Comment gérer une énorme base de données et comment l'interroger efficacement ? Ces questions, on se les pose dès que le volume devient ingérable et que répondre à de simples requêtes prend des heures.

Oubliez les SGBD traditionnels, ils peinent à passer à l'échelle ! Vous devez être capable de choisir la bonne solution parmi les dizaines qui s'offrent à vous.

Dans ce cours, vous découvrirez l'univers du NoSQL. Nous ferons un focus sur une solutions NoSQL extrêmement populaire qui est **MongoDb**. Vous apprendrez à stocker et à réaliser des requêtes sur vos données tout en assurant le passage à l'échelle.

Table des matières

Partie 1 - Immergez vos données dans le NoSQL

1. Choisissez votre famille NoSQL

♦ c'est quoi le NoSQL ?

Depuis les années 70, la base de données relationnelle était l'incontournable référence pour gérer les données d'un système d'information. Toutefois, face aux 3V (Volume, Velocity, Variety), le relationnel peut difficilement lutter contre cette vague de données. Le NoSQL s'est naturellement imposé dans ce contexte en proposant une nouvelle façon de gérer les données, sans reposer sur le paradigme relationnel, d'où le "Not Only SQL". Cette approche propose de relâcher certaines contraintes lourdes du relationnel pour favoriser la distribution (structure des données, langage d'interrogation ou la cohérence).

Dans un contexte bases de données, il est préférable d'avoir un langage de haut niveau pour interroger les données plutôt que tout exprimer en Map/Reduce. Toutefois, avoir un langage de trop haut niveau comme SQL ne facilite pas la manipulation. Et c'est en ce sens que l'on peut parler de "Not Only SQL", d'autres solutions peuvent être proposées pour résoudre le problème de distribution. Ainsi, le NoSQL est à la fois une autre manière d'interroger les données, mais aussi de les stocker.

Les besoins de stockage et de manipulation dans le cadre d'une base de données sont variables et dépendent principalement de l'application que vous souhaitez intégrer. Pour cela, différentes familles de bases NoSQL existent : Clé/Valeur, colonnes, documents, graphes. Chacune de ces familles répond à des besoins très spécifiques que nous allons développer par la suite.

♦ Les clés-valeurs

Le but de la famille clé-valeur est l'efficacité et la simplicité. Un système clé-valeur agit comme une énorme table de hachage distribuée sur le réseau. Tout repose sur le couple Clé/Valeur. La clé identifie la donnée de manière unique et permet de la gérer. La valeur contient n'importe quel type de données.

Le fait d'avoir n'importe quoi implique qu'il n'y ait ni schéma, ni structure pour le stockage. D'un point de vue de bases de données, il n'y a pas la possibilité d'exploiter ni de contrôler la structure des données et de fait, pas de langage (SQL = Structured Query Language). En soit ce n'est pas un problème si vous savez ce que vous cherchez (la clé) et que vous manipulez directement la valeur.

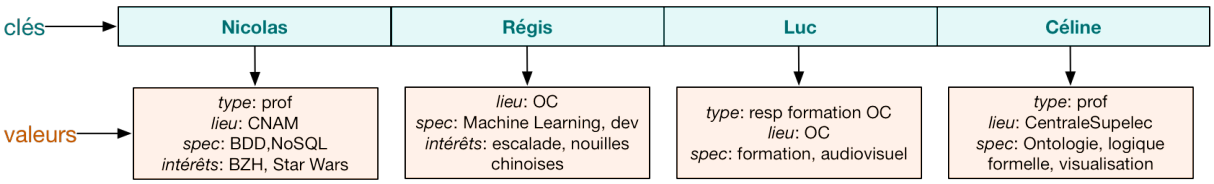


Figure 1 : Stockage Clé/Valeur

Les seules opérations de type CRUD peuvent être utilisées :

- Create (key,value)
- Read (key)
- Update (key,value)

☛ Quels types d'applications pourraient correspondre à cette solution ? Voici quelques idées qui pourraient orienter votre choix : *détection de fraude en temps réel, IoT, e-commerce, gestion de cache, transactions rapides, fichiers de logs, chat.*

Qui s'intéresse à l'orienté clé-valeur ?

- **Redis** (VMWare) : Vodafone, Trip Advisor, Nokia, Samsung, Docker
- **Memcached** (Danga) : LiveJournal, Wikipédia, Flickr, Wordpress
- **Azure Cosmos DB** (Microsoft) : Real Madrid, Orange tribes, MSN, LG, Schneider Electric
- **SimpleDB** (Amazon)

♦ Des lignes vers les colonnes

Traditionnellement, les données sont représentées en ligne, représentant l'ensemble des attributs. Le stockage orienté colonne change ce paradigme en se focalisant sur chaque attribut et en les distribuant. Il est alors possible de focaliser les requêtes sur une ou plusieurs colonnes, sans avoir à traiter les informations inutiles (les autres colonnes).

Stockage orienté lignes

id	type	lieu	spec	intérêts
Nicolas	prof	CNAM	BDD, NoSQL	BZH, Star Wars
Régis		OC	Machine Learning, Dev	escalade, nouilles chinoises
Luc	resp formation OC	OC	formation, audiovisuel	
Céline	prof	CentraleSupelec	Ontologie, logique formelle, visualisation	

Stockage orienté colonnes

id	type
Nicolas	prof
Céline	prof
Luc	resp formation OC

id	lieu
Céline	Centrale Supelec
Nicolas	CNAM
Régis	OC
Luc	OC

id	spec
Nicolas	BDD
Nicolas	NoSQL
Régis	Machine Learning
Régis	Dev
Luc	formation
Luc	audiovisuel
Céline	Ontologie
Céline	logique formelle
Céline	visualisation

id	intérêts
Nicolas	BZH
Nicolas	Star Wars
Régis	escalade
Régis	nouilles chinoises

Figure 2: Stockage orienté colonnes

Cette solution est très adaptée pour effectuer des traitements sur des colonnes comme les agrégats (comptage, moyennes, co-occurences...). D'une manière plus concrète, elle est adaptée à de gros calculs analytiques. Toutefois, cette solution est beaucoup moins appropriée pour la lecture de données spécifiques comme pour les clés/valeurs.

☛ Quelques exemples d'applications : Comptage (vote en ligne, compteur, etc), journalisation, recherche de produits dans une catégorie, reporting à large échelle.

Qui s'intéresse à l'orienté colonnes?

- **BigTable** (Google)
- **HBase** (Apache, Hadoop)
- **Spark SQL** (Apache)
- **Elasticsearch** (elastic) -> moteur de recherche

♦ La gestion de documents

Les bases orientées documents ressemblent sans doute le plus à ce que l'on peut faire dans une base de données classique pour des requêtes complexes. Le but de ce stockage est de manipuler des documents contenant des informations avec une structure complexe (types, listes, imbrications). Il repose sur le principe du clé/valeur, mais avec une extension sur les champs qui composent ce document.

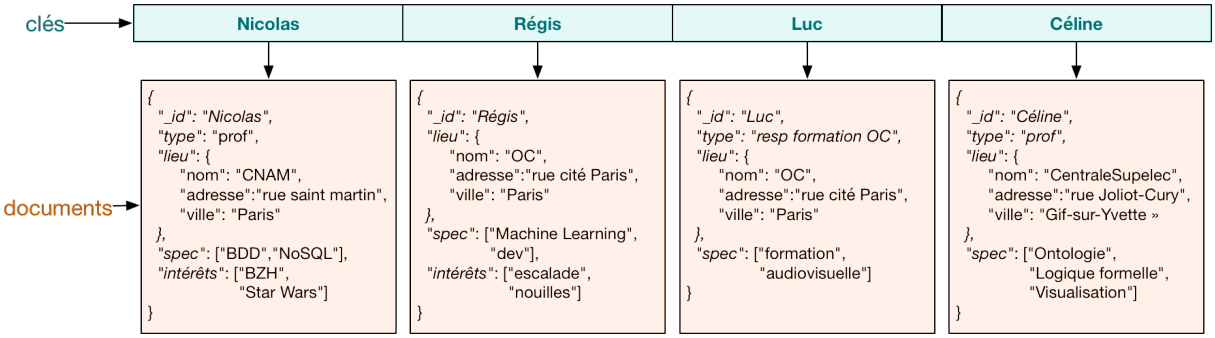


Figure 3 : Stockage orienté document

L'avantage de cette solution est d'avoir une approche structurée de chaque valeur, formant ainsi un document. De fait, ces solutions proposent des langages d'interrogation riches permettant de faire des manipulations complexes sur chaque attribut du document (et sous-documents) comme dans une base de données traditionnelles, tout en passant à l'échelle dans un contexte distribué.

