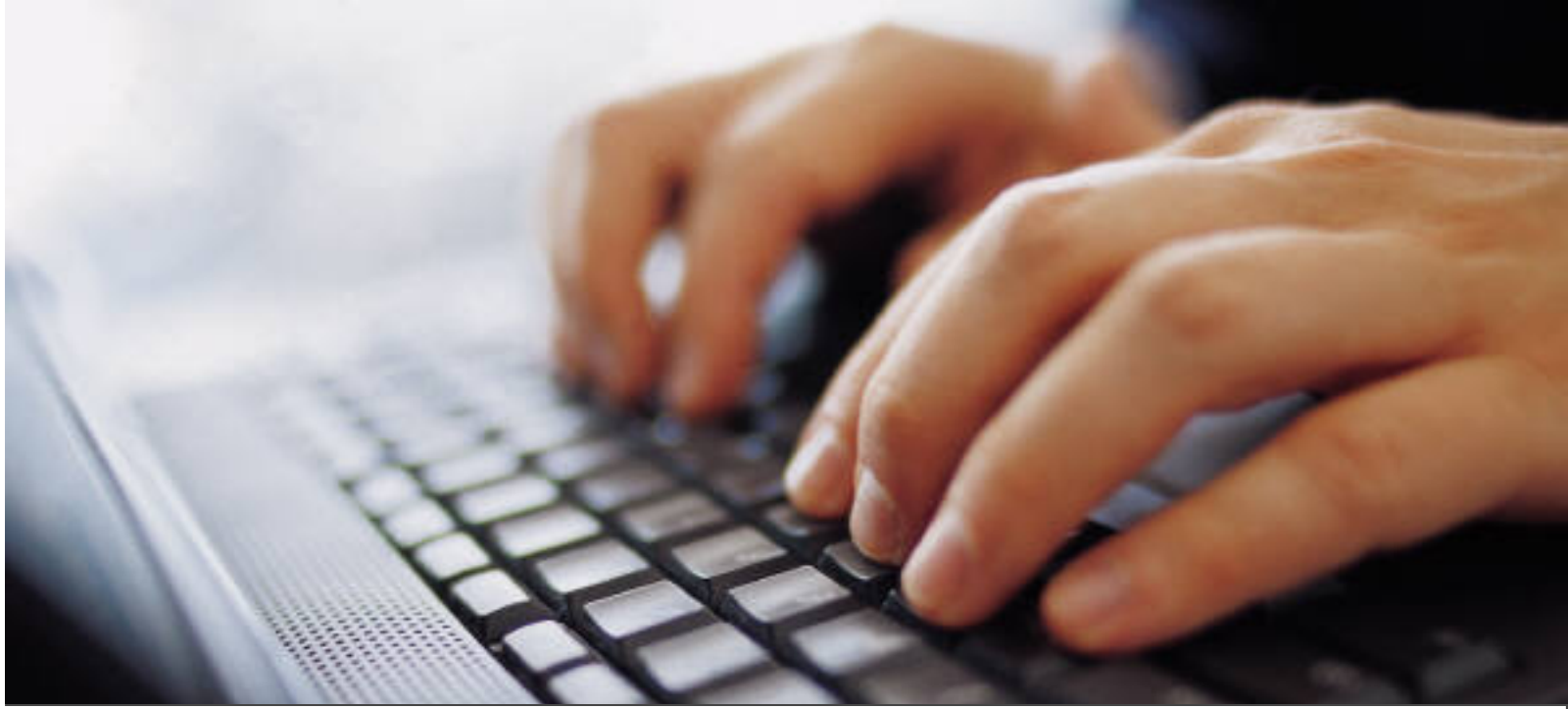
Hadoop / Big Data

Benjamin Renaut <renaut.benjamin@tokidev.fr>

MBDS

2019 - 2020



11

MongoDB et map/reduce

MongoDB: fonction mapreduce

- **Au delà des points d'intégration Hadoop qui vont être abordés par la suite, MongoDB possède également une fonction mapreduce() interne.**

