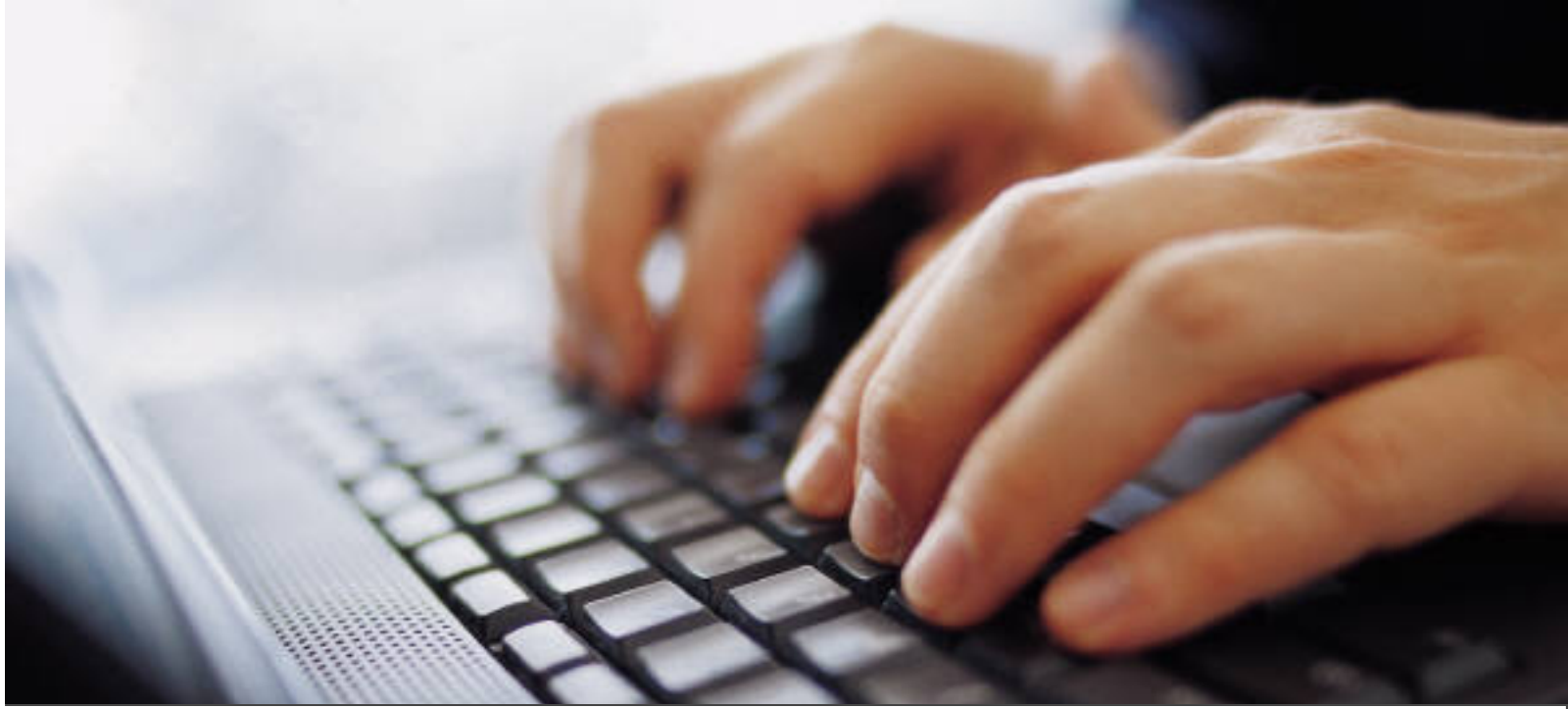

# Hadoop / Big Data

Benjamin Renaut <renaut.benjamin@tokidev.fr>

MBDS

2019 - 2020



8

# HDFS et bases de données relationnelles

# Présentation



- **On a présenté Sqoop, un outil polyvalent qui permet d'importer ou d'exporter des données entre HDFS et les bases de données « classiques » (Oracle, MySQL, PostgreSQL, etc...).**

- **Sqoop est extrêmement utile pour faire des échanges entre les bases de données et HDFS. Il s'agit cependant d'un outil utilisé pour des échanges de données ponctuels: on s'en servira par exemple pour effectuer des traitements Hadoop sur les données de la base toutes les nuits, ou encore toutes les N minutes, etc... en ré-important éventuellement les données dans la base après traitement.**

- **Il y a cependant des cas d'utilisation où on souhaite pouvoir échanger des données entre une base de données et la couche HDFS en *temps réel*. Par exemple, on pourrait avoir des retours statistiques sur un large réseau social depuis un *backend* centralisé, dont les données nécessitent des traitements massifs par le biais de Hadoop.**

# Les connecteurs



- **On pourrait imaginer utiliser un outil comme Sqoop de manière continue, en lançant en permanence la même tâche Sqoop. Cependant, pendant que la tâche s'exécute, de nouvelles données peuvent apparaître; par ailleurs, la charge appliquée au SGBD serait conséquente et provoquerait de potentiels problèmes d'accès aux bases (LOCK des tables).**

- **Pour répondre à ce problème, la plupart des bases de données largement utilisées ont développé des points d'intégration entre les SDBG et HDFS. On désigne souvent ces points d'intégration en parlant de *connecteurs Hadoop*. On voit également les termes *applier* ou encore *bridges*.**

- **Ces connecteurs ont tous en commun la capacité d'écrire sur HDFS les données de la base en permanence, au fur et à mesure. Parfois, ils sont aussi capables de répliquer des modifications faites sur HDFS dans la base de données.**

# MySQL applier for Hadoop



- **Dans le cas de MySQL, le connecteur est le *MySQL applier for Hadoop* d'Oracle.**

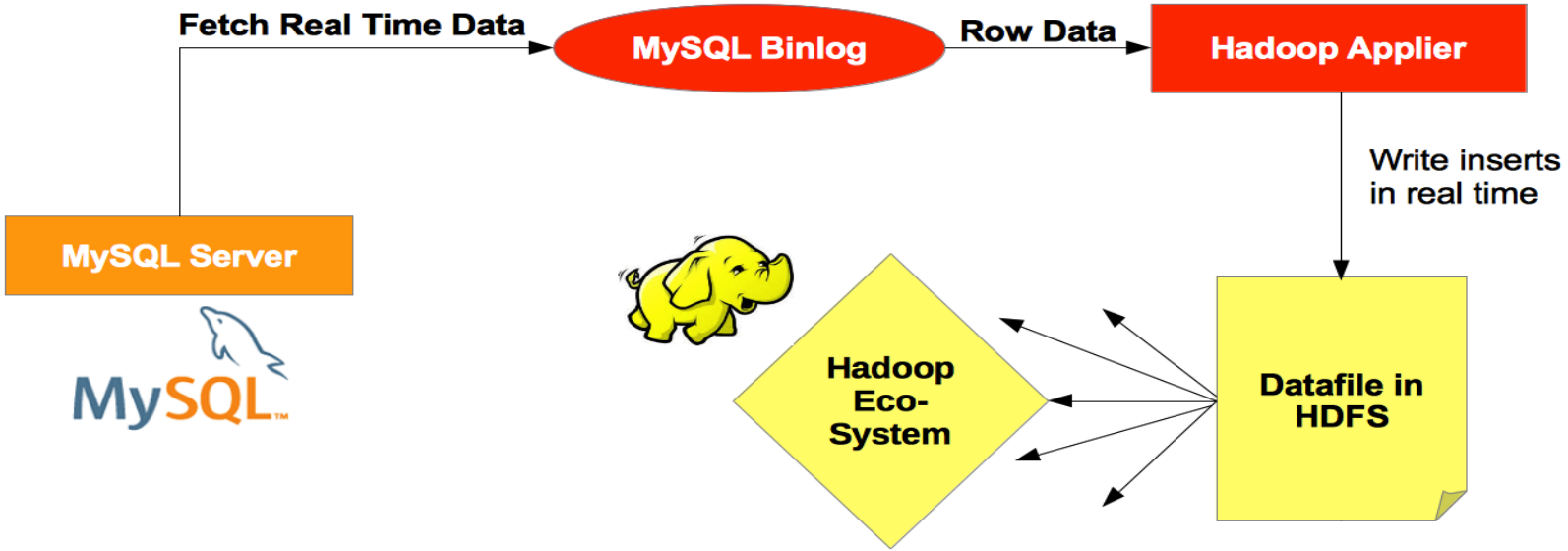
- **Il fonctionne en tirant partie des *binary logs* de MySQL – les logs binaires que MySQL peut générer (si activé) en permanence et qui indiquent toutes les modifications faites sur la base de données. Ce sont ces mêmes *binary logs* qui sont utilisés pour les mécanismes de réplication MySQL classiques (entre nœuds MySQL).**

