

Module

INGÉNIERIE BIG DATA NoSQL

Présenté par:

***Fatma
Abdelhedi***

 fatma.abdelhedi@trimane.fr

***Bilel
SDIRI***

 bilel.sdiri@trimane.fr



TRIMANE

THE DATA INTELLIGENCE
COMPANY

A PART OF THE BLOCKCHAIN GROUP

102 TERRASSE BOIELDIEU , TOUR W
92 800 PUTEAUX

www.trimane.fr

Plan

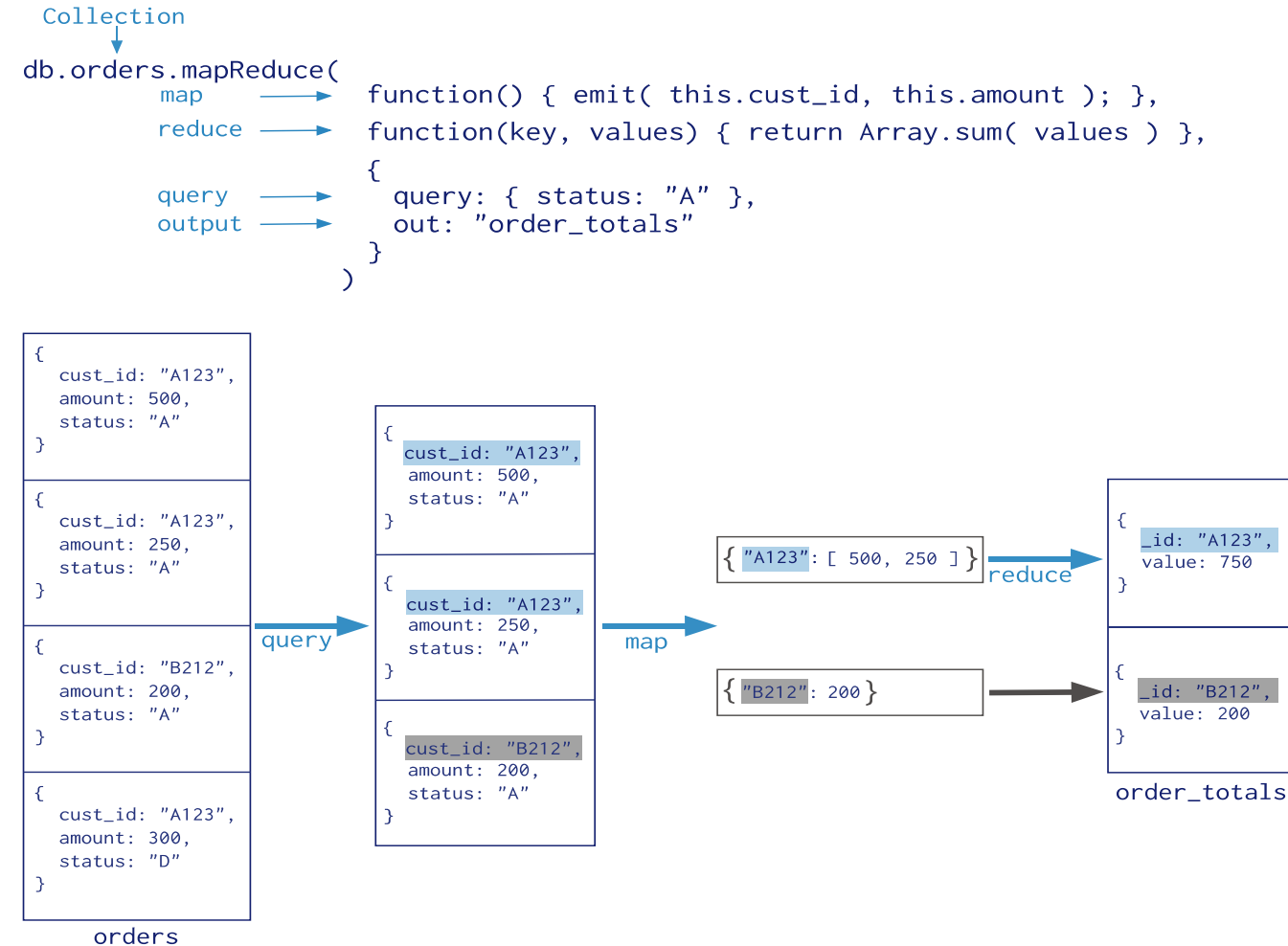
1. Introduction aux Big Data
2. Les bases de données NoSQL
3. La solution MongoDB
4. Modélisation des données Big Data
5. Manipulation des données avec MongoDB (CRUD)
6. Agrégation des données
 - 6.1. Map-Reduce avec MongoDB
 - 6.2. Framework d'agrégation (FA)
 - 6.3. Map-Reduce VS Framework d'agrégation
6. Jointures et références
7. Recherche d'information