Cassandra est souvent considérée comme une solution orientée colonnes. C'est une solution orientée documents appelée "wide-column store", dans le sens où le document est structuré comme en relationnel. La confusion vient du fait que l'on y définit des colonnes et des familles de colonnes dans le schéma, ce qui est différent du stockage de "colonnes".

Quelques exemples d'utilisation : *gestion de contenu (bibliothèques numériques, collections de produits, dépôts de logiciels, collections multimédia, etc.), framework stockant des objets, collection d'événements complexes, gestion des historiques d'utilisateurs sur réseaux sociaux.*

Quelles solutions pour gérer vos documents

- **MongoDB** (*MongoDB*) : ADP, Adobe, Bosch, Cisco, eBay, Electronic Arts, Expedia, Foursquare
- **CouchBase** (Apache, Hadoop) : AOL, AT&T, Comcast, Disney, PayPal, Ryanair
- **DynamoDB** (Amazon) : BMW, Dropcam, Duolingo, Supercell, Zynga
- **Cassandra** (Facebook -> Apache) : NY Times, eBay, Sky, Pearson Education

♦ Et les graphes ?

Les trois premières familles NoSQL n'adressent pas le problème de corrélations entre les éléments. Prenons l'exemple d'un réseau social : dans certains cas, il devient très complexe de calculer la distance entre deux personnes non directement connectées. Et c'est ce type d'approche que résolvent les bases orientées Graphe.

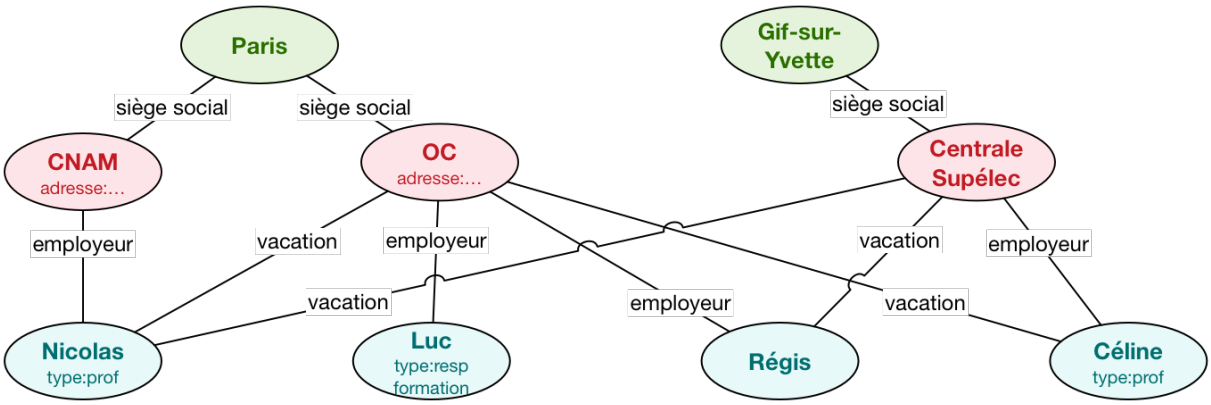


Figure 4 : Stockage orienté graphe

Dans la base orientée graphe, les données stockées sont : les nœuds, les liens et des propriétés sur ces nœuds et ces liens. Les requêtes que l'on peut exprimer sont basées sur la gestion de chemins, de propagations, d'agrégations, voire de recommandations. Toutefois, contrairement aux solutions précédentes la distribution des nœuds sur le réseau n'est pas triviale.

Quelques idées d'applications reposant sur des bases orientées graphes : *réseaux sociaux (recommandation, plus court chemin, cluster...), réseaux SIG (routes, réseau électrique, ...), web social (Linked Data).*

Quelles bases de données gèrent de gros graphes ?

- **Neo4j**: eBay, Cisco, UBS, HP, TomTom, The National Geographic Society
- **OrientDB** (*Apache*): Comcast, Warner Music Group, Cisco, Sky, United Nations, VErisign
- **FlockDB** (Twitter): Twitter

2. Maitrisez le théorème de CAP

♦ Propriétés fondamentales d'une base de données relationnelle

---

Et pourquoi donc faire encore de la base de données relationnelle ? Le NoSQL est très tentant lorsque l'on voit toutes les possibilités qu'il peut offrir. Mais il ne faut pas négliger certains fondamentaux qui rendent les SGBDR incontournables.

Cela se résume dans une combinaison qu'il ne faut pas négliger :

1. Faire des jointures entre les tables de la base de données
2. Faire des requêtes complexes avec un langage de haut niveau sans se préoccuper des couches basses
3. Gérer l'intégrité des données de manière infaillible

Et c'est surtout ce dernier point qui attire l'attention dans le cadre du NoSQL. Gérer l'intégrité des données veut dire être capable de garantir le contenu de la base de données, quelles que soient les mises à jour effectuées sur les données, ainsi que sa pérennité. Nous allons voir que la gestion de la cohérence d'une donnée n'est pas forcément garantie.

ACID vs BASE

Pour mettre en rapport les problématiques de la base de données relationnelle, on parle de propriétés **ACID** pour les transactions (séquences d'opérations/requêtes) :

- **A***tomicité* : Une transaction s'effectue entièrement ou pas du tout
- **C***ohérence* : Le contenu d'une base doit être cohérent au début et à la fin d'une transaction
- **I***solation* : Les modifications d'une transaction ne sont visibles/modifiables que quand celle-ci a été validée
- **D***urabilité* : Une fois la transaction validée, l'état de la base est permanent (non affecté par les pannes ou autre)

Toutefois, ces propriétés ne sont pas applicables dans un contexte distribué tel que le NoSQL. En effet, prenons l'exemple d'une transaction de cinq opérations (lecture/écriture) : cela implique une synchronisation entre cinq serveurs pour garantir l'atomicité, la cohérence et l'isolation. Au final, cela se traduit par des latences dans les transactions (en cours et en concurrence). Ce qui n'est pas tolérable lorsque justement on veut éviter ces latences en distribuant les calculs.

Le problème s'aggrave encore lorsque l'on distribue les données car il va falloir répliquer chaque donnée. Pourquoi ? Tout simplement parce que si un serveur tombe en panne, il faut pouvoir garantir de retrouver toutes les données présentes sur ce serveur, donc on fait de la réplication. Mais cela veut dire également qu'il va falloir synchroniser toutes mises à jour avec tous les réplicas de la donnée !

Figure 5: ACID vs BASE

Du coup, les propriétés **BASE** ont été proposées pour caractériser les bases NoSQL :

- **Ba***sically Available* : quelle que soit la charge de la base de données (données/requêtes), le système garantie un taux de disponibilité de la donnée
- **S***oft-state* : La base peut changer lors des mises à jour ou lors d'ajout/suppression de serveurs. La base NoSQL n'a pas à être cohérente à tout instant
- **E***ventually consistent* : À terme, la base atteindra un état cohérent

Ainsi, une base NoSQL relâche certaines contraintes, telles que la synchronisation des réplicas, pour favoriser l'efficacité. Le parallèle ACID / BASE repris du domaine de la chimie permet d'appuyer là où ça fait mal : la concurrence. L'*enfer* des transactions gérées par les bases de données relationnelles est transformé en *paradis* pour le temps de réponse en relâchant cette contrainte impossible à maintenir.