- **Elle est relativement limitée (essentiellement par le type de traitement qu'on peut effectuer sur les données), mais peut avoir son utilité.**

- **Sa syntaxe est la suivante:**

```
db.COLLECTION.mapreduce(FONCTION_MAP,
                        FONCTION_REDUCE,
                        {query:{CONDITIONS},
                         out: "NOM_SORTIE"})
```

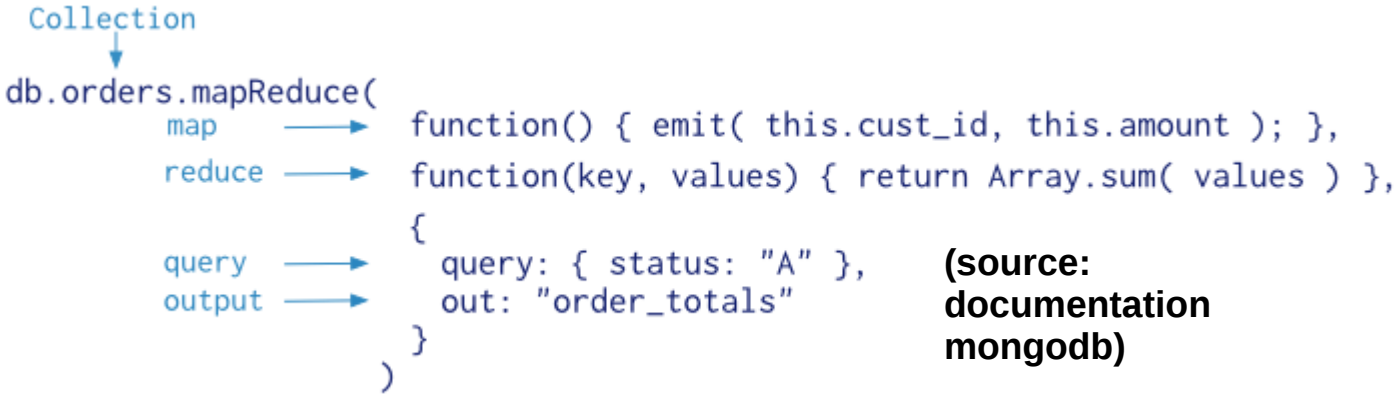
MongoDB: fonction mapreduce

- **Imaginons qu'on ait par exemple une collection « orders »; contenant les documents suivants:**

(source: documentation mongodb)