- Map-Reduce est un paradigme de traitement de données permettant de synthétiser de gros volumes de données en résultats agrégés.
- Pour effectuer des opérations MapReduce, MongoDB fournit la commande **MapReduce**.
- Toutes les opérations Map-Reduce de MongoDB utilisent des fonctions Javascript pour associer ou faire un mapping des valeurs à une clé.
 - **NB** : Si plusieurs valeurs sont associées à une clé, elles seront encapsulées dans un seul objet.
- L'exécution des fonction Javascripts se déroule dans le processus **mongod**.
- Map-Reduce peut retourner le résultat soit online ou le stocker dans une collection.

6. Agrégation des données

6.1. Map-Reduce avec MongoDB : workflow

1. Filtrer les documents en fonction des critères de la requête : **query**
2. Appliquer la phase **map** sur chaque document filtré par l'étape 1
 ➔ **Output map** : paires clés –valeur.
3. Appliquer la phase **reduce** sur les clés auxquelles plusieurs valeurs sont associées
 ➔ **Output reduce** : données synthétisées et agrégées.
4. Option: appliquer une phase **finalize** pour effectuer des traitements/calculs supplémentaires sur les données agrégées de l'étape 3.
5. Retourner le résultat online ou le stocker dans une collection.



Exemple : Calculer le montant total à payer pour chaque client en utilisant Map-Reduce de MongoDB.

1. Définir la fonction **map** qui traite chaque document en entré (input)

- **this** : le document en cours de traitement par l'opération Map-Reduce
- La fonction associe le prix (price) à l'identifiant du client (cust_id) **pour chaque document**, et retourne le couple (id_cust, price)

```
var mapFunction1 = function() {  
    emit(this.cust_id, this.price);  
};
```

Exemple : Calculer le montant total à payer pour chaque client en utilisant Map-Reduce de MongoDB.

2. Définir la fonction **reduce** avec deux arguments :

- La variable **ValuesPrices** est un Array dont les éléments sont les prix retournés par la fonction **map** et groupé en fonction de l'identifiant du client **keyCustId**
- La fonction « réduit » l'Array **ValuesPrices** à la somme de ses éléments.

```
var reduceFunction1 = function(keyCustId, valuesPrices) {  
    return Array.sum(valuesPrices);  
};
```

Exemple : Calculer le montant total à payer pour chaque client en utilisant Map-Reduce de MongoDB.

3. Effectuer l'opération Map-Reduce

- Effectuer l'opération Map-Reduce sur tous les documents de la collection « orders » en utilisant la fonction de map `mapFunction1` et la fonction reduce `reduceFunction1` définies précédemment.
- Le résultat sera stocké dans une collection intitulée « map_reduce_example ». Si la collection existe déjà, son contenu sera écrasé et remplacé par le résultat de cette nouvelle opération Map-Reduce.

```
db.orders.mapReduce(  
  mapFunction1,  
  reduceFunction1,  
  { out: "map_reduce_example" }  
)
```