Théorème de Brewer dit "théorème de CAP"

En 2000, Eric A. Brewer a formalisé un théorème très intéressant reposant sur 3 propriétés fondamentales pour caractériser les bases de données (relationnelles, NoSQL et autres) :

1. **C***onsistency* (Cohérence) : Une donnée n'a qu'un seul état visible quel que soit le nombre de réplicas
2. **A***vailability* (Disponibilité) : Tant que le système tourne (distribué ou non), la donnée doit être disponible
3. **P***artition Tolerance* (Distribution) : Quel que soit le nombre de serveurs, toute requête doit fournir un résultat correct

Le **théorème de CAP** dit :

Dans toute base de données, vous ne pouvez respecter au plus que 2 propriétés parmi la *cohérence*, la *disponibilité* et la *distribution*.

Cela s'illustre assez facilement avec les bases de données relationnelles, elles gèrent la cohérence et la disponibilité, mais pas la distribution.

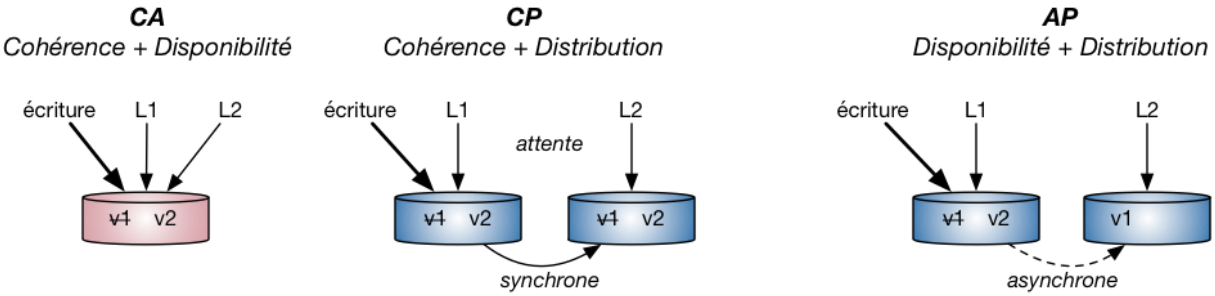


Figure 6 : CA - AP - CP

Prenons le couple **CA** (Consistency-Availability), il représente le fait que lors d'opérations concurrentes sur une même donnée, les requêtes L1 et L2 retournent la nouvelle version (v2) et sans délai d'attente. Cette combinaison n'est possible que dans le cadre de bases de données transactionnelles telles que les SGBDR.

Le couple **CP** (Consistency-Partition Tolerance) propose maintenant de distribuer les données sur plusieurs serveurs en garantissant la tolérance aux pannes (réplication). En même temps, il est nécessaire de vérifier la cohérence des données en garantissant la valeur retournée malgré des mises à jour concurrentielles. La gestion de cette cohérence nécessite un protocole de synchronisation des réplicas, introduisant des délais de latence dans les temps de réponse (L1 et L2 attendent la synchronisation pour voir v2). C'est le cas de la base NoSQL *MongoDB*.

Le couple **AP** (Availability-Partition Tolerance) à contrario s'intéresse à fournir un temps de réponse rapide tout en distribuant les données et les réplicas. De fait, les mises à jour sont asynchrones sur le réseau, et la donnée est "*Eventually Consistent*" (L1 voit la version v2, tandis que L2 voit la version v1). C'est le cas de *Cassandra* dont les temps de réponses sont appréciables, mais le résultat n'est pas garanti à 100% lorsque le nombre de mises à jour simultanées devient important.

Ainsi, la cohérence des données est incompatible avec la disponibilité dans un contexte distribué comme les bases NoSQL.

Le triangle de CAP et les bases de données

Grâce à ce théorème de CAP, il est alors possible de classer toutes les bases de données en les plaçant sur le "triangle de CAP", tout en ajoutant des codes couleurs pour chaque modèle de stockage présenté dans le chapitre précédent.

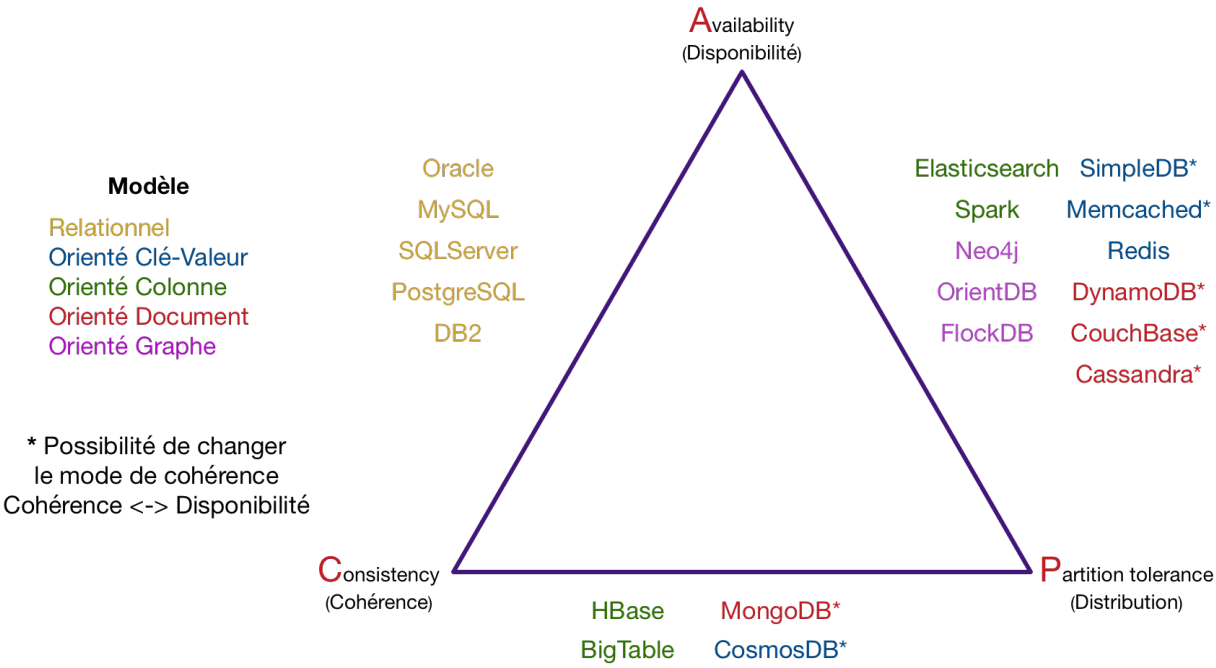


Figure 7 : Triangle de CAP

Nous pouvons constater que les bases de données relationnelles se retrouvent sur la face CA du triangle, combinant disponibilité et cohérence. Nous retrouvons bien MongoDB pour le couple CP (cohérence et distribution) mais également les solutions orientées colonnes comme HBase ou BigTable.

Cassandra est bien une solution orientée document.

Le couple AP (Disponibilité et distribution) regroupe le plus grand nombre de solutions NoSQL. En effet, la plupart cherchent les performances en relâchant volontairement la cohérence. Nous y retrouvons principalement des solutions orientées clé-valeur et graphes, mais également orientées documents (clé-valeur étendu).

Certaines solutions proposent également de modifier la politique de gestion de la concurrence (DynamoDB, CouchBase, Cassandra, MongoDB, CosmosDB...), dans ce cas, ils changent simplement de face sur ce triangle en passant de CP à AP.

L'avantage de ce triangle CAP est de fournir un critère de choix pour votre application. Vous pouvez ainsi vous reposer sur vos besoins en terme de fonctionnalité, de calculs et de cohérence. Le triangle servira de filtre sur les myriades de solutions proposées.