- **Le connecteur génère un répertoire par base de données sur HDFS. A l'intérieur de ce répertoire, il créé un répertoire différent par table présente dans la base de données. Enfin, à l'intérieur de ces répertoires, il génère des séries de fichiers contenant les données de la table concernée.**

- **Le format des fichiers est configurable: format texte ou format binaire *sequence files* (les mêmes que ceux vus avec Sqoop).**

# MySQL applier for Hadoop

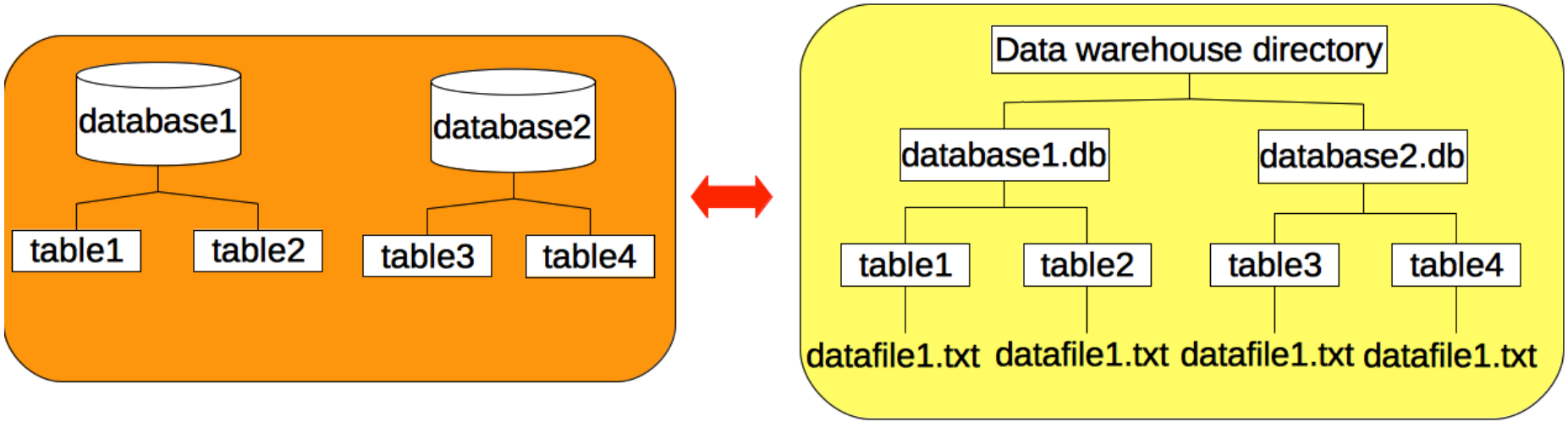


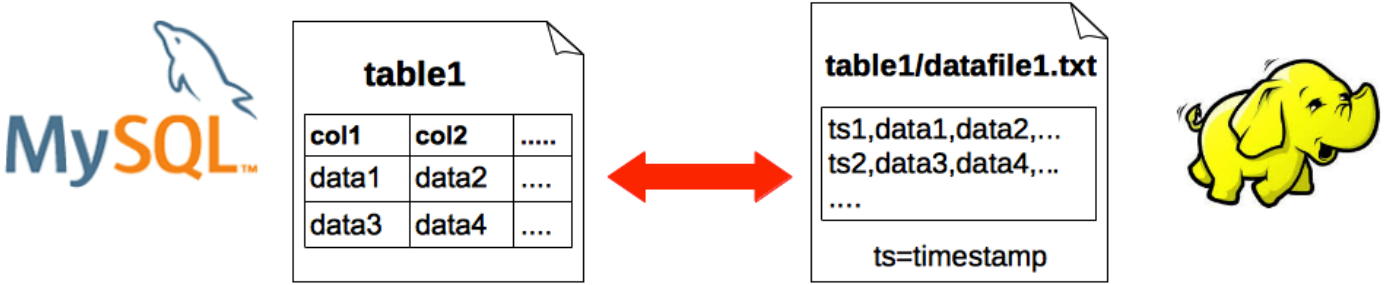



*(source: documentation MySQL)*

# MySQL applier for Hadoop






*(source: documentation MySQL)*

# Oracle SQL connector for HDFS



- **Dans le cas de Oracle, le connecteur est le *Oracle SQL connector for HDFS*.**

- **Il est sensiblement plus puissant que le connecteur de MySQL; notamment, il est capable de lire des données depuis HDFS qui ont été générées par des programmes Hadoop. Il fonctionne en utilisant le mécanisme des *tables externes* Oracle: des « tables » virtuelles Oracles qui désignent en réalité des données situées en dehors du SGBD.**

- **Pour connecter des dossiers ou fichiers HDFS à Oracle, il suffit de créer ces tables externes en définissant un schéma de données bien précis (comme pour une table classique). Ce schéma correspond à un format de données situé sur HDFS.**
  **Il est même possible d'effectuer des jointures entre des tables Oracle « classiques » et des tables virtuelles qui correspondent à des données sur HDFS.**

# Oracle SQL connector for HDFS



- **Le connecteur Oracle est capable de stocker les données sous HDFS dans deux formats principaux: un format texte, similaire au format texte utilisé dans Sqoop et le connecteur MySQL et humainement lisible, ou un format binaire.**
  **Le format binaire en question n'est pas celui des *sequences files* vu précédemment, mais un format propre à Oracle et désigné sous le terme de *data dumps*.**

- **Oracle fourni par ailleurs des APIs Java nécessaires pour accéder simplement et rapidement aux données stockées dans ce format – de la même manière que pour des *Sequence Files*.**
  **Ces APIs définissent notamment des classes *Input/OutputFormat* Hadoop, de la même manière que – par exemple – la classe Hadoop *SequenceFileInputFormat* vue en TP.**
  **On peut donc se servir de ces classes comme d'un format d'entrée pour une tâche Hadoop, ou d'un format de sortie, toujours sur HDFS.**

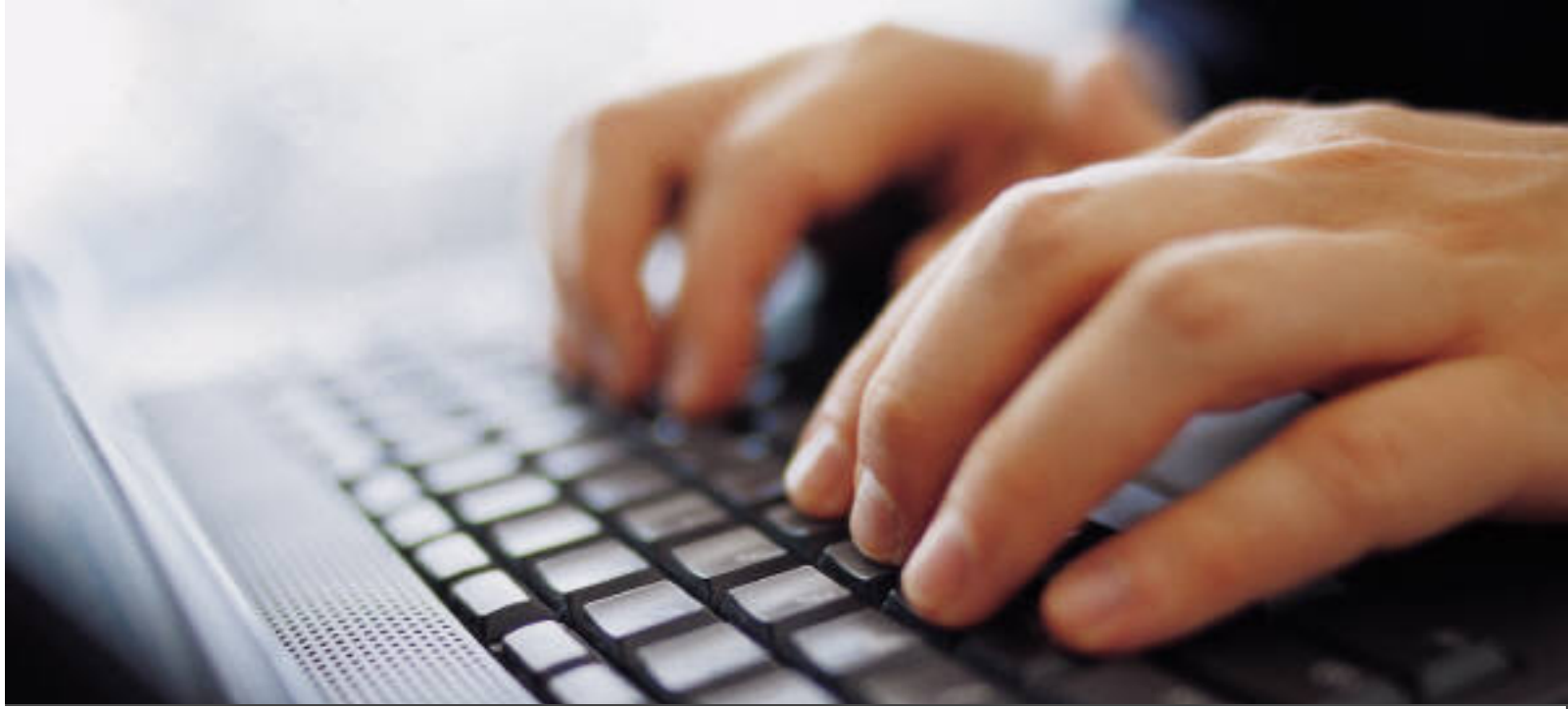
# Autres connecteurs et modèles



- **Il existe de nombreux autres connecteurs permettant l'accés à des données situées dans une base de données depuis Hadoop – certains propriétaires et commerciaux, d'autres Open Source et librement disponibles.**