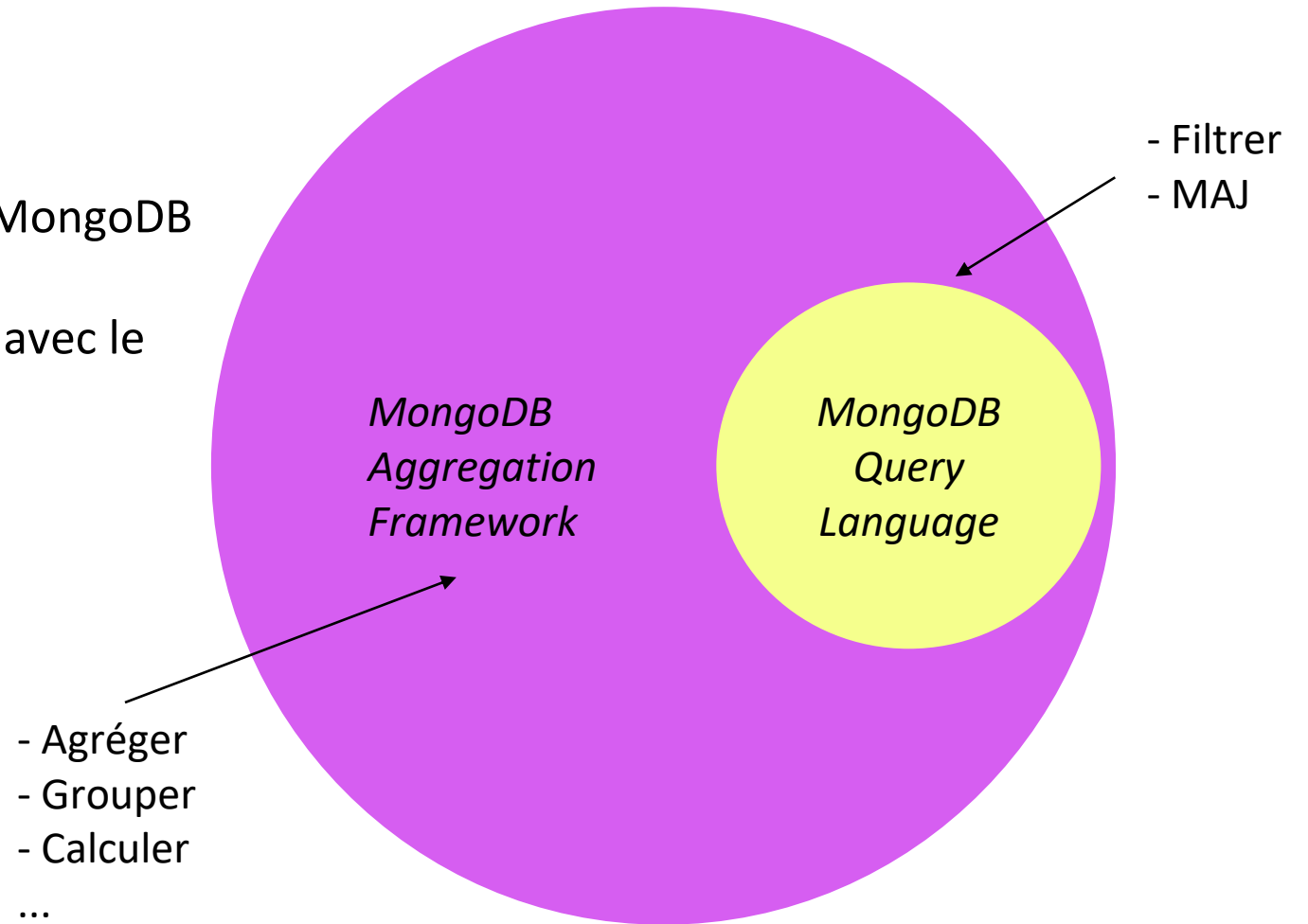
<Exemple cours shell>

Plan

1. Introduction aux Big Data
2. Les bases de données NoSQL
3. La solution MongoDB
4. Modélisation des données Big Data
5. Manipulation des données avec MongoDB (CRUD)
6. Agrégation des données
 - 6.1. Map-Reduce avec MongoDB
 - 6.2. Framework d'agrégation (FA)
 - 6.3. Map-Reduce VS Framework d'agrégation
6. Jointures et références
7. Recherche d'information

- Une autre façon pour interroger les données MongoDB
- Tout ce qu'on fait avec MQL est faisable aussi avec le Framework d'agrégation.