3.Mise en place d'une base de données MongoDB

MongoDB est une base de données NoSQL orientée documents. Comme nous le verrons, l'ensemble du système tourne autour de cette gestion de documents, y compris le langage d'interrogation, ce qui en fait son point fort. Nous allons nous attaquer dès maintenant à la mise en place d'un serveur Mongo et comment intégrer vos données dans cet environnement.

Installation

Pour bien fonctionner, une base *MongoDB* a besoin de trois choses :

- L'installation du serveur.
- La création d'un répertoire pour stocker les données. Par exemple : C:\data\db (par défaut)
- Le lancement du serveur, avec l'exécutable mongod (disponible sur $MONGO/bin). Le serveur attend une connexion sur le port 27017.

Modélisation de documents

Documents JSON

Maintenant que l'environnement est prêt, une question se pose : à quoi ressemblent mes données ? À des documents JSON. Le modèle est très simple :

1. Tout est clé/valeur : **"clé" : "valeur"**
2. Un document est encapsulé dans des accolades **{...}**, pouvant contenir des listes de clés/valeurs
3. Une valeur peut être un type scalaire (entier, nombre, texte, booléen, null), des listes de valeurs **[...]**, ou des documents imbriqués

On peut donc insérer un document dans notre collection "test":

```
db.test.save (

{

  "cours" : "NoSQL",

  "chapitres" : ["familles", "CAP", "sharding", "choix"],

  "auteur" : {

     "nom" : "Travers",

     "prenom" : "Nicolas"

  }

} )
```

Il faut savoir que MongoDB ne gère aucun schéma pour les collections. Donc pas de structure fixe ni de typage, ce qui fait la force des SGBDR. Du coup, il est difficile lors de l'interrogation de connaître le contenu de la base de données. Le logiciel **Studio3T** permet d'extraire des statistiques sur la structure des documents. Ce qui permet d'en avoir une vision globale.

Relationnel vers JSON - comment faire ?

Cela semble simple, c'est à s'y méprendre ! En effet, les données utilisées de manière classiques sont en relationnel (Exportation en CSV pour faire simple). Mais voilà, nous devons faire face à un gros problème dans les bases NoSQL : Il faut proscrire les jointures !

De fait, si vous avez un SGBDR avec deux tables, vous ne pourrez pas faire de requêtes de jointure (ou alors, ça vous coûtera horriblement cher). Il faut dans ce cas trouver une solution pour fusionner les deux tables pour n'en produire qu'une seule en sortie. C'est ce qu'on appelle la dénormalisation.

Prenons notre exemple : une personne peut avoir plusieurs domaine d'expertise, des emplois successifs, et une habitation :

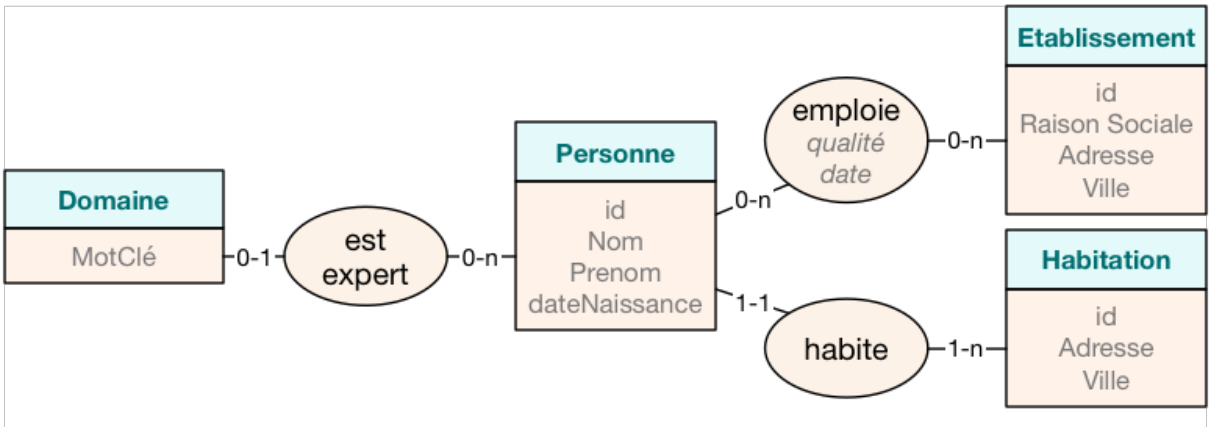


Figure 8 : Schéma Entité/Association original

Si je dois intégrer une base de données avec une collection par entité (rectangle) et association (ovales), le nombre de jointures pour les requêtes sur la base NoSQL risque de faire exploser le système. Du coup, des fusions sont nécessaires pour réduire le coût des requêtes. Mais quelles tables doit-on imbriquer ? Dans quel sens le faire ?

Dénormalisation du schéma

Voici quelques étapes de modélisation qui vont vous permettre de produire des documents JSON qui répondront à votre demande, tout en minimisant les problèmes de jointures et d'incohérences :

- **Des données fréquemment interrogées conjointement.**
  Par exemple, les requêtes demandent fréquemment le lieu d'habitation d'une personne. De fait, la jointure devient coûteuse. Accessoirement, cette information étant peu mise à jour, cela pose peu de problèmes. Résultat, l'entité 'Habitation' et l'association 'habite' sont intégrés à l'entité Personne. Habitation devient un document imbriqué à l'intérieur de Personne, représenté par : "{habite}"
- **Toutes les données d'une entité sont indépendantes.**
  Prenons l'exemple des domaines d'expertise d'une personne, ils sont indépendants des

domaines d'une autre personne. De fait, rapatrier les données de cette entité n'impacte aucune autre instance de Personne. Ainsi, la liste des domaines est importé dans Personne et représenté par : "[domaines]"

- **Une association avec des relations 1-n des deux côtés.**
  Cette fois-ci, c'est plus délicat pour l'entité Etablissement. Une personne peut avoir plusieurs emplois et un employeur, plusieurs employés. De fait, une imbrication de l'employeur dans Personne peut avoir de gros impacts sur les mises à jour (tous les employés à mettre à jour !). Il est donc peu recommandé d'effectuer une fusion complète. Pour cela, seule l'association est imbriquée sous forme d'une liste de documents, intégrant les attributs (qualité et date), ainsi qu'une référence vers l'employeur. Ainsi : "[{emploie+ref}]"
- **Même taux de mises à jour.**
  Dans le cas des emplois d'une personne, là également nous pourrions effectuer une fusion de l'association "emploie". En effet, le taux de mises à jour des emplois est équivalent à celui de la Personne, de fait, sans incidence sur les problèmes de cohérence de données.