MongoDB: fonction mapreduce

- **On va exécuter la commande suivante:**



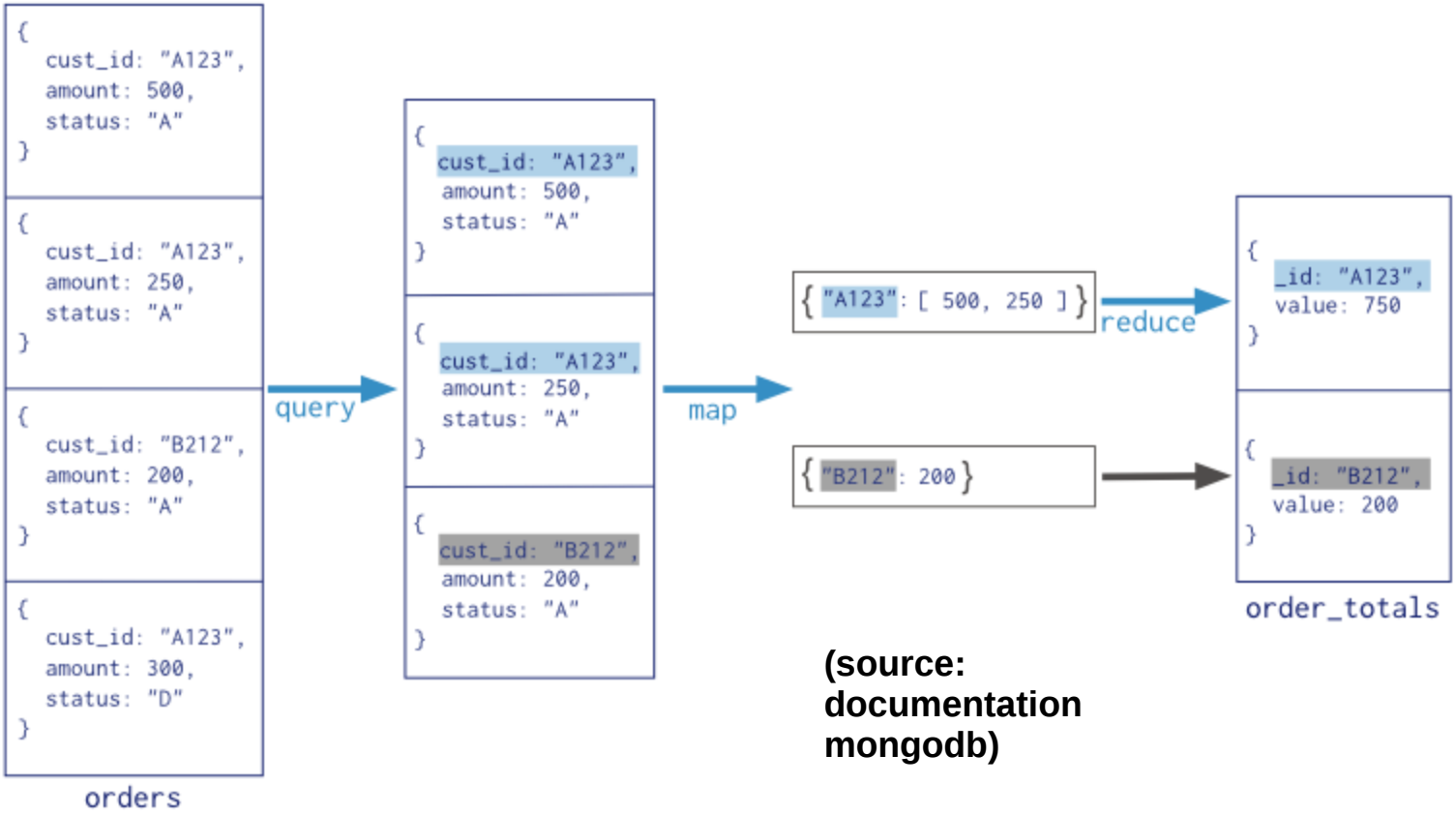
(source: documentation mongodb)

… afin d'appliquer un traitement map/reduce aux documents dont le champs « status » vaut « A »; on générera des couples (clef;valeur) contenant l'identifiant client et le total de la commande dans map, et on renverra la somme de tous les totaux obtenus après *shuffle* pour chaque clef distincte dans reduce.

MongoDB: fonction mapreduce

- **Le traitement et son résultat:**

(lui-même stocké dans un champs order_totals)



(source: documentation mongodb)

MongoDB: connecteur Hadoop

- **MongoDB dispose d'un connecteur Hadoop. Ce connecteur vise à permettre à des programmes map/reduce Hadoop de directement adresser MongoDB comme source des données d'entrée du programme ou comme destination des données de sorties.**

- **Ce connecteur est lui-aussi Open Source (licence AGPLv3). Il n'est en revanche pas inclus par défaut dans la distribution de MongoDB.**

- **URL du *repository* du projet:**

  https://github.com/mongodb/mongo-hadoop

  (un sous-dépôt du dépôt principal github MongoDB)

MongoDB: connecteur Hadoop

- **Ce connecteur est disponible au sein d'une librairie Java .jar, et téléchargeable depuis le dépôt Github.**

- **Il a par ailleurs pour dépendance la librairie fournissant l'API Java de connexion à MongoDB (la même que celle qui a été utilisée lors du TP4).**