⚠ L'inverse n'est pas nécessairement vrai

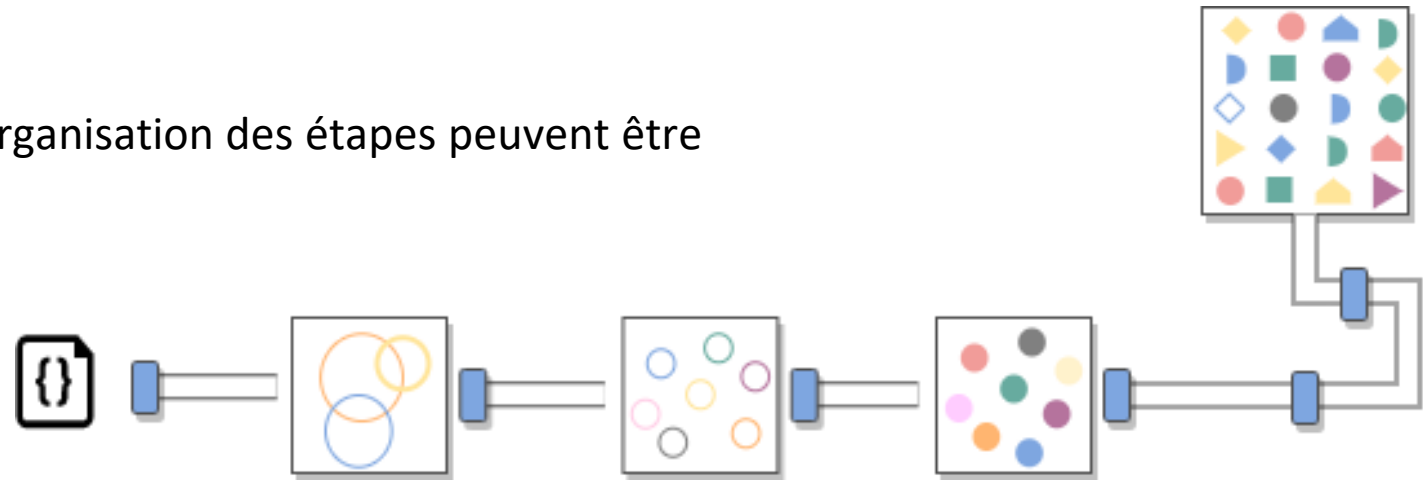
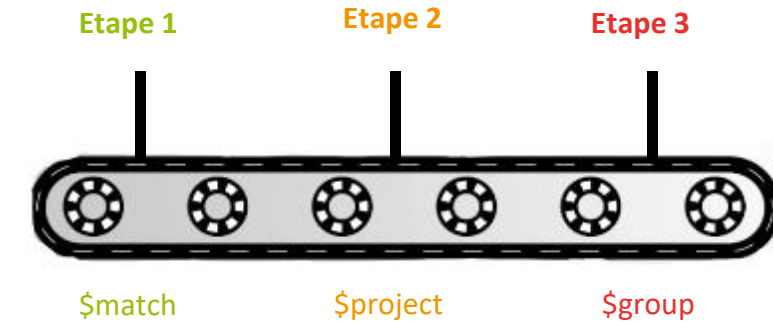


6. Agrégation des données

6.2. Framework d'agrégation (FA)

6.2.1. Rappel : Pipeline

- Limite de MongoDB : analyse analytique
- Solution dans MongoDB : moteur d'agrégation basé sur la notion de Pipeline.
- Les pipelines sont composés par plusieurs étapes configurables pour filtrer les données et effectuer les transformations souhaitées.
- Les documents traversent les étapes comme les pièces d'une chaîne d'assemblage.
- A quelques exceptions près, le nombre et l'organisation des étapes peuvent être décidés en fonction du besoin.