Ces étapes de modélisation sont un résumé de plusieurs articles de recherches en Système d'Information. Si vous souhaitez en lire d'avantage : Query-Driven [Chebotko15], MDA-NoSQL [Abdelhedi et al. 17], Document-oriented NoSQL modelling [Mason15].
Au vu de ces critères, voici le schéma qui pourrait être obtenu par fusion successives :

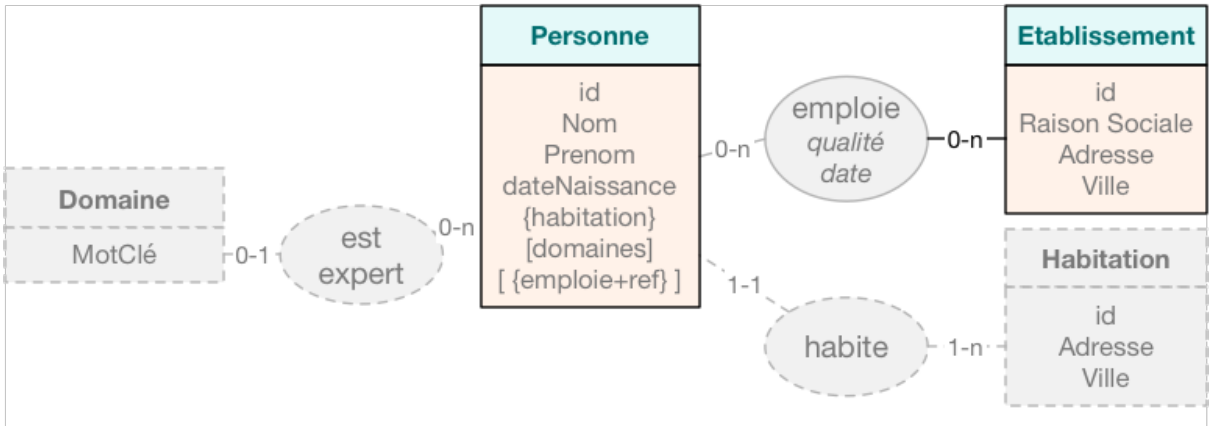


Figure 9 : Schéma dénormalisé

Les listes sont représentées par des crochets et les imbrications par des accolades. Nous pourrons remarquer que les établissements sont stockés dans une association à part, mais que la référence est gardée dans une liste intégrée à l'entité "Personne". Voici un exemple d'instance de cette représentation. Vous pouvez remarquer que les établissements sont référencés et des informations peuvent ne pas être renseignées.

```
{
  "_id" : 1,    "nom" : "Travers",   "prenom" : "Nicolas",
  "domaines" : ["SGBD", "NoSQL", "RI", "XML"],
  "emplois" : [
    {"id_etablissement" : "100", "qualité" : "Maître de Conférences",
```

```
        "date" : "01/09/2007"},
    {"id_etablissement" : "101", "qualité" : "Vacataire",
        "date" : "01/09/2012"}
  ],
  "Habite" : {"adresse" : "292 rue Saint Martin", "ville" : "Paris"}
}
```

Maintenant, nous sommes prêts pour concevoir des bases NoSQL orientées documents avec des fusions. Ne reste plus qu'à faire la conversion de données relationnelles vers ces documents. Pour cela, il est possible de créer un ETL (Extract-Transform-Load) avec votre langage de programmation préféré qui va, à la volée, chercher les informations nécessaires pour produire des données correspondant au schéma. Une seconde possibilité est d'utiliser l'outil OpenRefine (anciennement *GoogleRefine)* pour nettoyer et définir un format de sortie JSON. La dernière possibilité est d'intégrer chaque fichier dans MongoDB (chacun une collection distincte), puis d'utiliser les opérateurs $lookup et $redact pour produire la sortie souhaitée (les opérateurs MongoDB seront étudiés dans le chapitre suivant).

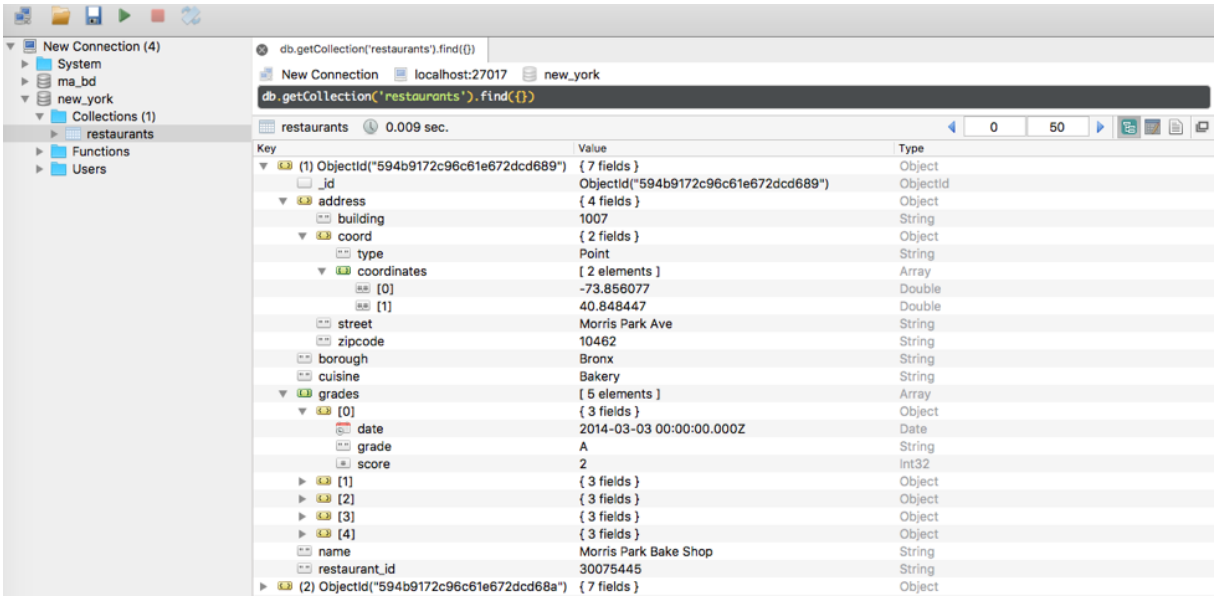
Importer les données

Nous allons maintenant importer un jeu de données déjà formaté à notre base de données. Un jeu de données OpenData est disponible sur des restaurants de New York produits par la mairie sur les résultats des inspections. Pour l'importation :

1. Téléchargez l'archive suivante : restaurants.zip
2. Décompresser l'archive (j'appellerai le répertoire utilisé **$RESTAURANT**)
3. Nous allons créer une base de données "**new_york**" (paramètre *--db*) et une collection "**restaurants**" (paramètre *--collection*). Attention, il ne faut pas de majuscules !
4. Dans une console (Windows : invite de commande, Linux : Shell/Konsole), aller dans le répertoire *$MONGO/bin*
5. Exécutez la commande suivante :

```
$MONGO/bin/mongoimport --db new_york --collection restaurants
$RESTAURANT/restaurants.json
```

Ca y est, vous avez une base de données de 25 357 restaurants que vous pouvez consulter directement avec Robo3T.



Screenshot de la collection "restaurants" sous Robo3T

Nous allons maintenant pouvoir interroger cette collection avec le langage proposé par MongoDB.

Interrogez vos données avec MongoDB

L’interrogation de données avec MongoDB peut être légèrement déroutante au début. En effet, beaucoup ont l’habitude d’un langage "à la SQL" qui permet de spécifier simplement ce que l’on veut, mais pas comment. Toutefois, ce type de langage déclaratif devient inadapté aux nombreuses subtilités de la manipulation de données qui fonctionne difficilement dans un contexte distribué.

Le langage de MongoDB repose sur Javascript, ce qui facilite l’utilisation de JSon, des appels de fonctions, la création de librairies ou la définition de programmes Map/Reduce.

Toute commande sur la collection restaurants utilise le préfixe : "**db.restaurants**".

Il suffira d’y associer la fonction souhaitée pour avoir un résultat.
Avant de commencer, il faut voir à quoi ressemble un document de notre collection *restaurants*. Utilisons la fonction "*findOne()*".

```
db.restaurants.findOne(){

    "_id" : ObjectId("594b9172c96c61e672dcd689"),

    "restaurant_id" : "30075445",

    "name" : "Morris Park Bake Shop",

    "borough" : "Bronx",

    "cuisine" : "Bakery",

    "address" : {

        "building" : "1007",

        "coord" :{"type":"Point","coordinates":[-73.856077,40.848447]},

        "street" : "Morris Park Ave",

        "zipcode" : "10462"

    },

    "grades" : [

        {"date" : ISODate("2014-03-03T00:00:00.000Z"),"grade" : "A","score" : 2},

        {"date" : ISODate("2013-09-11T00:00:00.000Z"),"grade" : "A","score" : 6},
```

```
        {"date" : ISODate("2013-01-24T00:00:00.000Z"),"grade" : "A","score" : 10},
        {"date" : ISODate("2011-11-23T00:00:00.000Z"),"grade" : "A","score" : 9},
        {"date" : ISODate("2011-03-10T00:00:00.000Z"),"grade" : "B","score" : 14}
    ]
}
```

Chaque restaurant a un nom, un quartier (*borough*), le type de cuisine, une adresse (document imbriqué, avec coordonnées GPS, rue et code postale), et un ensemble de notes (résultats des inspections).