- **Le connecteur fonctionne en mettant à disposition du développeur des formats Hadoop supplémentaires d'entrée et de sortie, permettant de directement lire et écrire au sein de bases/collections MongoDB plutôt que depuis/sur HDFS.**

- **L'inclusion de la librairie (et de sa dépendance) sera donc nécessaire au sein du projet Java map/reduce si on souhaite s'en servir.**

MongoDB: connecteur Hadoop

- **La librairie propose par ailleurs un *helper*: une classe, MongoTool, dont on va hériter dans la classe principale (la classe driver) du programme Hadoop afin de simplifier la configuration du programme et la spécification des points d'entrée et de sortie MongoDB.**

- **La librairie recommande par ailleurs d'utiliser un *wrapper* Hadoop standard au sein du *main* du programme pour le lancer, et de réduire ce *main* à une ligne unique qui se charge de lancer le programme en instanciant un objet de la classe Driver.**

- **En conséquence, c'est dans le *constructeur* de la classe driver qu'on va effectuer les opérations usuellement effectuées dans la fonction *main*: configuration Hadoop, spécification des points d'entrée et de sortie, etc.**

MongoDB: connecteur Hadoop

- **Une classe driver modèle aura donc une définition similaire à celle-ci:**

```
public class Driver extends MongoTool
```

- **Et le *main* de cette classe aura le modèle suivant:**

```
public static void main(String[] args) throws Exception
{
  System.exit(ToolRunner.run(new Driver(), args));
}
```

… qui se charge de lancer le programme via une instanciation de la classe driver, en s'occupant d'une partie des opérations automatiquement (par exemple la lecture des arguments Hadoop).

MongoDB: connecteur Hadoop

- **C'est dans le constructeur qu'on va spécifier les diverses options de configuration.**

- **En général, on commencera d'abord par créer un objet Configuration (comme cela a été le cas jusqu'ici):**

```
Configuration conf=new Configuration();
```

… qui sera utilisé pour plusieurs opérations par la suite.

- **Ensuite, on utilisera une seconde classe du connecteur pour spécifier toutes les options. Cette classe est `MongoConfigUtil`. Elle est statique (elle n'a donc pas besoin d'être instanciée).**

MongoDB: connecteur Hadoop

- La classe `MongoConfigUtil` propose la plupart des méthodes offertes d'ordinaire par la classe `Job` désignant le programme map/reduce, mais avec des noms plus simples (par exemple, `setMapperClass` devient `setMapper`).

- D'une manière générale, toutes les méthodes de configuration appelées prennent en premier argument l'objet `Configuration` créé précédemment.

- Une fois que la configuration est terminée, par ailleurs, on appellera la méthode:
  `setConf(OBJET_CONFIGURATION)` … de la classe mère. Cette méthode se chargera d'appliquer la configuration après modifications.

MongoDB: connecteur Hadoop

- **Les différentes méthodes suivantes sont applicables à la classe d:**
  - **`setMapper()` pour définir la classe Mapper.**
  - **`setReducer()` pour définir la classe Reducer.**
  - **`setOutputKey()` pour définir le type de clef en sortie du programme.**
  - **`setOutputValue()` pour définir le type de valeur en sortie du programme.**
  - **`setMapperOutputKey()` - définir le type de clef en sortie de Map.**

MongoDB: connecteur Hadoop

- **`setMapperOutputValue()` pour définir le type de clef en sortie de Reduce.**

- **`setInputFormat()` pour définir le type de format d'entrée du programme.**
  Ce type constitue une classe héritant de la classe Hadoop `FileInputFormat`, et indique à Hadoop la classe en charge de charger les données d'entrée (une possibilité jusqu'ici non utilisée, y compris dans les Tps).
  Par exemple, par défaut et si on ne spécifie pas de type explicite, alors le type utilisé est `TextInputFormat`, qui est la classe de chargement par défaut que nous avons utilisé jusqu'ici.
  Pour lire les données directement depuis MongoDB, on utilisera à la place la classe `MongoInputFormat`.