- **D'autres connecteurs évitent complètement HDFS – Hadoop va alors récupérer ses données dans le SGBD directement lors de l'exécution d'une tâche. Il suffit au programmeur d'utiliser une autre API pour préciser à Hadoop la source de ses données.**

- **Exemple: VoltDB. Il s'agit d'un SGBD Open Source innovant qui utilise un stocke ses données en mémoire vive (un peu comme le moteur MEMORY de MySQL), de manière distribuée et répliquée, et constitue ainsi une alternative à HDFS. Il propose des APIs Java pour l'accés aux données depuis Hadoop, et offre ainsi l'avantage de pouvoir directement s'intégrer à Hadoop et venir « remplacer » HDFS.**



10

# Introduction à MongoDB

# Introduction - Les SGBD NoSQL



- **MongoDB fait partie d'une catégorie de systèmes de gestion de bases de données qu'on désigne sous le terme NoSQL.**

- **Ces systèmes, contrairement aux SGBD relationnels classiques, ne sont pas (pour la plupart) adressables par le biais de SQL.**

- **Ils ne structurent pas non plus les données sous une forme relationnelle « forte », mais utilisent des modèles alternatifs.**

- **Au delà de ces aspects, ils ont généralement pour caractéristiques d'être extrêmement scalables, et de pouvoir stocker de très larges quantités de données. Ils sont donc tout particulièrement adaptés aux problématiques relatives au big data.**

# Introduction - Les SGBD NoSQL



- **On les classe généralement dans trois catégories distinctes:**
  - **Les SGBD NoSQL clef;valeur: ils ont pour vocation de stocker de simples couples (clef;valeur); et sont donc pour des raisons évidentes tout particulièrement adaptés au map/reduce. Exemples: redis, riak.**
  - **Les SGBD NoSQL *document-based*: ils stockent les données sous la forme de *documents* au format variable mais structuré (généralement JSON ou XML); c'est à cette catégorie qu'appartient MongoDB.**
  - **Les SGBD NoSQL *graph-based*: ils ont pour vocation de stocker les données sous la forme de graphes; et sont donc utilisés sont les use cases correspondants. Par exemple: Neo4J, OpenCog.**

# Introduction - Les SGBD NewSQL



- **Les problématiques Big Data ont aussi vu l'avènement de SGBD qualifiés de « NewSQL »: des SGBD partiellement ou entièrement relationnels et accessibles par le biais de SQL, mais qui sont caractérisés par une forte scalabilité possible.**

- **On en distingue généralement deux catégories:**

  - **Les SGBD NewSQL « inédits », développés pour répondre aux problématiques Big Data; par exemple VoltDB ou encore Google Spanner.**

  - **Les adaptations aux SGBD relationnels existants, par exemple le moteur NDBCluster de MySQL.**

# MongoDB - Présentation



- **MongoDB est un SGBD NoSQL *document-based*.**
- **Il s'agit d'un logiciel libre; initialement propriétaire (2007), il a été plus tard diffusé en licence AGPLv3 (2009).**
- **Il est développé et maintenu essentiellement par une entreprise prisée, MongoDB Inc.; mais également par de nombreux bénévoles et entités tierces qui l'utilisent.**
- **Site officiel:**

  **http://www.mongodb.org/**

# MongoDB - Présentation



- **MongoDB est utilisé aujourd'hui par de nombreux gros acteurs, comme par exemple:**
  - **Ebay**
  - **Craigslist**
  - **Adobe**
  - **LinkedIn**
- **Ses performances sont excellentes; et sont estimées dans la plupart des *benchmarks* comme supérieures à la plupart des autres SGBD *document-based* équivalents (comme Cassandra).**

# Fonctionnement



- **MongoDB se présente sous la forme d'un serveur; un *daemon*/service système.**

- **Par défaut, pour y accéder, on se connectera en TCP/IP sur le port TCP 27017. Comme la plupart des SGBD, il supporte la gestion d'utilisateurs/mots de passe pour s'identifier.**

- **Des APIs sont disponibles pour de nombreux langages: C/C++, Java, Python, PHP, PigLatin…**

- **Un connecteur Hadoop est par ailleurs disponible; permettant de lire les données d'entrée d'un programme Hadoop directement depuis MongoDB, et d'écrire les résultats d'un tel programme directement dans MongoDB également.**

# Fonctionnement – BSON



- **MongoDB stocke les données non pas sous la forme de tables à la structure statique, mais sous la forme de documents rédigés dans un langage proche de JSON: BSON (Binary jSON).**

- **En interne, les données sont stockées sous un format binaire; mais elles sont évidemment la plupart du temps lues, écrites et manipulées sous un format textuel (c'est le SGBD qui compile le format texte).**

- **BSON est en réalité un *subset* de JSON qui inclut quelques fonctionnalités additionnelles (notamment des types et fonctions non disponibles en JSON).**

# Fonctionnement – BSON



- Un exemple de document MongoDB:

```
{
    prenom: 'John',
    nom: 'Smith',
    datenaissance: ISODate("1984-12-11T00:00:00Z"),
    "notes": [
          {
             "matiere" : 'Big Data',
             "note" : 12.0
          },
          {
             "matiere" : 'Gestion de projets',
             "note" : 14.0
          }
      ]
}
```

# Fonctionnement – BSON



- **Les documents sont par ailleurs insérés au sein de *collections*. Il s'agit de groupes de documents, généralement similaires.**

- **Il n'ont cependant pas l'obligation de respecter le même format, et peuvent comporter des champs différents.**

- **Au delà de ce concept, on retrouve également dans MongoDB le concept de base de données.**

- **Une base de données pourra ainsi contenir plusieurs collections, elles-mêmes contenant chacune de nombreux documents (ayant potentiellement un format variable même au sein d'une même collection).**

# Fonctionnement – BSON



- **Chaque document comporte par ailleurs quoiqu'il arrive un champs interne à MongoDB: `_id`, qui est garanti comme étant unique dans la collection contenant le document.**

- **Cela veut dire que le nom de champs « _id » est en théorie réservé; on peut cependant l'utiliser si on souhaite utiliser son propre format d'identifiant (mais cela est cependant déconseillé).**

- **Par défaut, lorsqu'on insère un nouveau document, si aucun _id n'est spécifié, alors MongoDB en générera un par lui même.**

- **On peut également le spécifier explicitement; il va alors de soit que la valeur spécifiée ne doit pas déjà exister au sein de la collection (si c'est le cas, l'insertion échouera).**

# Fonctionnement – BSON



- **Par défaut, si on ne spécifie pas de valeur explicite pour le champs _id, alors MongoDB en générera un de 24 caractères alpha-numériques (en réalité 12 octets binaires, représentés sous une forme hexadécimale). Par exemple: `56563b19f4c8c7c9597d7d95`**

- **Par défaut, cet identifiant est composé:**

  - **D'un timestamp sur 4 octets (indiquant la date/heure de création).**
  - **D'un identifiant de machine sur 3 octets.**
  - **D'un identifiant de processus sur 2 octets.**
  - **D'un compteur sur 3 octets, démarrant à une valeur aléatoire.**

- **Cela veut dire que, pour tout document, on dispose d'ores et déjà d'une date de création sans avoir à créer de champs explicite.**

# Fonctionnement – BSON – Types



- **Les types principaux suivants sont disponibles au sein de BSON (il y en a d'autres):**
  - **String**
  - **Integer (32 ou 64 bits selon l'architecture)**
  - **Boolean**
  - **Double**
  - **Tableaux (structure JSON classique)**
  - **Timestamp**
  - **Date (au format ISO)**
  - **ObjectID (un identifiant unique binaire de document MongoDB)**
  - **Binary**