Filtrer et Projeter les données avec un motif JSon

Le point fondamental à retenir pour le langage de MongoDB est le "clé/valeur", avec des définitions de motifs en documents **JSON**. Ainsi, chaque requête sera un document, avec des clés et des valeurs. Le deuxième point est la structure d'un document. Chaque document est traité indépendamment, comme en relationnel, mais les documents contiennent maintenant des imbrications et des listes, ce qui n'est pas habituel. Nous allons en illustrer les conséquences à partir d'exemples.

Filtrage

Commençons par un type simple de requête : le filtrage. Nous allons pour cela utiliser la fonction "find" à appliquer directement à la collection. L'exemple ci-dessous récupère tous les restaurants dans le quartier (borough) de Brooklyn.

```
db.restaurants.find( { "borough" : "Brooklyn" } )
```

6 085 restaurants sont retournés.

Pour compter, il suffit d'ajouter la fonction "count"

```
db.restaurants.find( { "borough" : "Brooklyn" } ).count()
```

En effet, nous sommes en *Javascript* : le résultat d'un *find* est une liste dont on peut compter les documents avec la méthode de liste "count" Maintenant, nous cherchons parmi ces restaurants ceux qui font de la cuisine italienne. Pour combiner deux "clés/valeurs", il suffit de faire le document motif donnant quels paires clés/valeurs sont recherchés :

```
db.restaurants.find(
    { "borough" : "Brooklyn",
      "cuisine" : "Italian" }
```

```
)
```

Ensuite, on va chercher les restaurants présents sur la 5° Avenue. La clé "street" est imbriquée dans l'adresse ; comment filtrer une clé imbriquée (obtenue après fusion) ? Il faut utiliser les deux clés, en utilisant un point "." comme si c'était un objet.

```
db.restaurants.find(

    { "borough" : "Brooklyn",

      "cuisine" : "Italian",

      "address.street" : "5 Avenue" }

)
```

Pourquoi ne pas chercher le mot "pizza" dans le nom du restaurant ? Pour cela, il faut utiliser les expressions régulières avec "/xxx/i" (le i pour "Insensible à la casse" - majuscule/minuscule).

```
db.restaurants.find(

    { "borough" : "Brooklyn",

      "cuisine" : "Italian",

      "address.street" : "5 Avenue",

      "name" : /pizza/i }

)
```

Il ne reste maintenant que 2 restaurants Italiens dans le quartier de Brooklyn dans la 5° Avenue, dont le nom contient le mot "pizza".

Projection

Et si nous ne gardions dans le résultat que le nom ? C'est ce que l'on appelle une projection. Un deuxième paramètre (optionnel) de la fonction *find* permet de choisir les clés à retourner dans le résultat. Pour cela, il faut mettre la clé, et la valeur "1" pour projeter la valeur (on reste dans le concept "clé/valeur").

```
db.getCollection('restaurants').find(

    {"borough":"Brooklyn",

     "cuisine":"Italian",

     "name":/pizza/i,

     "address.street" : "5 Avenue"},

    {"name":1}

)



{"_id" : ObjectId("594b9173c96c61e672dd074b"), "name" : "Joe'S Pizza"}

{"_id" : ObjectId("594b9173c96c61e672dd1c16"), "name" : "Gina'S Pizzaeria/ Deli"}
```

L'identifiant du document "_id" est automatiquement projeté, si vous souhaitez le faire disparaître il suffit de rajouter la valeur "0" : {"name":1, "_id":0}

Et si nous regardions les résultats des inspections des commissions d'hygiène ? Il faut pour cela regarder la valeur de la clé "*grades.score*".

```
db.getCollection('restaurants').find(

    {"borough":"Brooklyn",

     "cuisine":"Italian",

     "name":/pizza/i,

     "address.street" : "5 Avenue"},

    {"name" : 1,

     "grades.score" : 1}

)



{
```

```
    "name" : "Joe'S Pizza",
    "grades" : [
        {"score" : 12},
        {"score" : 13},
        {"score" : 42},
        {"score" : 25},
        {"score" : 19},
        {"score" : 40},
        {"score" : 22}
    ]
}
{
    "name" : "Gina'S Pizzaeria/ Deli",
    "grades" : [
        {"score" : 12},
        {"score" : 23},
        {"score" : 13}
    ]
}
```

Ce qui est intéressant est de voir que le score est retrouvé dans les documents imbriqués de la liste "grades", et que tous les scores sont projetés.

À noter qu'un score bas est un bon score pour une inspection aux États-Unis

Filtrage avec des opérations

Le filtrage par valeur exacte n'est pas suffisant pour exprimer tout ce que l'on souhaiterait trouver dans la collection. Je cherche maintenant les noms et scores des restaurants de Manhattan ayant un score inférieur à 10. On peut utiliser des opérateurs arithmétiques sur les clés (valeur numérique). Pour cela, il sera préfixé d'un dollar **$**. "Plus petit que" en anglais se dit "*less than*", l'opérateur est donc **$lt**. Il faut y associer la valeur souhaitée : **"grades.score" : {"$lt" : 10}**. Vous trouverez les détails ici.

L'opérateur est **toujours** imbriqué dans un document. Le concept "clé/valeur" doit être respecté. Voici les opérateurs disponibles avec des exemples associés :

$gt, $gte	>, ≥	Plus grand que (*greater than*)	"a" : {"$gt" : 10}
$lt, $lte	<, ≤	Plus petit que (*less than*)	"a" : {"$lt" : 10}
$ne	≠	Différent de (*not equal*)	"a" : {"$ne" : 10}
$in, $nin	∈, ∉	Fait parti de (ou ne doit pas)	"a" : {"$in" : [10, 12, 15, 18] }
$or	∨	OU logique	"a" : {“$or” : [{"$gt" : 10}, {“$lt” : 5}] }
$and	∧	ET logique	"a" : {“$and” : [{"$lt" : 10}, {“$gt” : 5}] }
$not	¬	Négation	“a" : {“$not” : {"$lt" : 10} }
$exists	∃	La clé existe dans le document	“a” : {“$exists” : 1}
$size		test sur la taille d'une liste (uniquement par égalité)	“a” : {“$size” : 5}

La requête donne ceci :

```
db.getCollection('restaurants').find(

    {"borough":"Manhattan",

     "grades.score":{$lt : 10}

    },

    {"name":1,"grades.score":1, "_id":0})



{

    "name" : "Dj Reynolds Pub And Restaurant"

    "grades" : [

        {"score" : 2},

        {"score" : 11},

        {"score" : 12},

        {"score" : 12}

    ]

}
```

Ce qui est déroutant dans ce résultat, c'est le fait que l'on trouve des scores supérieurs à 10 !

Nous ne sommes plus en relationnel, ainsi, l'opération **"grades.score" : {"$lt" : 10}**, veut dire :

Est-ce que la liste "grades" contient un score (**au moins**) avec une valeur inférieure à 10 ?