MongoDB: connecteur Hadoop

- **`setOutputFormat()` pour définir le type de format de sortie du programme. Même remarque que pour `setInputFormat` (mais avec `TextOutputFormat` la classe par défaut utilisée jusqu'ici).**

- **Toutes les fonctions précédentes ont des pendants dans l'API Hadoop. Il en existe par ailleurs d'autres (se référer à la documentation).**

- **On utilisera par ailleurs possiblement les fonctions suivantes (qui n'ont *pas* d'équivalent direct dans l'API Hadoop):**

  - **`setInputURI()` pour spécifier l'URL source des données sur MongoDB (uniquement si on souhaite lire les données depuis MongoDB).**

MongoDB: connecteur Hadoop

- **`setOutputURI()` pour spécifier l'URL destination des données de sortie sur MongoDB (uniquement si on souhaite écrire les données finale vers MongoDB).**

- **`setQuery()` pour spécifier, sous la forme d'une *string*, un document BSON décrivant les critères de sélection à appliquer sur la collection des données sources (uniquement si on souhaite lire les données sources depuis MongoDB).**

  Son format est identique à celui de la fonction `find()` de MongoDB (par exemple "`{nom: 'John'}`").

MongoDB: connecteur Hadoop

- **Les URLs MongoDB devront respecter le format suivant:**

```
mongodb://[USER:PASS@]HOST[:PORT]/DB.COLLECTION
```

Exemples:

```
mongodb://localhost/hadoop.anagrammes

mongodb://localhost:23000/hadoop.anagrammes

mongodb://bob:secret@10.0.0.1/hadoop.anagrammes
```

MongoDB: connecteur Hadoop

- **Le connecteur va par ailleurs nous permettre, si on lit/écrit *via* MongoDB plutôt que HDFS, de passer des objets contenant directement les données BSON dans les classes map et/ou reduce, à la place des types Hadoop standards comme IntWritable ou Text.**

- **Cependant, le connecteur utilise encore partiellement l'ancienne API MongoDB; et on ne pourra donc pas utiliser le type Document vu jusq'ici dans l'API.**

- **A la place, on devra utiliser le type `BSONObject` (très similaire); et une seconde classe permettant facilement de générer un objet de ce type appelée `BasicDBObjectBuilder`.**

  Toutes deux sont présentes dans la librairie Java MongoDB.

MongoDB: connecteur Hadoop

- **Pour construire un document BSON au sein d'un objet `BSONObject` avec `BasicDBObjectBuilder`, on utilisera par exemple la syntaxe suivante:**

```
BSONObject doc=BasicDBObjectBuilder.start()
                .add('prenom', 'John')
                .add('nom', 'Smith')
                …
                .get();
```

- **Comme pour la classe Document et ses méthodes `append()`, on peut imbriquer ici des documents (`BSONObject`), des tableaux, etc. et construire un BSON aussi complexe que nécessaire. Comme la classe Document, `BSONObject` propose par ailleurs une méthode `get()` permettant de récupérer la valeur d'un champs du document.**

MongoDB: connecteur Hadoop

- **Ainsi, on pourrait par exemple spécifier dans le constructeur:**

```
Configuration conf = new Configuration();
MongoConfigUtil.setOutputFormat(conf,
                                MongoOutputFormat.class);
MongoConfigUtil.setInputFormat(conf,
                                MongoInputFormat.class);
…
MongoConfigUtil.setMapperOutputKey(conf, Text.class);
MongoConfigUtil.setMapperOutputValue(conf, Text.class);
MongoConfigUtil.setOutputKey(conf, ObjectId.class);
MongoConfigUtil.setOutputValue(conf, BSONObject.class);
setConf(conf);
```

MongoDB: connecteur Hadoop

… ce qui indique qu'on souhaite lire *et* écrire via MongoDB, et que notre type de valeur finale sera un objet BSON (`BSONObject`). On indique que notre type de clef finale sera de type `ObjectId`.