# Fonctionnement – BSON



- **Lorsqu'on insère un champs dans un document, par défaut, MongoDB choisira le type qui lui semble le plus approprié par rapport à la valeur spécifiée.**

- **On peut également explicitement spécifier le type, par le biais d'appel à des fonctions.**

- **Par exemple:**

```
{
    prenom: 'John',
    nom: String('Smith'),
    Note: NumberInt(5)
}
```

# Architecture



- **MongoDB a deux propriétés importantes en terme d'architecture interne:**
  - **Il peut être répliqué, pour éviter toute perte de données.**
  - **Il est extrêmement scalable, par le biais d'une approche qu'on appelle le *sharding*.**
- **L'une comme l'autre sont importantes dans le cadre d'une problématique « Big Data »; la première pour éviter toute perte de données critique, et la seconde parce qu'on est évidemment susceptible de stocker de très larges quantités de données au sein des bases de données.**

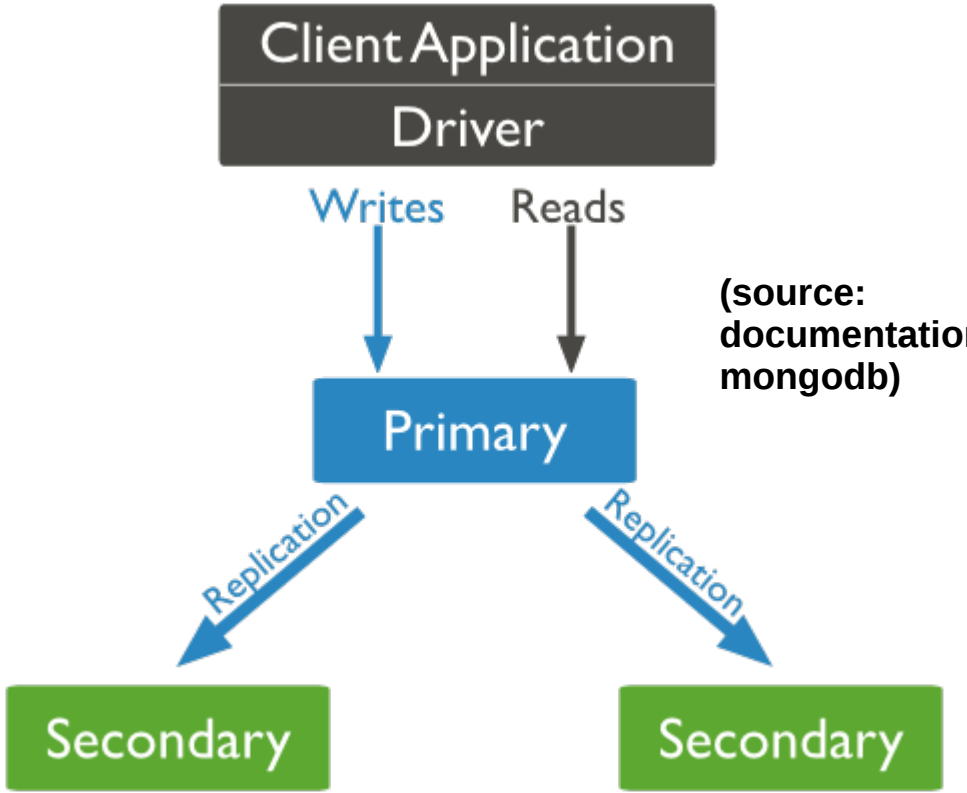
# Architecture – Réplication



- **La redondance/réplication est possible par le biais de plusieurs instances du processus serveur de MongoDB (mongod).**

- **Dans ce cas, on considère qu'une instance est *primaire* et que les autres sont *secondaires*.**

- **Les nœuds secondaires répliquent toutes les opérations effectuées sur le nœud primaire *via* un mécanisme de logs (oplog, similaire aux binary logs de MySQL).**

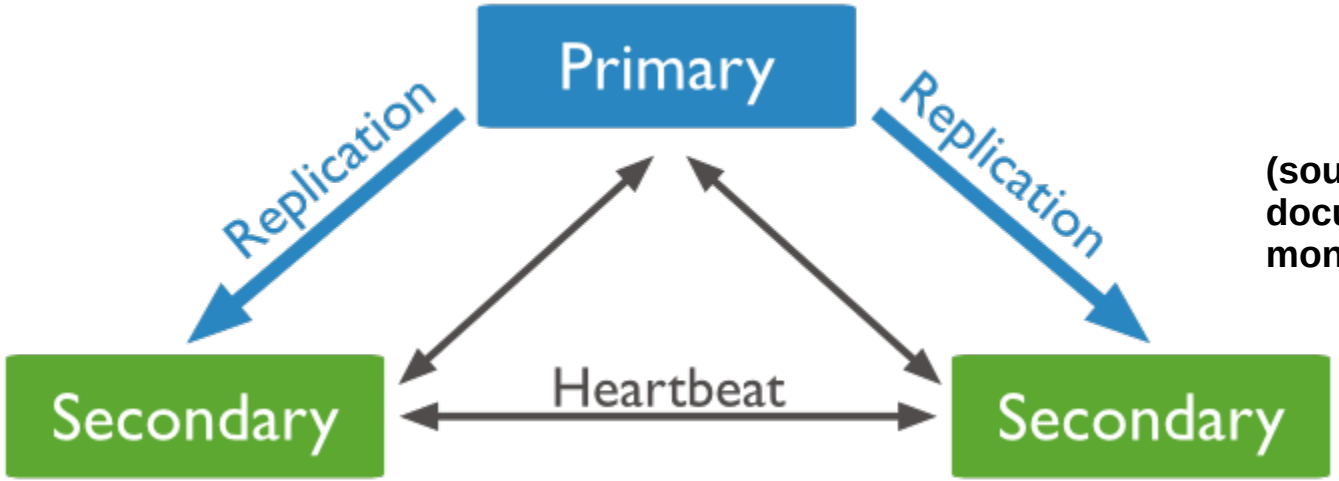



**(source: documentation mongodb)**

# Architecture – Réplication



- **Les nœuds communiquent en permanence par le biais de *heartbeats*. Si l'absence de *heartbeats* pour le nœud primaire est détecté, alors l'ensemble des nœuds restants élira un nouveau nœud parmi eux pour qu'il devienne le nouveau nœud primaire.**

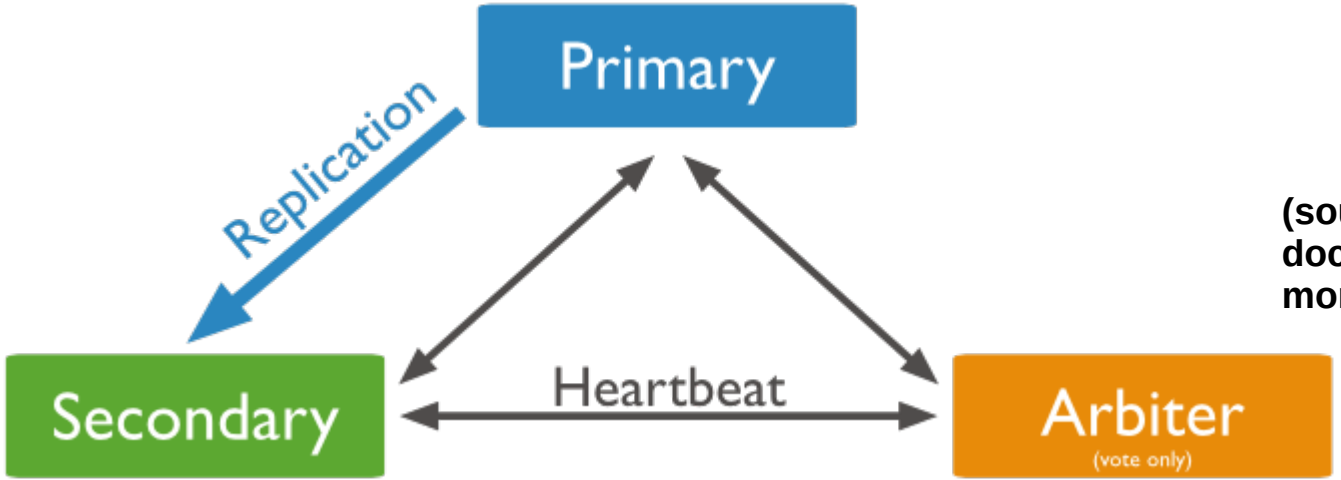



**(source: documentation mongodb)**

# Architecture – Réplication



- **On peut également définir des nœuds *arbitres*: ils s'agit de nœuds légers ne répliquant pas les opérations du nœud primaire mais visant uniquement à assurer une majorité en cas de vote égal parmi les nœuds secondaires si le nœud primaire vient à tomber.**