Et en effet, il y a un score à "2" respectant la question.
Si l'on souhaite ne récupérer que ceux qui n'ont pas de score supérieur à 10, il faut alors combiner la première opération avec une négation de la condition " ≥ 10 ". La condition est alors vérifiée sur chaque élément de la liste.

```
db.getCollection('restaurants').find(

    {"borough":"Manhattan",

     "grades.score":{

         $lt:10,

         $not:{$gte:10}

     }

    },

    {"name":1,"grades.score":1, "_id":0})



{

    "name" : "1 East 66Th Street Kitchen",

    "grades" : [

        {"score" : 3},

        {"score" : 4},

        {"score" : 6},

        {"score" : 0}

    ]

}
```

On pourrait même corser la chose en cherchant les restaurants qui ont un grade ‘C’ avec un score inférieur à 40.

```
db.restaurants.find({

    "grades.grade" : "C",

    "grades.score" : {$lt : 30}

   },

   {"Grades.grade":1, "grades.score":1}

);



{

    "_id" : ObjectId("594b9172c96c61e672dcd695"),

    "grades" : [

       {"grade" : "B","score" : 21},

       {"grade" : "A","score" : 7},

       {"grade" : "C","score" : 56},

       {"grade" : "B","score" : 27},

       {"grade" : "B","score" : 27}

    ]

}

{

    "_id" : ObjectId("594b9172c96c61e672dcd6bc"),

    "grades" : [

       {"grade" : "A","score" : 9},

       {"grade" : "A","score" : 10},

       {"grade" : "A","score" : 9},
```

```
        {"grade" : "C","score" : 32}

    ]

}
```

Le résultat est pour le moins étrange concernant le premier document car il y a bien un grade C, mais avec un score de 56 ! Si l'on regarde la requête de plus près, ce n'est pas étonnant. Y a-t-il un grade 'C' ? oui. Y a-t-il un score inférieur à 40 ? oui… Nous avons oublié de préciser qu'il fallait que ce soit vérifié sur chaque élément ! La vérification se fait sur l'intérieur de la liste, pas sur chaque élément de la liste. Pour cela, un opérateur permet de faire une vérification instance par instance : **$elemMatch**.

```
db.restaurants.find({

    "grades" : {

            $elemMatch : {

                "grade" : "C",

                "score" : {$lt :40}

            }

    }

  },

  {"grades.grade" : 1,"grades.score" : 1}

);



{

    "_id" : ObjectId("594b9172c96c61e672dcd6bc"),

    "grades" : [

            {"grade" : "A","score" : 9},

            {"grade" : "A","score" : 10},

            {"grade" : "A","score" : 9},
```

```
            {"grade" : "C","score" : 32}

    ]

}
```

Cette fois-ci le résultat est bon.

Une dernière pour finir, je voudrais maintenant les noms et quartiers des restaurants dont la dernière inspection (la plus récente, donc la première de la liste) a donné un grade 'C'. Il faut donc chercher dans le premier élément de la liste. Pour cela il est possible de rajouter l'indice recherché (indice 0) dans la clé.

```
db.restaurants.find({

    "grades.0.grade":"C"

  },

  {"name":1, "borough":1, "_id":0}

);



{"borough" : "Queens","name" : "Mcdonald'S"}

{"borough" : "Queens","name" : "Nueva Villa China Restaurant"}

{"borough" : "Queens","name" : "Tequilla Sunrise"}

{"borough" : "Manhattan","name" : "Dinastia China"}
```

Distinct

Au fait, quels sont les différents quartiers de New York ? Pour cela, on peut utiliser la fonction "distinct()" avec la clé recherchée pour avoir une liste de valeur :

```
db.restaurants.distinct("borough")



[

    "Bronx",

    "Brooklyn",
```

```
    "Manhattan",
    "Queens",
    "Staten Island",
    "Missing"
]
```

Et si nous le faisions sur une liste de valeurs comme les grades donnés par les inspecteurs ?

```
db.restaurants.distinct("grades.grade");



[
    "A",
    "B",
    "Z",
    "C",
    "P",
    "Not Yet Graded"
]
```

La fonction distincte va donc chercher les instances de la liste pour en extraire chaque grade et produire une liste distincte (plutôt que de faire une liste de liste distincte).

Créer une séquence d'opérations avec "*aggregate*"

Et si maintenant, nous regardions des opérations plus complexes pour bien manipuler nos données ? Pour cela, la fonction "**aggregate ()**" permet de spécifier des chaînes d'opérations, appelées pipeline d'agrégation.

Cette fonction aggregate prend une liste d'opérateurs en paramètre. Il existe plusieurs types d'opérateurs de manipulation de données. Nous allons nous concentrer par la suite sur les principaux :

- **{$match : {} }** : C'est le plus simple, il correspond au premier paramètre de la requête find que nous avons fait jusqu'ici. Il permet donc de filtrer le contenu d'une collection.
- **{$project : {} }** : C'est le second paramètre du find. Il donne le format de sortie des documents (projection). Il peut par ailleurs être utilisé pour changer le format d'origine.
- **{$sort : {} }** : Trier le résultat final sur les valeurs d'une clé choisi.
- **{$group : {} }** : C'est l'opération d'agrégation. Il va permettre de grouper les documents par valeur, et appliquer des fonctions d'agrégat. La sortie est une nouvelle collection avec les résultats de l'agrégation.
- **{$unwind : {} }** : Cet opérateur prend une liste de valeur et produit pour chaque élément de la liste un nouveau document en sortie avec cet élément. Il pourrait correspondre à une jointure, à ceci près que celle-ci ne filtre pas les données d'origine, juste un complément qui est imbriqué dans le document. On pourrait le comparer à une jointure externe avec imbrication et listes.

Vous aurez remarqué que chaque opérateur est imbriqué dans un document, et la valeur est elle-même un autre document. C'est la structure imposée pour la définition de l'opérateur. Pour illustrer un aggregate, je vais reprendre notre dernière requête *find* :

```
db.restaurants.aggregate( [

    { $match : {

        "grades.0.grade":"C"

    }},

    { $project : {

        "name":1, "borough":1, "_id":0

    }}

] )
```

Le résultat est identique, toutefois, c'est un peu plus lourd à écrire. On identifie bien chaque opérateur, avec leur définition.

Rappelez-vous que l'interpréteur mongodb utilise le langage Javascript. Chaque opérateur peut alors être défini par une variable que nous pouvons utiliser directement :

```
varMatch = { $match : { "grades.0.grade":"C"} };

varProject = { $project : {"name":1, "borough":1, "_id":0}};

db.restaurants.aggregate( [ varMatch, varProject ] );
```

Nous utiliserons ces variables par la suite pour alléger le code en ré-utilisant des variables.

Tri

Trions maintenant le résultat par nom de restaurant par ordre croissant (croissant : 1, décroissant : -1) :

```
varSort = { $sort : {"name":1} };

db.restaurants.aggregate( [ varMatch, varProject, varSort ] );
```

Groupement simple

Comptons maintenant le nombre de ces restaurants (premier rang ayant pour valeur C). Pour cela, il faut définir un opérateur $group. Celui-ci doit contenir obligatoirement une clé de groupement (_id), puis une clé (total) à laquelle on associe la fonction d'agrégation ($sum) :

```
varGroup = { $group : {"_id" : null, "total" : {$sum : 1} } };

db.restaurants.aggregate( [ varMatch, varGroup ] );



{"_id" : null, "total" : 220}
```

Cette requête est équivalente à :

```
db.restaurants.count({"grades.0.grade":"C"})

db.restaurants.find({"grades.0.grade":"C"}).count()
```

Mais si vous souhaitez faire un pipeline d'agrégation, vous serez obligé d'utilisé la fonction "aggregate".

Ici, pas de valeur de groupement demandé (on compte simplement). Nous avons choisi de mettre la valeur *null*, on aurait pu mettre "toto", cela fonctionne également. La clé "total" est créée dans le résultat et va effectuer la "somme des 1" pour chaque document source. Faire une somme de 1 est équivalent à compter le nombre d'éléments.

