': ObjectId('5a6618bd79f75da8566874a4'), 'id': 'UniProtKB:P04637!415151383', '
{'_id': ObjectId('5a6618bd79f75da8566874a5'), 'id': 'UniProtKB:P04637!415151384', '

```

### 1.0.5 Mise à jour

**Mettre à jour un document d’une collection** PyMongo peut mettre à jour les documents d’une collection. Pour voir le nombre de documents qui ont matché la condition, l’attribut *matched\_count* peut être utilisé sur l’objet. En revanche pour voir le nombre de document *mis à jour* par l’opération, l’attribut *modified\_count* peut être utilisé.

```
In [45]: result=collection.update_one({'id': 'UniProtKB:P04637!415151353'},
    {
        "$set":{
            'goName': 'Changement de GoName'
        }
    })
    result.matched_count
    result.modified_count

    for post in collection.find({'id': 'UniProtKB:P04637!415151353'}):
        print(post)

{'_id': ObjectId('5a6618bd79f75da856687486'), 'id': 'UniProtKB:P04637!415151353', '

```

### Mettre à jour plusieurs documents d’une collection

```
In [50]: result=collection.update_many({'goAspect': 'cellular_component'},
    {
        "$set":{
            'taxonName': 'Homo erectus'
        }
    })
    print("Matched count {}".format(result.matched_count))
    print("Modified count {}".format(result.modified_count))

    for post in collection.find({'goAspect': 'cellular_component'})[:10]:
        print(post)

Matched count 2
Modified count 0
{'_id': ObjectId('5a6618bd79f75da856687485'), 'id': 'UniProtKB:P04637!415151352', '
{'_id': ObjectId('5a6618bd79f75da856687486'), 'id': 'UniProtKB:P04637!415151353', '

```

## Remplacer un document d'une collection

```
In [28]: result = collection.replace_one(
        {"id": "UniProtKB:P04637!415151353"},
        {
            "nom": "Master BIMS",
            "adresse": {
                "coord": [-73.9557413, 40.7720266],
                "street": "Place Emile Blondel",
                "zipcode": "76"
            }
        }
    )
    print("Matched count {}".format(result.matched_count))
    print("Modified count {}".format(result.modified_count))
```

Matched count 1

Modified count 1

```
In [34]: for post in collection.find({"_id": ObjectId('5a66e40979f75dd082aa61bb')}):
        print(post)
```

```
{'_id': ObjectId('5a66e40979f75dd082aa61bb'), 'nom': 'Master BIMS', 'adresse': {'coord': [-73.9557413, 40.7720266], 'street': 'Place Emile Blondel', 'zipcode': '76'}}
```

### 1.0.6 Indexation

Comme dans MySQL, les index permettent d'accélérer les requêtes en évitant au moteur de base de données de devoir scanner tous les documents de la collection afin de trouver ceux correspondant à la requête.

```
In [25]: collection.create_index([('goId', 'text')], name='goId_index')
        collection.index
```

```
Out [25]: Collection(Database(MongoClient(host=['localhost:27017'], document_class=Dict)))
```

### 1.0.7 Suppression

#### Supprimer tous les documents matchant une condition

```
In [59]: result = collection.delete_many({'goAspect': 'cellular_component'})
        result.deleted_count
```

```
Out [59]: 1
```



## Supprimer tous les documents

```
In [14]: result = collection.delete_many({})
         result.deleted_count
```

```
Out[14]: 100
```

```
In [59]: collection.find().count()
```

```
Out[59]: 100
```

## Supprimer une collection

```
In [13]: collection.drop()
```

## 1.0.8 Représentations de la collection

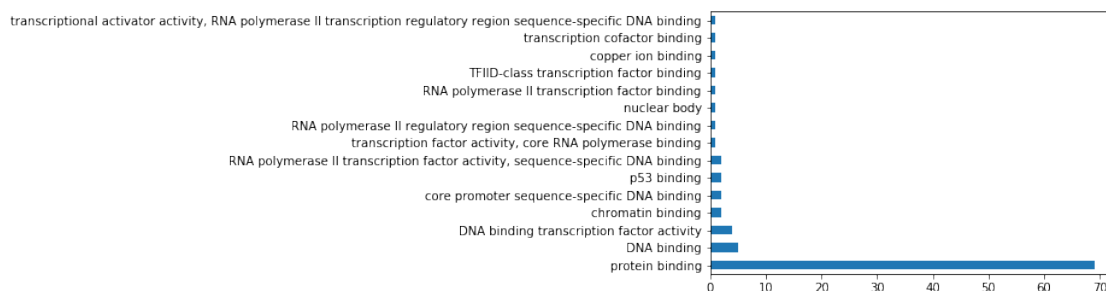
La possibilité d'exploiter MongoDB avec Python nous permet également de directement manipuler les bases de données puis de les analyser (analyses statistiques, visualisation, etc). Par exemple, nous pouvons explorer notre collection en utilisant le module pandas.