**(source: documentation mongodb)**

# Architecture – *Sharding*



- **Le *sharding* est la technique utilisée pour assurer la scalabilité au sein de MongoDB.**
- **Elle consiste à avoir plusieurs instances de MongoDB (et en fait généralement plusieurs groupes de nœuds primaires+secondaires) concrètement indépendantes mais logiquement « unifiées ».**
- **Chaque instance aura la charge de stocker une partie des données.**
- **L'ensemble des instances contiendra ainsi la totalité des données et sera considéré conceptuellement comme la base de données unique « logique » MongoDB.**

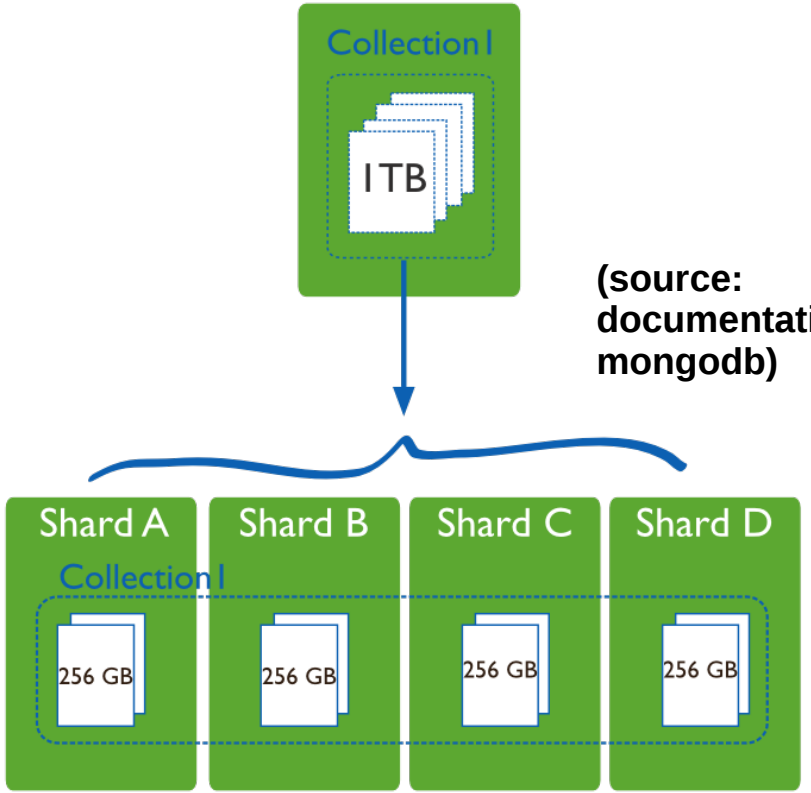
# Architecture – *Sharding*



- **Ici, on stocke 256GB de données au sein de chacune des instances MongoDB.**

- **Conceptuellement, en revanche, on considère l'ensemble de ces instances comme un seul SGBD MongoDB; et on adressera le SGBD comme s'il s'agissait d'une instance unique.**

- **C'est MongoDB qui s'occupera de rediriger les écritures/lectures vers l'instance / le *shard* approprié.**




**(source: documentation mongodb)**

# Architecture – *Sharding*



- **Afin d'assurer le *sharding*, on désignera un champs particulier des documents concernés (par exemple, mais pas nécessairement et même pas optimalement l'ObjectID des documents) dont la valeur déterminera le *shard* qui devra contenir le document. On désigne ce champs sous le terme de *shard key*.**

- **Pour déterminer le *shard* à adresser, on pourra utiliser deux approches:**

  - **Une approche basée sur la valeur elle-même, par exemple un ID numérique. On pourrait ainsi par exemple configurer le *cluster* de telle sorte que les ID de 0 à 10000 soients stockés sur le *shard* 1, les IDs de 10001 à 20000 sur le *shard* 2, etc.**

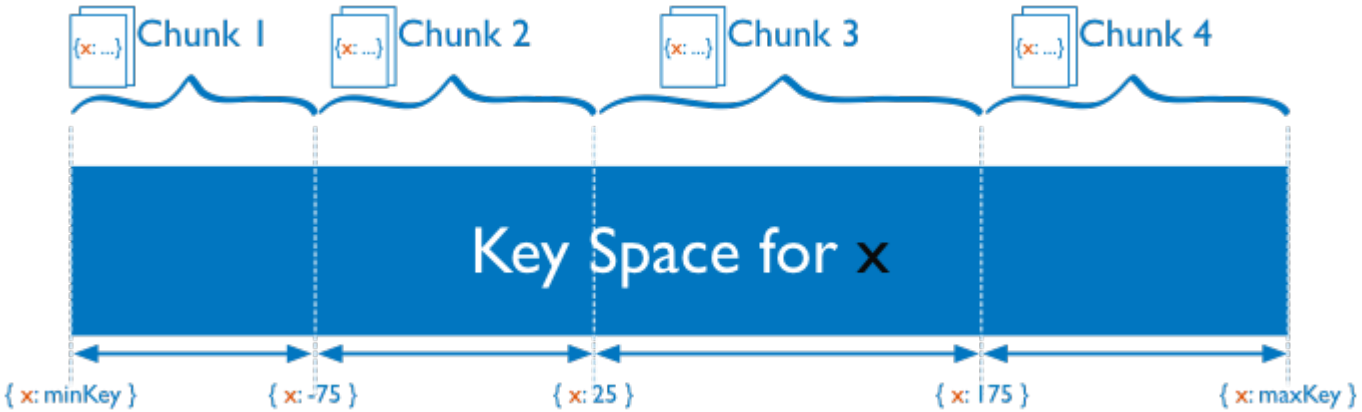
  - **Une approche basée sur un *hash* de la valeur, afin d'assurer une meilleure répartition dans l'espace des possibilités.**

# Architecture – *Sharding*



- **Si on choisi de se baser sur la valeur, alors deux valeurs proches seront probablement au sein du même *shard*.**

- **Selon les valeurs rencontrées, l'utilisation et le champs choisi, la répartition sera possiblement variable entre les *shards*.**



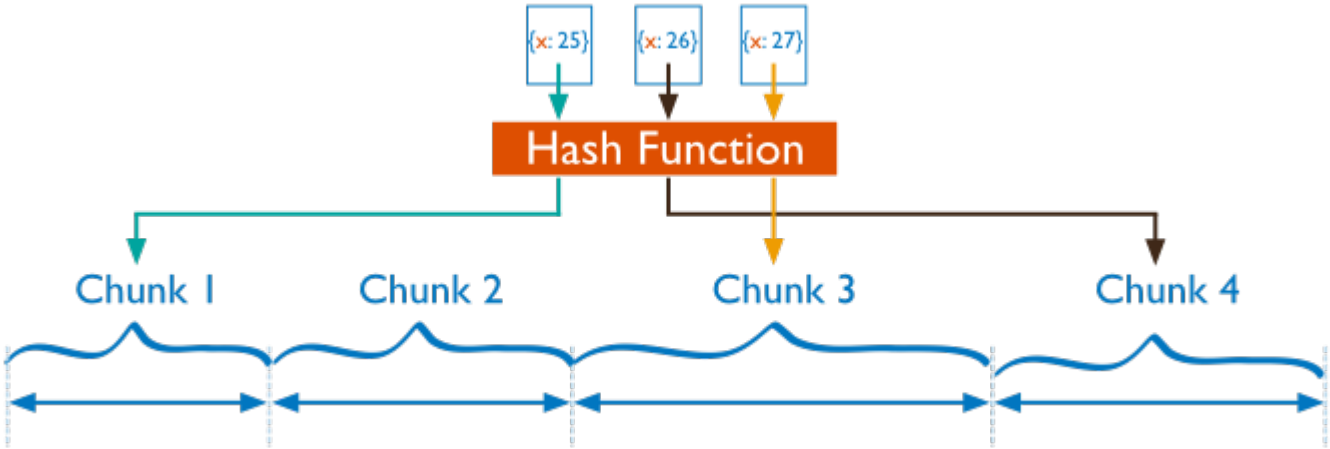

**(source: documentation mongodb)**

# Architecture – *Sharding*



- **Si en revanche on choisi de se baser sur un *hash* de la valeur de la *shard key*, alors deux valeurs proches seront probablement dans des *shards* différents, puisque la fonction de *hashing* répandra les valeurs dans l'espace de possibilités.**