Groupement par valeur

Comptons maintenant par quartier le nombre de ces restaurants. Il faut dans ce cas changer la clé de groupement en y mettant la valeur de la clé "borough". Mais si l'on essaye ceci :

```
varGroup2 = { $group : {"_id" : "borough", "total" : {$sum : 1} } };

db.restaurants.aggregate( [ varMatch, varGroup2 ] );



{"_id" : "borough", "total" : 220}
```

La valeur de la clé de groupement prend une unique valeur "*borough*". Le problème vient du fait que nous n'avons pas pris la valeur, mais seulement précisé un texte ! Pour prendre la valeur, il faut préfixer la clé par un dollar "$borough", cela va, lors de l'exécution, indiquer qu'il faut utiliser la valeur de la clé pour l'agrégation.

```
varGroup3 = { $group : {"_id" : "$borough", "total" : {$sum : 1} } };

db.restaurants.aggregate( [ varMatch, varGroup3 ] );



{"_id" : "Bronx",         "total" : 27.0}

{"_id" : "Staten Island",     "total" : 7.0}

{"_id" : "Manhattan",         "total" : 83.0}

{"_id" : "Brooklyn",         "total" : 56.0}

{"_id" : "Queens",         "total" : 47.0}
```

Ce qui fait bien un total de 220 !

Unwind

Maintenant, trouvons le score moyen des restaurants par quartiers, et trions par score décroissant. Pour cela, nous groupons les valeurs par quartier (*$borough*), nous utilisons la fonction d'agrégat "*$avg*" pour la moyenne (*average*), puis nous trions sur la nouvelle clé produite "$moyenne".

Un problème se pose comment calculer la moyenne sur une "*grades.score*" qui est une liste ? Il faudrait au préalable retirer chaque élément de la liste, puis faire la moyenne sur l'ensemble des valeurs. C'est l'opérateur **$unwind** qui s'en charge, il suffit de lui donner la clé contenant la liste à désimbriquer.

```
varUnwind = {$unwind : "$grades"}

varGroup4 = { $group : {"_id" : "$borough", "moyenne" : {$avg : "$grades.score"} }
};

varSort2 = { $sort : { "moyenne" : -1 } }

db.restaurants.aggregate( [ varUnwind, varGroup4, varSort2 ] );



{ "_id" : "Queens", "moyenne" : 11.634865110930088 }

{ "_id" : "Brooklyn", "moyenne" : 11.447723132969035 }

{ "_id" : "Manhattan", "moyenne" : 11.41823125728344 }

{ "_id" : "Staten Island", "moyenne" : 11.370957711442786 }

{ "_id" : "Bronx", "moyenne" : 11.036186099942562 }

{ "_id" : "Missing", "moyenne" : 9.632911392405063 }
```

Ce qu'il est important de noter dans cette fonction *aggregate*, c'est que les opérateurs sont composés en séquences : chaque opérateur prend la collection produite par l'opérateur précédent (et non la collection de départ). Ceci permet de créer de longues chaînes d'opérateurs pour faire des calculs lourds.

Attention ! L'ordre des opérations est très important (projection après les filtrages, tri sur le résultat d'un groupe, filtrage avant ou après *unwind*/*group*...).
Pour des calculs plus compliqués ne pouvant être réalisés avec un *aggregate*(structure conditionnelle, multi-output/fusion, calculs complexes...), MongoDB offre la possibilité de programmer en Map/Reduce (avec javascript). Comme cela sort du cadre de cours sur le langage d'interrogation, vous pourrez trouver la documentation ici et un TP de mise en pratique ici.

Sachez par ailleurs que tous les opérateurs sont programmés en *Map/Reduce*pour faire l'évaluation. Le langage de MongoDB est un langage de plus haut-niveau permettant de faire l'interrogation.

Mettre à jour les données

Une base de données ne serait pas utile si nous ne pouvions faire des mises à jour sur les données. Nous avons déjà vu l'insertion (*save*) dans le chapitre précédent, il faut maintenant regarder les mises à jour (*update*) et les suppressions (*delete*). À chaque modification, il faut préciser le document

ciblé (*_id*) et la modification (pour les mises à jour des documents existants).

Update

Pour commencer, nous allons ajouter un commentaire sur un restaurant (opération **$set**) :

```
db.restaurants.update (

   {"_id" : ObjectId("594b9172c96c61e672dcd689")},

   {$set : {"comment" : "My new comment"}}

);



WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

On peut voir qu'un document a été trouvé (*nMatched*) et modifié (*nModified*). Vous pouvez vérifier s'il a été modifié avec une requête *find*.

Pour supprimer une clé, il suffit de remplacer par l'opérateur **$unset**.

```
db.restaurants.update (

   {"_id" : ObjectId("594b9172c96c61e672dcd689")},

   {$unset : {"comment" : 1}}

);
```

Faisons une requête avec filtrage pour choisir les restaurants à commenter (faire les documents un par un est fastidieux). Nous allons attribuer un commentaire en fonction des *grades* obtenus. S'il n'a pas eu de note 'C', nous rajouterons le commentaire 'acceptable'.

```
db.restaurants.update (

   {"grades.grade" : {$not : {$eq : "C"}}},

   {$set : {"comment" : "acceptable"}}

);



WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

Le résultat est identique. Il n'y aurait qu'un seul restaurant sans note 'C' ? Faites la requête find correspondante, vous verrez qu'il y en a 22 649 ! MongoDB interdit par défaut les modifications multiples avec un filtre. Cette opération pourrait être lourde et ralentir le système. Du coup, il faut explicitement dire que les modifications peuvent être multiples :

```
db.restaurants.update (

   {"grades.grade" : {$not : {$eq : "C"}}},

   {$set : {"comment" : "acceptable"}},

   {"multi" : true}

);
```

Update et javascript

Et si l'on cherchait à donner une note en fonction de l'ensemble des inspections effectuées par restaurant ? Cela devient compliqué avec l'opération **$set** (prend au mieux la valeur d'une clé avec un "**$**"). On va pour cela programmer un peu en *javascript* avec des fonctions itératives, et sauvegarder le résultat.

Attention, cela peut être long car il faut exécuter la fonction pour chaque élément de la base, le modifier puis le sauvegarder.
Nous allons ajouter 3 points pour un rang A, 1 point pour B, et -1 pour C.

```
db.restaurants.find( {"grades.grade" : {$not : {$eq : "C"}}} ).forEach(

    function(restaurant){

        total = 0;

        for(i=0 ; i<restaurant.grades.length ; i++){

            if(restaurant.grades[i].grade == "A")         total += 3;

            else if(restaurant.grades[i].grade == "B")    total += 1;

            else                                          total -= 1;

        }

        restaurant.note = total;

        db.restaurants.save(restaurant);
```

```
    }

);
```

Le résultat ci-dessous montre que la meilleure note est obtenue par le "*Taco Veloz*" avec 24 points.

```
db.restaurants.find({}, {"name":1,"_id":0,"note":1,"borough":1}).sort({"note":-
1});
```

```
{ "borough" : "Queens", "name" : "Taco Veloz", "note" : 24 }

{ "borough" : "Manhattan", "name" : "Gemini Diner", "note" : 22 }

{ "borough" : "Manhattan", "name" : "Au Za'Atar", "note" : 22 }

{ "borough" : "Brooklyn", "name" : "Lucky 11 Bakery", "note" : 22 }

{ "borough" : "Queens", "name" : "Mcdonald'S", "note" : 21 }

{ "borough" : "Queens", "name" : "Rincon Salvadoreno Restaurant", "note" : 21 }

{ "borough" : "Manhattan", "name" : "Kelley & Ping", "note" : 21 }
```

Remove

Nous allons maintenant supprimer tous les restaurants qui ont une note de 0.

```
db.restaurants.remove(

   {"note":0},

   {"multi" : true}

);
```

Save

Pour finir, une insertion (en dehors des possibilités d'importation avec mongoimport) s'effectue avec la fonction "*save*" :

```
db.restaurants.save({"_id" : 1, "test" : 1});
```

Dans le cas où un document de la collection aurait le même identifiant "_id", le serait automatiquement écrasé par la nouvelle valeur.