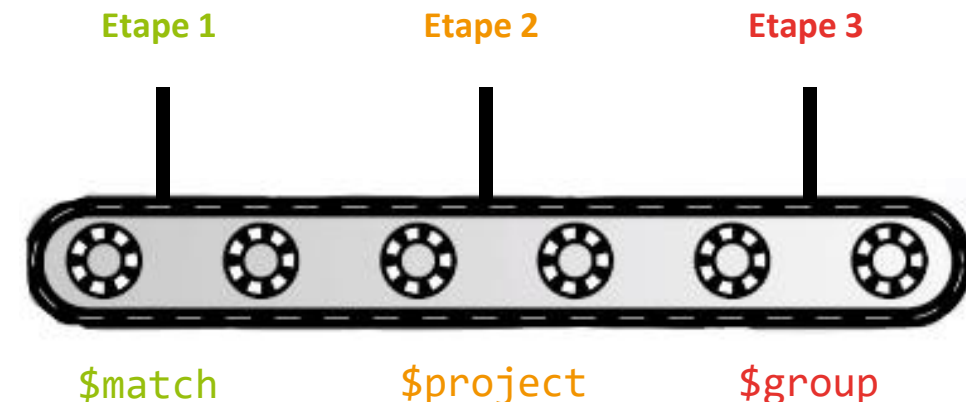
```
> db.collection.aggregate( [{Etape 1}, {Etape 2},...{Etape N}], {options})
```

- Les pipelines sont toujours présentés comme étant une **liste** d'étapes successives, d'où **l'importance de l'ordre**.
- Une étape est composée par un ou plusieurs opérateurs d'agrégation, ou expressions.
- Une expression peut prendre comme paramètre un seul argument ou une liste d'arguments.
- Certaines expressions sont exclusives à des étapes spécifiques.*

NB

Syntaxe MQL : { <champ> : { <opérateur> : valeur } }
Syntaxe agrégation : { <opérateur> : { <champ>, <valeur> } }

* Ne peuvent pas être utilisées dans d'autres étapes



```
> db.collection.aggregate([  
  {"$match" : {}} ]  
)
```

- Il est recommandé d'utiliser **\$match** au début du pipeline.
- **\$match** utilise la même syntaxe d'interrogation que la méthode find().
- On ne peut pas utiliser \$where avec **\$match**.
- Si l'étape **\$match** contient l'opérateur d'interrogation \$text, elle doit obligatoirement être placée comme première étape du pipeline.
- Afin de profiter de la rapidité des index, il faut placer \$match en première étape, ce qui permet d'accélérer l'exécution des requêtes.

❑ Requête :

Trouver les logements qui proposent le wifi comme service.

➤ MQL

```
> db.airbnb.find({ "amenities": "Wifi" })
```

➤ Framework d'agrégation

```
> db.airbnb.aggregate([  
    { "$match": { "amenities": "Wifi" } },  
])
```

```
> db.collection.aggregate([  
  {"$project" : { <specifications> } }  
])
```

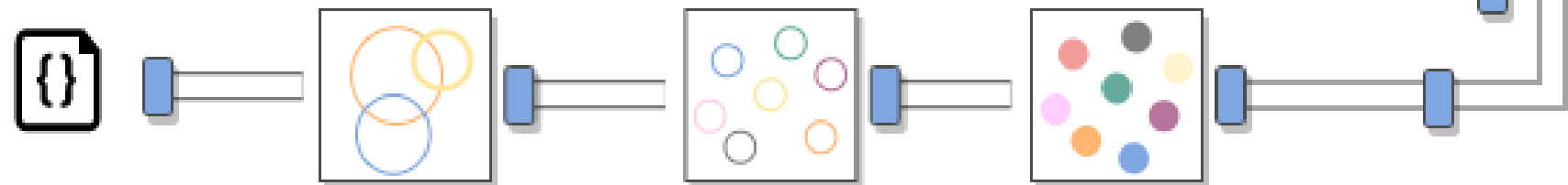
- **Différente** de la fonctionnalité de projection de la méthode find()
- Permet de :
 - ✓ Sélectionner des champs à retenir/enlever.
 - ✓ Réaffecter/initialiser la valeur d'un champs existant.
 - ✓ Créer de nouveaux champs.
- Projeter un champ (par le **binaire 1** ou **par un calcul**) entraîne l'**obligation** de spécifier **tous** les autres champs à retenir, à l'exception du champs `_id`.
- \$project peut être utilisée autant de fois que nécessaire dans un pipeline d'agrégation.

```
> db.airbnb.aggregate([  
  { "$match": { "amenities": "Wifi" } },  
  { "$project": { "price": 1, "address": 1, "_id": 0 } }  
])
```