- **La répartition entre les *shards* sera alors plus optimale.**




**(source: documentation mongodb)**

# Architecture – *Sharding*



- **En terme d'architecture logicielle, MongoDB assure cette mécanique de *sharding* par le biais de trois composantes:**

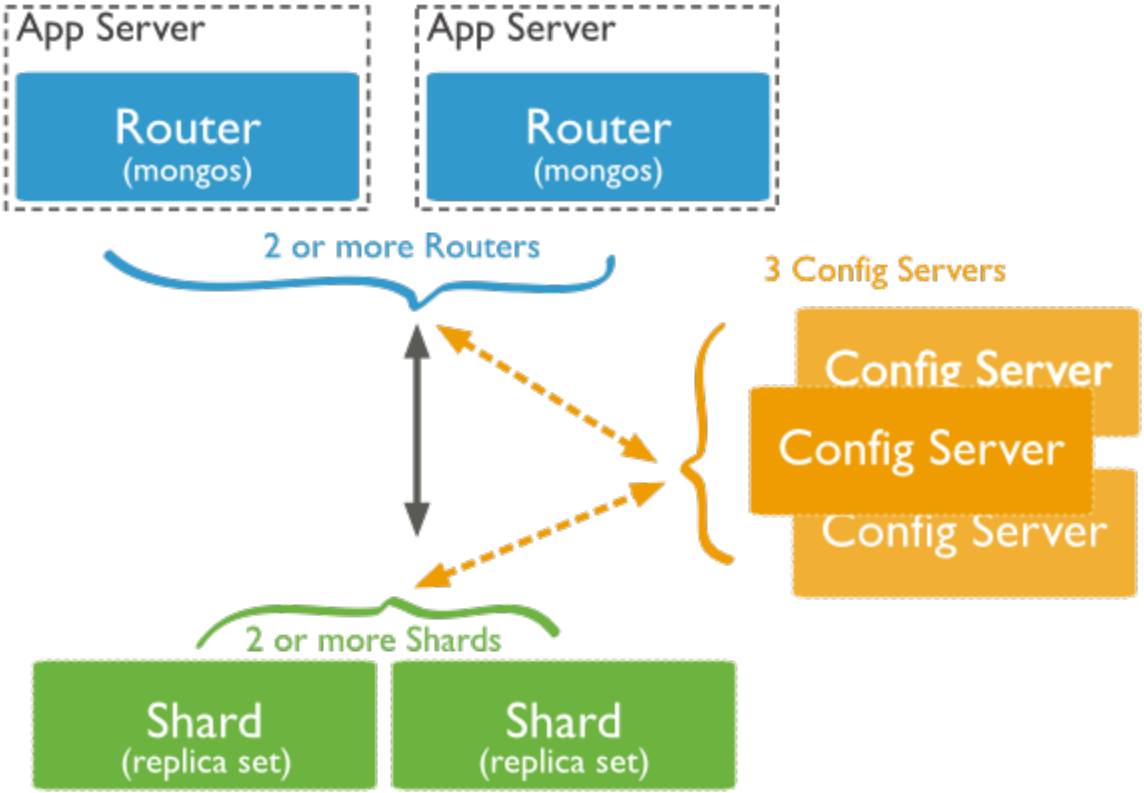
  - **Un ou plusieurs *routeurs de requêtes*. Il s'agit de serveurs indépendants (processus mongos); c'est sur ces serveurs que les clients se connecteront, et ce sont ces serveurs qui redirigeront les écritures et lectures sur le bon *shard*.**

  - **Un ensemble de trois *serveurs de configuration*. Ils constituent ensemble la table de *mappage* assurant l'associations entre les valeurs de la clef de *sharding* et le *shard* à adresser, et sont en communication avec les routeurs et les *shards*.**

  - **Enfin, les *shards* eux-mêmes (composé chacun à minima d'une instance de serveur MongoDB mais plus généralement d'un ensemble d'instances primaires et secondaires pour assurer la réplication).**

# Architecture – *Sharding*






**(source: documentation mongodb)**

# Collections et documents – Exemple



- **Imaginons qu'on souhaite stocker la liste des élèves d'une promotion avec, pour chacun d'entre eux:**
  - **Des informations générales (nom, prénom, date de naissance, etc.).**
  - **Une liste de notes obtenues à des examens.**
  - **Une liste de remarques liées au dossier scolaire.**
- **Sur un SGBD classique, on stockerait ces informations par le biais d'au moins trois tables, par exemple « eleves », « notes », et « remarques »; avec une jointure assurée par exemple par un identifiant numérique situé dans la table « eleves » et référencé au sein des deux tables « notes » et « remarques » (par exemple *via* un champs id_eleve).**

# Collections et documents – Exemple



- **Si on souhaite en revanche stocker ces données au sein de MongoDB (et d'une manière générale dans un SGBD NoSQL *document-based*), on utilisera à la place probablement simplement une seule collection « eleves »; dans laquelle on inclura un document pour chacun des élèves et qui contiendra toutes les informations (y compris les notes et les remarques, à quantité variable).**

- **Les documents auraient alors probablement par exemple une structure de ce type:**

```
{
   prenom: 'John',
   nom: 'Smith',
   ...
   notes: [12, 14.5, 8.5, 17],
   remarques: ["retard le 12/10", "redoublement envisagé"]
}
```

# Documents – Relations



- **L'approche montrée dans l'exemple précédent constitue déjà une forme d'implémentation d'une relation (ici 1:N entre un élève et ses notes/ses remarques).**

- **On peut également au sein d'un document référencer un autre document. Pour ce faire, on pourra plutôt que d'inclure les données directement au sein du même document spécifier une série d'ObjectIDs: ceux des adresses au sein d'une autre collection adresse_eleves, par exemple.**

- **On pourrait même spécifier une base de données alternative au sein d'un autre champs; ou encore se servir d'un champs ne contenant pas un ObjectID mais bien une valeur « classique », qui soit unique et qui nous permette de récupérer les informations par le biais de la relation.**

# Indexs



- **MongoDB supporte également la définition d'indexs.**
- **Ces indexs peuvent être définis sur un seul champs, ou encore être constitué de la combinaison de plusieurs champs.**
- **Ils auront les avantages et inconvénients habituels:**
  - **Fortes améliorations de performance pour la lecture si on se sert de ces champs pour accéder aux enregistrements.**
  - **Baisse de performance en écriture puisque l'index doit être modifié lorsqu'on ajoute ou modifie un enregistrement.**

# Le *shell* MongoDB



- **Un peu comme MySQL/MariaDB (et beaucoup d'autres SGBD), MongoDB est livré avec un logiciel client console: le *shell* MongoDB.**

- **C'est cet outil qu'on utilisera souvent pour créer des collections, les consulter, effectuer des tâches de maintenance, consulter l'état actuel des bases de données et des collections qu'elles contiennent, etc.**

- **Ce *shell* est accessible par le biais de la commande:**

  ```
  mongo
  ```

  **(on peut par ailleurs spécifier des options variées indiquant comment se connecter à l'instance MongoDB visée, avec quel utilisateur, etc.)**

# Le *shell* MongoDB



- `db.dropDatabase()`
  **Pour supprimer la base de données courante.**

- `db.createCollection(COLLECTION)`
  **Pour créer une nouvelle collection.**

- `db.COLLECTION.drop()`
  **Pour supprimer la collection COLLECTION.**

- `db.COLLECTION.insert(DOCUMENT_BSON)`
  **Pour insérer le document spécifié dans la collection COLLECTION. Si la collection n'existe pas, elle sera créée. On peut donc créer une collection également de cette manière.**

# Le *shell* MongoDB



- `show collections`
  **Pour voir les collections disponibles dans la base de données courante.**

- `db.COLLECTION.find()`
  **Pour afficher les documents contenus dans la collection COLLECTION.**

- `db.COLLECTION.find().pretty()`
  **Pour afficher les documents contenus dans la collection COLLECTION, formattés d'une manière plus humainement lisible (indentés).**

- `db.dropDatabase()`
  **Pour supprimer l'intégralité de la base de données courante.**

# Le *shell* MongoDB



- **La méthode find() permet également d'appliquer des traitements de recherche équivalents au WHERE du langage SQL des SGBD relationnels.**

- **Pour effectuer une recherche WHERE sur un ou plusieurs champs, on utilisera la syntaxe:**

```
db.COLLECTION.find({FIELD1: "VALUE1", FIELD2: "VALUE2", …});
```

- **Par exemple:**

```
db.eleves.find({nom: 'Smith', prenom: 'John'});
```

# Le *shell* MongoDB



- **On peut également utiliser l'opérateur $or pour indiquer un OU logique, sur le modèle:**

```
db.COLLECTION.find({$or: [ {FIELD1: VALUE1},
                           {FIELD2: VALUE2} ]})
```