- Cet aspect est important: en effet, lors de l'écriture vers MongoDB, le connecteur prendra la clef et l'attribuera au champs MongoDB "_id"; et il prendra la valeur et l'intégrera au document à la suite.

- Le fait qu'on souhaite lire depuis MongoDB implique par ailleurs la même chose, dans le sens inverse: nos couples (clef;valeur) d'entrée auront pour type de clef un `ObjectId`, et pour type de valeur un document `BSONObject` (lui-même contenant également l'`ObjectId` du document).

MongoDB: connecteur Hadoop

- **Par conséquent, la classe Mapper aura une définition du type:**

```
public class Map extends Mapper<ObjectId,
BSONObject, ..., ...>
```

- **Et la classe Reducer aura une définition du type:**

```
public class Reduce extends Reducer<..., ..., ObjectId,
BSONObject>
```

MongoDB: connecteur Hadoop

- **Il s'agit là à minima des seules modifications nécessaires pour lire/écrire depuis/vers MongoDB au sein d'un programme map/reduce Hadoop.**

- **Le connecteur possède de nombreuses autres options.**

- **Par exemple, il permet de définir diverses options relatives à la tâche map/reduce par le biais de fonctions identiques ou proches à celles qu'on applique d'ordinaire sur l'objet Job d'une tâche map/reduce.**

- **Pour plus de détails, se référer à la documentation.**

MongoDB: connecteur Hadoop - Exemple

- **Le programme de détection des anagrammes, version lecture/écriture MongoDB. Classe driver:**

```
package org.mbds.hadoop.mongo;

import org.apache.hadoop.conf.Configuration;
...
import org.bson.BSONObject;

public class MongoAnagrammes extends MongoTool
{
 public MongoAnagrammes()
 {
  Configuration conf = new Configuration();

  MongoConfigUtil.setOutputFormat(conf, MongoOutputFormat.class);
```

MongoDB: connecteur Hadoop - Exemple


```
 MongoConfigUtil.setInputFormat(conf, MongoInputFormat.class);
 MongoConfigUtil.setInputURI(conf,
                    "mongodb://localhost/hadoop.commonwords" );
 MongoConfigUtil.setQuery(conf, "{}");
 MongoConfigUtil.setOutputURI(conf,
                     "mongodb://localhost/hadoop.anagrammes" );
 MongoConfigUtil.setMapper(conf, MongoAnagrammesMap.class);
 MongoConfigUtil.setReducer(conf, MongoAnagrammesReduce.class);
 MongoConfigUtil.setMapperOutputKey(conf, Text.class);
 MongoConfigUtil.setMapperOutputValue(conf, Text.class);
 MongoConfigUtil.setOutputKey(conf, ObjectId.class);
 MongoConfigUtil.setOutputValue(conf, BSONObject.class);

 setConf(conf);
}
```

MongoDB: connecteur Hadoop - Exemple


```
 public static void main(String[] args) throws Exception
 {
  System.exit(ToolRunner.run(new MongoAnagrammes(), args));
 }
}
```

MongoDB: connecteur Hadoop - Exemple

- **Classe map:**

```
package org.mbds.hadoop.mongo;
import org.bson.BSONObject;
…
public class MongoAnagrammesMap
        extends Mapper<Object, BSONObject, Text, Text>
{
  protected void map(Object key, BSONObject value,
       Context context) throws IOException, InterruptedException
  {
    String val=(String)value.get("word");
    char[] letters=val.toLowerCase().toCharArray();
    Arrays.sort(letters);
    context.write(new Text(new String(letters)), new Text(val));
  }
}
```

MongoDB: connecteur Hadoop - Exemple

- **Classe reduce:**

```
package emsimbds;
import org.apache.hadoop.io.Text;
...

public class MongoAnagrammesReduce extends
                Reducer<Text, Text, ObjectId, BSONObject>
{
  public void reduce(Text key, Iterable<Text> values,
       Context context) throws IOException, InterruptedException
  {
    Iterator<Text> i=values.iterator();
    ArrayList<String> anaglist=new ArrayList<String>();
    while(i.hasNext())   // Pour chaque valeur...
      anaglist.add(i.next().toString());
```