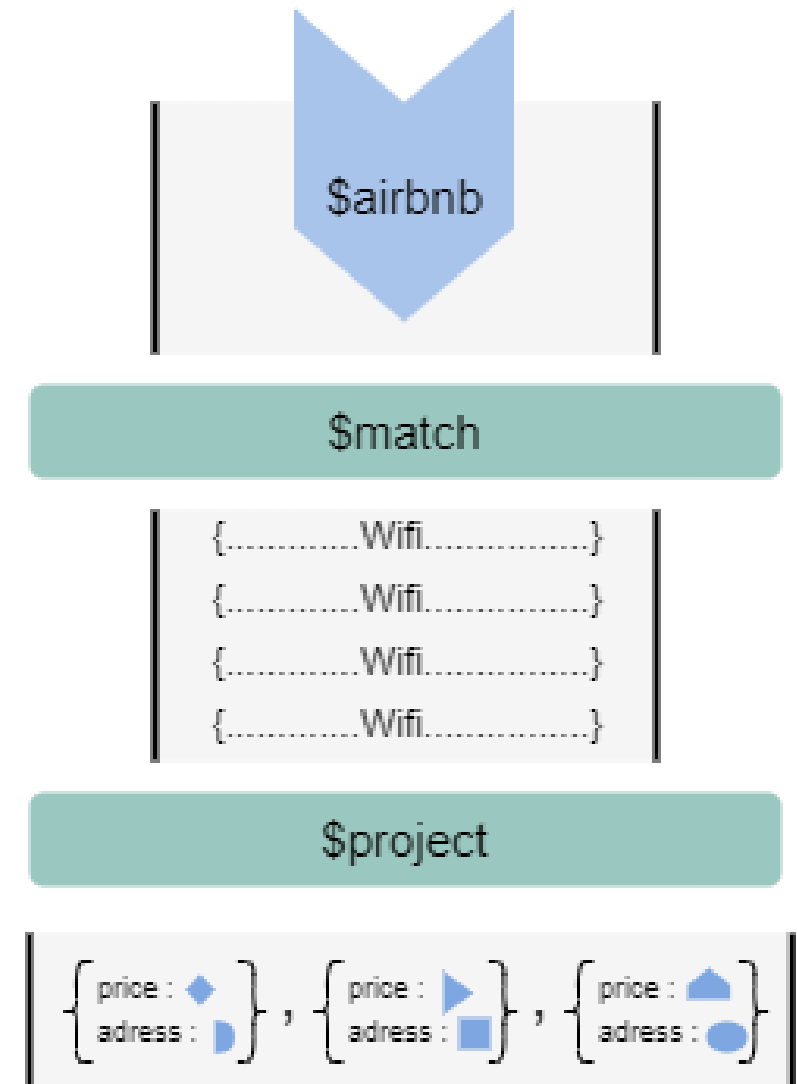


Utilisation du format liste ordonnée pour agréger les données. L'ordre des éléments est important dans une liste.

- ➔ Le Framework d'agrégation fonctionne comme un pipeline typique où l'ordre des actions est important pour établir le résultat final.
- ➔ Chaque action est exécutée en fonction de son ordre d'apparition dans liste d'agrégation (dans notre exemple: matcher d'abord puis projeter).



```
> db.airbnb.aggregate([  
  {"$match": { "amenities": "Wifi" } },  
  { "$project": { "price": 1, "address": 1, "_id": 0}}  
])
```



- Les accumulateurs d'expression disponibles sont :
 - ✓ **\$sum** : Calculer la somme
 - ✓ **\$avg** : Calculer la moyenne
 - ✓ **\$max** : Retourner le Maximum
 - ✓ **\$min** : Retourner le Minimum
 - ✓ **\$stdDevPop** , **\$stdDevSam** : Calculer l'écart-type pour resp. toute la population ou pour un sous ensemble de la population

➔ NB: Les accumulateurs d'expressions, utilisés dans l'étape \$project opèrent **uniquement** sur un **Array du document projeté**, sans effectuer des **calculs inter-documents** (mémoire intra-document uniquement)

```
> db.exemple.find()

{"_id" : 0 , data : [ 1, 2, 3, 4, 5 ] }
{"_id" : 1 , data : [ 1, 3, 5, 7, 9 ] }
{"_id" : 2 , data : [ 2, 4, 6, 8, 10 ] }
```