- **Par exemple:**

```
db.eleves.find({$or: [ {prenom: 'John'},
                       {prenom: 'Bob'} ]})
```

# Le *shell* MongoDB



- **Plutôt que d'indiquer une valeur fixe dans ces instructions conditionnelles, on peut également utiliser des opérateurs tel que « supérieur à », « inférieur à », etc.. ils respectent la syntaxe suivante:**

```
{FIELD: {OPERATOR:VALUE}}
```

- **Et on dispose des opérateurs suivantes:**

  - `$gt:` **plus grand que.**
  - `$gte:` **plus grand ou égal à.**
  - `$lt:` **plus petit que.**
  - `$lte:` **plus petit ou égal à.**
  - `$ne:` **différent de.**

# Le *shell* MongoDB



**… par exemple:**

```
db.eleves.find({age: {$gte:30}})
```

**… pour sélectionner les élèves ayant 30 ans ou plus, ou encore:**

```
db.eleves.find({$or: [ {age:{$lt:20}},
                       {section:{$ne:'MBDS'}} ]})
```

**… pour sélectionner les élèves ayant moins de 20 ans OU tous ceux n'étant pas dans la section « MBDS ».**

# Le *shell* MongoDB



- **On peut également indiquer, par le biais d'un second argument à find(), quels champs on souhaite sélectionner. Si ce second argument n'est pas spécifié, tous les champs du document sont renvoyés.**

- **La syntaxe:**

  `.find({…},{FIELD1:[0|1],FIELD2:[0|1],…})`
  **(par défaut, tous les champs sont à 1)**

- **Par exemple:** `db.eleves.find({}, {prenom:0,nom:0})`

  **… pour sélectionner tous les élèves mais sans extraire leurs noms et prénoms.**

# Le *shell* MongoDB



- **On peut également appliquer un équivalent de l'opérateur LIMIT par le biais des fonctions limit() et skip() à appliquer respectivement à find() et à limit(). Syntaxe:**

  `.find(…).limit(N)` **… pour sélectionner les `N` premiers documents.**

  `.find(…).limit(N).skip(M)`
  **… pour sélectionner les `N` premier documents à partir du `M`ième document.**

- **Par exemple:**

```
db.eleves.find().limit(10).skip(5)
```

# Le *shell* MongoDB



- **On peut également trier les documents résultant de notre recherche. Pour ce faire, on va utiliser la fonction .sort() à appliquer à find(). Syntaxe:**

```
.find(…).sort({FIELD1:[1|-1], FIELD2:[1|-1], …)
```

**…où 1 designe un ordre ascendant et -1 un ordre descendant.**
**Par exemple:**

```
db.eleves.find().sort({prenom:1, nom:-1})
```

**… qui serait l'équivalent d'un « `ORDER BY prenom ASC, nom DESC` » en SQL.**

# Le *shell* MongoDB



- **Pour mettre à jour un document, on utilisera la méthode update(). Syntaxe:**

```
db.COLLECTION.update({CRITERIA},
                     {$set: {FIEL1:VAL1, FIEL2:VAL2, ...}})
```

**Par exemple:**

```
db.eleves.update({eleve_id:'781638'},{$set:{prenom:
'Smith'}})
```

**Ou encore:**

```
db.eleves.update({$or:[{nom:'Smith'},{prenom:'John'}]},
                 {$set:{notes:[12, 13, 14.5]}})
```

# Le *shell* MongoDB



- **Par défaut, si le critère spécifié correspond à plusieurs enregistrements, MongoDB n'en mettra à jour qu'un seul.**

- **Si on souhaite qu'il mette à jour tous les documents correspondant au critère, on devra définir un troisième argument « multi » à true:**

```
db.COLLECTION.update({CRITERIA},
                     {$set: {FIEL1:VAL1, FIEL2:VAL2, …}},
                     {multi:true})
```

**Exemple:**

```
db.eleves.update({year:'2015'},{$set:{section: 'MBDS'}},
                 {multi:true})
```

# Le *shell* MongoDB



- **On peut également remplacer un document en utilisant la méthode save(). Sa syntaxe:**

```
db.COLLECTION.save(DOCUMENT_BSON)
```

**… si le document spécifié contient un identifiant _id et que celui-ci existe déjà dans la collection, alors le document ayant cet identifiant sera remplacé par le document spécifié.**

- **Si en revanche aucun identifiant _id n'est spécifié ou si il est spécifié mais que cet identifiant n'existe pas dans la collection, alors un nouveau document sera inséré.**

# Le *shell* MongoDB



- **On peut également définir un index; on utilise pour ce faire la fonction ensureIndex(). Syntaxe:**

```
db.COLLECTION.ensureIndex({FIELD1:[1|-1],FIELD2:[1|-1]}, …)
```

**… en spécifiant 1 pour un index ascendant et -1 pour un index descendant sur les champs spécifié. Si on spécifie plusieurs champs, un index multiple sur ces champs cumulés est créé.**

**Par exemple:**

```
db.eleves.ensureIndex({nom:1,prenom:1})
```

# Le *shell* MongoDB



- La fonction ensureIndex() peut également prendre un second argument, contenant une liste d'options, sur le format:

```
.ensureIndex({...}, {OPTION1:VALUE1, OPTION2:VALUE2, …}
```

- Avec les plus importantes:
  - `background` (true/false): construit l'index en fond, sans appliquer de lock bloquant sur la collection. Par défaut, false.
  - `unique` (true/false): indique que l'index spécifié est unique; autrement dit, que le champs ou l'ensemble de champs spécifiés ne peuvent pas avoir deux fois la même valeur au sein de la collection. Par défaut: false.
  - `name` (une string): donne un nom à l'index. Par défaut: auto-généré.

# Le *shell* MongoDB



- **Enfin, on peut également obtenir la liste des indexs propre à une collection avec la fonction:**

```
db.COLLECTION.getIndexes()
```

- **La fonction renverra par ailleurs toutes les options relatives à l'index (unique, name, etc.).**

# Le *shell* MongoDB



- **Enfin, pour supprimer un élément, on utilisera la fonction remove(). Syntaxe:**

```
db.COLLECTION.remove({CRITERE},[0|1])
```

**… avec le premier argument un critère de suppression, et le second indiquant s'il est positionné à 1 qu'un seul enregistrement doit être supprimé même si plusieurs enregistrements correspondent au critère.**

**Par exemple:**

```
db.eleves.remove({nom:'John',prenom:'Smith'},0)
```

# Le *shell* MongoDB



- **Un autre opérateur utilisable au sein des instructions conditionnelles est l'opérateur `$in`. Il correspond au « IN » de SQL. Sa syntaxe:**

```
{FIELD: {$in: [VALUE1, VALUE2, VALUE3, …]}}
```

- **Par exemple:**

```
db.eleves.find({section:{$in:  ['MBDS', 'MIAGE']}})
```

**… pour obtenir tous les documents dont le champs « section » est « MBDS » ou « MIAGE ».**

# Le *shell* MongoDB



- **Enfin, on peut également stocker les résultats de nos requêtes dans des variables, par le biais de la syntaxe:**

```
var VARNAME=db.COLLECTION.find(…)
```

- **Et ensuite accéder aux champs du résultat par le biais de la syntaxe:**

```
VARNAME[X][FIELDNAME]
```

**… pour obtenir le champs FIELDNAME de la ligne de résultat X.**

# Le *shell* MongoDB



- **On peut d'ailleurs utiliser cette possibilité pour référencer au sein de nouvelles requêtes des champs issues de résultats de requêtes précédentes.**

- **Imaginons qu'on ait une collection « eleves » dont les documents contiennent un tableau « examen_ids » référençant des identifiants d'une seconde collection « examens ».**

```
{
        "prenom" : "John",
        "nom" : "Smith",
        ...
        "examen_ids": [ObjectId("565662cbf4c8c7c9597d7d99"),
                       ObjectId("565333cbf4c8c7c9597d7d99") ]
}
```

# Le *shell* MongoDB



- **On pourrait alors faire par exemple:**

```
var data=db.eleves.find({nom: 'Smith', prenom: 'John'})
```

**Suivi de:**

```
var exams=db.examens.find({_id:{"$in":data['examen_ids']}})
```

**… pour obtenir les documents de la collection « examens » pour l'élève « John Smith », et les stocker dans une variable « exams ».**

# MongoDB – Java



- **Comme indiqué précédemment, MongoDB propose entre autres une API Java (mise à disposition sous la forme d'un .jar) pour accéder à une instance MongoDB depuis un programme Java.**

- **Cette API est téléchargeable depuis le site officiel.**

- **D'une manière générale, toutes les fonctions vues précédemment *via* le shell sont également utilisables en Java.**