MongoDB: connecteur Hadoop - Exemple

11-27

```
    String[] anag=anaglist.toArray(new String[anaglist.size()]);
    BSONObject doc=BasicDBObjectBuilder.start()
                   .add("letters", key.toString())
                   .add("anagrammes", anag).get();
    context.write(new ObjectId(), doc);
  }
}
```

MongoDB: connecteur Hadoop - Exemple

- **Si, dans l'exemple précédent, la collection hadoop.commonwords contient par exemple les documents suivants:**

```
{ "_id" : ObjectId("5657e09...63fa7a4"), "word" : "acrimonious" }
{ "_id" : ObjectId("5657e09...63fa7a5"), "word" : "aerodyne" }
{ "_id" : ObjectId("5657e09...63fa7a6"), "word" : "acariasis" }
{ "_id" : ObjectId("5657e09...63fa7a7"), "word" : "actually" }
{ "_id" : ObjectId("5657e09...63fa7a8"), "word" : "abomasum" }
{ "_id" : ObjectId("5657e09...63fa7a9"), "word" : "achene" }
{ "_id" : ObjectId("5657e09...63fa7aa"), "word" : "abscess" }
{ "_id" : ObjectId("5657e09...63fa7ab"), "word" : "adorn" }
{ "_id" : ObjectId("5657e09...63fa7ac"), "word" : "acentric" }
{ "_id" : ObjectId("5657e09...63fa7ad"), "word" : "absentee" }
{ "_id" : ObjectId("5657e09...63fa7ae"), "word" : "afterheat" }
...
```

MongoDB: connecteur Hadoop - Exemple

- **Alors à l'issue de l'exécution, une nouvelle collection hadoop.anagrammes sera créée qui contiendra:**

```
{ "_id" : ObjectId("56...8d"), "letters" : "aaabcehinrt",
  "anagrammes" : [ "abranchiate" ] }
{ "_id" : ObjectId("56...8e"), "letters" : "aaabdelpt",
  "anagrammes" : [ "adaptable" ] }
{ "_id" : ObjectId("56...69"), "letters" : "elmno",
  "anagrammes" : [ "melon", "lemon" ] }
{ "_id" : ObjectId("56...8f"), "letters" : "aaabdfr",
  "anagrammes" : [ "abfarad" ] }
{ "_id" : ObjectId("56...90"), "letters" : "aaabdn",
  "anagrammes" : [ "abadan" ] }
{ "_id" : ObjectId("56...53"), "letters" : "aekl",
  "anagrammes" : [ "leak", "lake" ] }
...
```

MongoDB: connecteur Hadoop - Exécution

- **Pour exécuter le programme, on aura besoin de spécifier à Hadoop les deux librairies à charger en plus pour que les classes relatives au connecteur soient chargées correctement:**
  - **La librairie du connecteur (mongo-hadoop-core.jar).**
  - **Sa dépendance, l'API mongodb (mongo-java-driver.jar).**
- **On spécifiera l'emplacement de ces librairies par le biais de l'option additionnelle -libjars de la commande « hadoop jar », en les séparant par des virgules.**
- **Ces librairies doivent non seulement exister sur le système de fichier local mais également sur HDFS, au même emplacement; cela permet à tous les nœuds du *cluster* d'y avoir accès.**

MongoDB: connecteur Hadoop - Exécution

- **Ainsi, on exécutera par exemple:**

```
hadoop jar programme.jar org.mbds.Programme
    -libjars /opt/mongo-hadoop-core.jar,/opt/mongo-java-
driver.jar
```

- **Selon la version de Hadoop, il peut également être nécessaire de définir une variable d'environnement HADOOP_CLASSPATH sur la machine locale qui pointe également sur ces deux librairies, mais cette fois séparée par un symbole « : » (comme un classpath Java classique).**