<Exemple cours shell>

```
> db.exemple.aggregate ([
    { "$project" : { "moyenne" : { "$avg" : "$data" } } }
])

{"_id" : 0 , moyenne : 3 }
{"_id" : 1 , moyenne : 5 }
{"_id" : 2 , moyenne : 6 }
```

```
> db.collection.aggregate([
  {"$addFields" : { <nouveau champs> : <expression de transformation> } }
])
```

1.1. Etape :

- **But** : Ajouter un ou plusieurs champs aux documents retournés par curseur.
- Une transformation dans un \$project est interprétée comme étant une projection : retenir les champs transformés et supprimer les autres.

➡ Si on souhaite garder d'autres champs quand même, nous devons **explicitement** les projeter avec le **Booléen 1**.

- ⊖ Tâche fastidieuse **si nous avons beaucoup de champs à garder** (non transformé) car il faut les projeter un par un.
- ⊖ Déconcentrer l'attention sur les éventuelles transformation à faire.

```
> db.collection.aggregate([  
  {"$addField" : { <nouveau champs> : <expression de transformation> } }  
])
```

- ➡ \$addField ajoute les champs transformés sans influencer la projection (i.e. sans supprimer aucun champ) et sans avoir à spécifier explicitement les champs à garder un par un(Exemple)
- ➡ Combiner l'utilisation de \$project et \$addField est utile quand on a des pipelines complexes avec beaucoup de calculs et transformations

Exemple de document type de la collection : solarSystem

```
{
  "_id" : ObjectId("59a06674c8df9f3cd2ee7d54"),
  "name" : "Earth",
  "type" : "Terrestrial planet",
  "orderFromSun" : 3,
  "radius" : {"value" : 6378.137, "units" : "km"},
  "mass" : {"value" : 5.9723e+24, "units" : "kg"},
  "sma" : {"value" : 149600000, "units" : "km"},
  "orbitalPeriod" : {"value" : 1, "units" : "years"},
  "eccentricity" : 0.0167,
  "meanOrbitalVelocity" : {"value" : 29.78, "units" : "km/sec"},
  "rotationPeriod" : {"value" : 1, "units" : "days"},
  "inclinationOfAxis" : {"value" : 23.45, "units" : "degrees"},
  "meanTemperature" : 15,
  "gravity" : {"value" : 9.8, "units" : "m/s^2"},
  "escapeVelocity" : {"value" : 11.18, "units" : "km/sec"},
  "meanDensity" : 5.52,
  "atmosphericComposition" : "N2+O2",
  "numberOfMoons" : 1,
  "hasRings" : false,
  "hasMagneticField" : true
}
```

➤ Initialiser le champs « gravity » à la valeur de la gravité

```
db.solarSystem.aggregate ([
  {"$project" : {"gravity" : "$gravity.value"}}
])
```

```
{ "_id" : ObjectId("59a06674c8df9f3cd2ee7d54"), "gravity" : 9.8 }
{ "_id" : ObjectId("59a06674c8df9f3cd2ee7d59"), "gravity" : 11.15 }
{ "_id" : ObjectId("59a06674c8df9f3cd2ee7d58"), "gravity" : 8.87 }
{ "_id" : ObjectId("59a06674c8df9f3cd2ee7d57"), "gravity" : 10.44 }
{ "_id" : ObjectId("59a06674c8df9f3cd2ee7d56"), "gravity" : 24.79 }
{ "_id" : ObjectId("59a06674c8df9f3cd2ee7d53"), "gravity" : 8.87 }
{ "_id" : ObjectId("59a06674c8df9f3cd2ee7d52"), "gravity" : 3.24 }
{ "_id" : ObjectId("59a06674c8df9f3cd2ee7d51"), "gravity" : 274 }
{ "_id" : ObjectId("59a06674c8df9f3cd2ee7d55"), "gravity" : 3.71 }
```