# MongoDB – Java



- **Avant toute opération, on devra d'abord créer un objet de type `MongoClient`. La classe en question:**

```
com.mongodb.MongoClient
```

- **Par exemple:**

```
MongoClient client = new MongoClient();
```

- **On pourra ici également spécifier l'IP/nom d'hôte de l'instance à laquelle on souhaite se connecter, et le port TCP qui lui correspond. Par exemple:**

```
MongoClient client = new MongoClient("localhost" , 27017);
```

# MongoDB – Java



- **Ensuite, on sélectionnera la base de données voulue par le biais de la méthode `getDatabase()` de cet objet, qui renvoie un objet de type `MongoDatabase`. Par exemple:**

```
MongoClient client = new MongoClient("localhost" , 27017);
MongoDatabase db = client.getDatabase("universite");
```

**… ici pour sélectionner la base de données « universite ».**

- **A noter que la fonction est strictement identique au « use » du *shell* MongoDB; et qu'une base de données non existante sera donc automatiquement « créée ».**

# MongoDB – Java – Formater les données



- D'une manière générale, le type utilisé pour formater les documents MongoDB est la classe `org.bson.Document`. C'est celle qu'on utilisera pour construire un document BSON; qu'il s'agisse d'un document au sens propre ou d'un argument aux différentes fonctions de MongoDB (par exemple find(), ou encore sort()).

- La méthode la plus utilisée de cette classe est la méthode `append()`. On fera par exemple:

```
Document doc=new Document().append("nom", "Smith");
doc.append("prenom", "John");
```

- Ce qui créerait un document:

```
{nom: "Smith", prenom: "John"}
```

# MongoDB – Java – Formater les données



- **On peut évidemment inclure des documents à l'intérieur de documents; et également inclure des tableaux (toujours avec la méthode `append()`).**

- **Si on souhaite inclure un identifiant _id de type ObjectId, on utilisera la classe org.bson.types.ObjectId. Son constructeur permet entre autre d'auto-générer un identifiant, ou encore de spécifier explicitement en hexadécimal un identifiant. Par exemple:**

```
ObjectId id1=new ObjectId();
ObjectId id2=new ObjectId(5656f0c1c3b1180e59f84ad7);
```

- **Là aussi, on pourra l'inclure au sein d'un document par le biais de la méthode `append()`.**

# MongoDB – Java – Formater les données



- **Si par exemple on souhaite construire le document suivant:**

```
{
        "prenom" : "John",
        "nom" : "Smith",
        "addresse" : {
                      "rue": "Sidi belyout",
                      "numero": "9",
                      "ville": "Casablanca"
                     },
        "notes" : [ 14.5, 12, 15 ],

        "examen_ids" : [ ObjectId("5656f171c3b1180e924e2403"),
                         ObjectId("5656efc9c3b1180e26bc8961") ]
}
```

# MongoDB – Java – Formater les données



- **On fera:**

```
import static java.util.Arrays.asList;

/* ... */

new Document().append("prenom", "John")
              .append("nom", "Smith")
              .append("adresse", new Document()
                                     .append("rue": "Sidi belyout")
                                     .append("numero": 9)
                                     .append("ville": "Casablanca"))
              .append("notes", asList(14.5, 12, 15))
.append("examen_ids", asList(new ObjectId("5656f...4e2403"),
                             new ObjectId("5656e...26bc8961"));
```

# MongoDB – Java – Insertion



- **Pour sélectionner une collection, on utilisera la méthode `getCollection()` de notre objet `MongoDatabase`, qui renvoie un objet de type `com.mongodb.client.MongoCollection`.**

- **Par exemple:**

```
db.getCollection("eleves");
```

- **On pourra ensuite insérer un document dans la collection en utilisant la méthode `insertOne` (entre autres). Elle prends un argument: le document BSON à insérer, lui-même de type `org.bson.Document`.**

# MongoDB – Java – Sélection



- Pour obtenir un document depuis une collection, on utilisera la méthode `.find()` d'un objet `MongoCollection`. Cette méthode renvoie un Iterable MongoDB de type `com.mongodb.client.FindIterable;` sur lequel on pourra itérer pour accéder aux différents éléments du document.

- Elle est strictement identique à la méthode `find()` du shell; et prends en paramètre les mêmes arguments (eux-mêmes des documents BSON).

- Par exemple:

```
FindIterable<Document> it =
db.getCollection("eleves").find();
```

# MongoDB – Java – Sélection



- **On peut également utiliser toutes les fonctions vues précédemment dans le *shell* MongoDB: sort(), limit(), skip(), et les opérateurs utilisables dans ces fonctions: $or, $ne, $gt, $gte, etc.**

- **Exemples:**

  - `.find(new Document("nom", "Smith"));`

  - ```
    .find(new Document("age", new Document()
                               .append("$gt", 30));
    ```

  - ```
    .find(new Document(), new Document().append("nom",
    0).append("prenom", 0));
    ```

# MongoDB – Java – Sélection



- `.find().sort(new Document().append("prenom", 1)`
  `.append("nom", -1))`

- `.find().skip(5).limit(10)`

- **D'une manière générale, toutes les commandes vues précédemment dans le *shell* MongoDB sont utilisables de cette manière dans l'API Java.**

# MongoDB – Java – Sélection



- **Pour itérer sur les résultats d'un find(), on appliquera un traitement de type com.mongodb.Block à chacun des enregistrements par le biais de la méthode forEach de l'object FindIterable renvoyé par la méthode find(). Par exemple:**

```
FindIterable<Document>
iterable=db.getCollection("eleves").find();

iterable.forEach(new Block<Document>() {
    @Override
    public void apply(final Document document) {
        System.out.println(document);
    }
});
```

# MongoDB – Java – Sélection



- **On peut également obtenir un champs précis avec la méthode get() d'un objet Document.**

- **Par exemple:**

```
FindIterable<Document>
iterable=db.getCollection("eleves").find();

iterable.forEach(new Block<Document>() {
    @Override
    public void apply(final Document document) {
        System.out.println(document.get("prenom"));
    }
});
```

# MongoDB – Java – Mise à jour



- **Le pendant Java de la méthode `update` du *shell* vue précédemment est la méthode `updateOne`. Elle prend les mêmes arguments: un critère de sélection (lui-même un document), et les champs à mettre à jour (même remarque).**

- **Ainsi, l'équivalent de:**
  `db.eleves.update({eleve_id:'781638'},{$set:{prenom: 'Smith'}})`

- **Serait en Java:**

```
db.getCollection("eleves").updateOne(new Document()
                    .append("eleve_id", "781638"),
                    new Document("$set", new Document()
                                .append("prenom", "smith")));
```

# MongoDB – Java – Mise à jour



- **On dispose également d'une méthode `updateMany()`, qui correspond à l'appel d'une commande *shell* `update` mais avec l'option « `multi` » définie à true.**

- **L'une comme l'autre de ces méthodes renvoient un objet de type `com.mongodb.client.result.UpdateResult`, qui permet d'obtenir des informations sur l'opération de mise à jour.**

- **Entre autre chose, on peut appeler la fonction getModifiedCount() qui renvoie le nombre d'enregistrements ayant été modifiés.**

# MongoDB – Java – Suppression



- **Pour supprimer un document, on dispose des deux fonctions `deleteOne()` et `deleteMany()` - toujours sur l'objet MongoCollection. Elles correspondent à la fonction remove du shell, avec son deuxième argument respectivement à 1 et à 0.**

- **Par exemple:**

```
db.getCollection("eleves").deleteOne(new Document()
                          .append("eleve_id", "781638"));
```

**… qui correspondrait à:**

```
db.eleves.remove({eleve_id: "781638"},1)
```

# MongoDB – Java – Exemple



- **Un exemple complet de programme Java qui se connecte à une instance MongoDB, insère un enregistrement dans une collection; puis récupère l'intégralité des documents de la collection pour les afficher à l'écran:**

```
import com.mongodb.MongoClient;
/*...*/
import static java.util.Arrays.asList;

public class Mongo
{
  // Le main du programme.
  public static void main(String[] args) throws Exception
  {
    MongoClient mongoClient = new MongoClient();
    MongoDatabase db = mongoClient.getDatabase("universite");
```

# MongoDB – Java – Exemple



```
    db.getCollection("eleves").insertOne(new Document()
                                        .append("nom", "Smith")
                                        .append("prenom", "John")
                                .append("notes", asList(14, 12.5, 15)));
    FindIterable<Document> it = db.getCollection("eleves")
                    .find().sort(new Document().append("_id", 1));
   it.forEach(new Block<Document>() {
    @Override
    public void apply(final Document document) {
      System.out.println(document.get("prenom")
                              +" "+document.get("nom"));
    }
    });
  }
}
```