```
In [15]: %matplotlib inline
```

Et voir les GO les plus représentés dans la base de données.

```
In [16]: df = pd.DataFrame(list(collection.find()))
         df["goName"].value_counts()[ :15].plot(kind="barh")
```

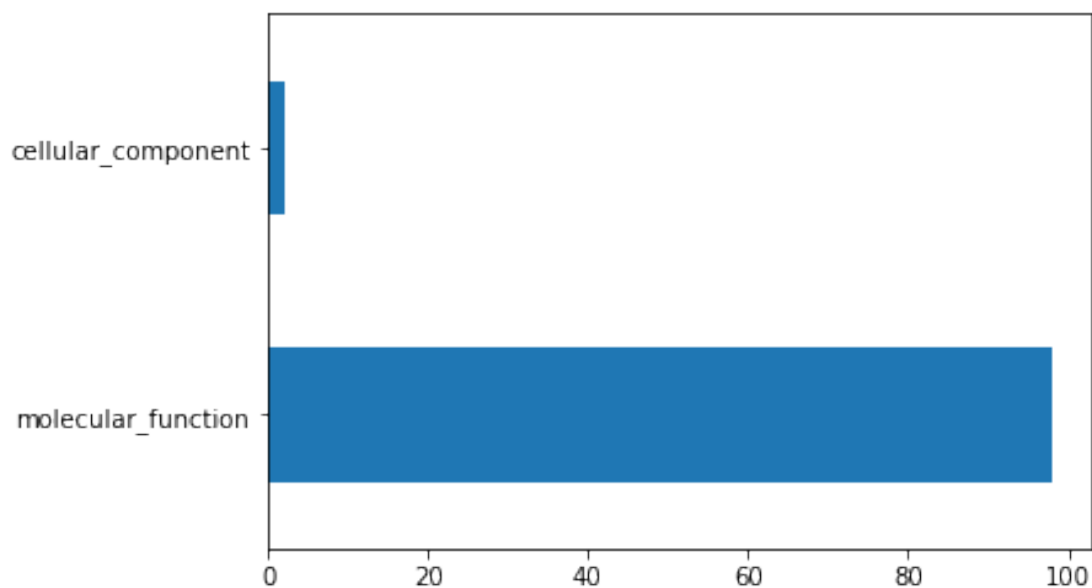
```
Out[16]: <matplotlib.axes._subplots.AxesSubplot at 0xae5cc0>
```



Ou visualiser rapidement la répartition des GO Aspect.

```
In [17]: df["goAspect"].value_counts().plot(kind="barh")
```

```
Out[17]: <matplotlib.axes._subplots.AxesSubplot at 0xb579e80>
```



```
In [73]: df.head(1)
```

```
Out [73]:
```

	_id	assignedBy	date	evidenceCode	extensions
0	5a66e40979f75dd082aa61bb	BHF-UCL	2009-06-17	ECO:0000314	None

	geneProductId	goAspect	goEvidence	goId	\
0	UniProtKB:P04637	cellular_component	IDA	GO:0005669	

	goName	...	interactingTaxonId	\
0	transcription factor TFIID complex	...		0

	name	qualifier	reference	symbol	\
0	Cellular tumor antigen p53	colocalizes_with	PMID:15053879	TP53	

	synonyms	targetSets	taxonId	taxonName	withFrom
0	P53_HUMAN, TP53, P53	[BHF-UCL, KRUK]	9606	Homo sapiens	None

[1 rows x 21 columns]

# Bibliography

- [1] *Bigtable Service de base de données NoSQL évolutif / Google Cloud Platform*. URL: <https://cloud.google.com/bigtable/?hl=fr> (visited on 01/13/2018).
- [2] *Bloom filters*. URL: [https://en.wikipedia.org/wiki/Bloom\\_filter](https://en.wikipedia.org/wiki/Bloom_filter) (visited on 01/18/2018).
- [3] *MongoDB built-in roles*. URL: <https://docs.mongodb.com/manual/reference/built-in-roles/> (visited on 01/03/2018).
- [4] *MongoDB Security Best Practices / February 2015 / MongoDB*. URL: <https://www.mongodb.com/blog/post/mongodb-security-best-practices> (visited on 12/26/2017).
- [5] *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. 2012. DOI: 0321826620.
- [6] *Security Checklist MongoDB Manual 3.6*. URL: <https://docs.mongodb.com/manual/administration/security-checklist/> (visited on 12/26/2017).
- [7] *The Neo4j Developer Manual v3.3*. URL: <http://neo4j.com/docs/developer-manual/current/cypher/schema/index/%7B%5C#%7Dquery-schema-index-introduction> (visited on 01/10/2018).
- [8] *Webinar: Architecting Secure and Compliant Applications with MongoDB / MongoDB*. URL: <https://www.mongodb.com/presentations/webinar-architecting-secure-and-compliant-applications-with-mongodb> (visited on 01/19/2018).