Exemple de document type de la collection : solarSystem

```
{
  "_id" : ObjectId("59a06674c8df9f3cd2ee7d54"),
  "name" : "Earth",
  "type" : "Terrestrial planet",
  "orderFromSun" : 3,
  "radius" : {"value" : 6378.137, "units" : "km"},
  "mass" : {"value" : 5.9723e+24, "units" : "kg"},
  "sma" : {"value" : 149600000, "units" : "km"},
  "orbitalPeriod" : {"value" : 1, "units" : "years"},
  "eccentricity" : 0.0167,
  "meanOrbitalVelocity" : {"value" : 29.78, "units" : "km/sec"},
  "rotationPeriod" : {"value" : 1, "units" : "days"},
  "inclinationOfAxis" : { "value" : 23.45, "units" : "degrees"},
  "meanTemperature" : 15,
  "gravity" : {"value" : 9.8, "units" : "m/s^2"},
  "escapeVelocity" : {"value" : 11.18, "units" : "km/sec"},
  "meanDensity" : 5.52,
  "atmosphericComposition" : "N2+O2",
  "numberOfMoons" : 1,
  "hasRings" : false,
  "hasMagneticField" : true
}
```

- Faire la **même transformation** pour les champs : mass, radius, et sma et **garder l'affichage** des champs name, meanTemperature, et meanDensity ?

```
db.solarSystem.aggregate ([
  { "$project" : {
    "_id" : 0,
    "name": 1,
    "gravity" : "$gravity.value"
    "meanTemperature" : 1,
    "meanDensity": 1,
    "mass" : "$mass.value",
    "radius" : "$radius.value",
    "sma" : "$sma.value"
  }
}]
```

NB: nous avons **explicitement** projeter **tous** les champs non transformés **un par un** (avec Booléen 1).

➡ Tâche fastidieuse

- Utiliser \$addFields pour les champ à transformer : isoler l'opération de transformation.

```
db.solarSystem.aggregate ([  
  
  {"$addFields" : {  
    "gravity" : "$gravity.value"  
    "mass" : "$mass.value",  
    "radius" : "$radius.value",  
    "sma" : "$sma.value"  
  }  
}]
```

Grâce à \$addField, nous avons effectué toutes les transformations nécessaires tout en gardant tous les champs affichés



Pas besoin de spécifier les champs à garder un par un.

```
{  
  "_id" : ObjectId("59a06674c8df9f3cd2ee7d54"),  
  "name" : "Earth",  
  "type" : "Terrestrial planet",  
  "orderFromSun" : 3,  
  "radius" : 6378.137,  
  "mass" : 5.9723e+24,  
  "sma" : 149600000,  
  "gravity" : 9.8,  
  "orbitalPeriod" : {"value" : 1, "units" : "years"},  
  "eccentricity" : 0.0167,  
  "meanOrbitalVelocity" : {"value" : 29.78,"units" : "km/sec"},  
  "rotationPeriod" : { "value" : 1,"units" : "days"},  
  "inclinationOfAxis" : {"value" : 23.45,"units" : "degrees"},  
  "meanTemperature" : 15,  
  "escapeVelocity" : { "value" : 11.18,"units" : "km/sec"},  
  "meanDensity" : 5.52,  
  "atmosphericComposition" : "N2+O2",  
  "numberOfMoons" : 1,  
  "hasRings" : false,  
  "hasMagneticField" : true  
}
```

6. Agrégation des données

6.2. Framework d'agrégation (FA)

6.2.5. Etape \$addField : Exemple

- Combiner l'utilisation de \$project et \$addField

```
db.solarSystem.aggregate ([  
  {"$project" : {  
    "_id" : 0,  
    "name": 1,  
    "gravity" : 1,  
    "mass" : 1,  
    "radius" : 1  
  }  
} // end $project  
,  
{ "$addField" : {  
  "gravity" : "$gravity.value",  
  "mass" : "$mass.value",  
  "radius" : "$radius.value"  
}  
} // end $addField  
]).pretty()
```

- ✓ Détacher la projection de la transformation des données

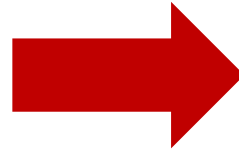
➡ Pas besoin de spécifier les champs à garder un par un.

```
{  
  "name" : "Earth",  
  "radius" : 6378.137,  
  "mass" : 5.9723e+24,  
  "gravity" : 9.8  
}  
{  
  "name" : "Neptune",  
  "radius" : 24765,  
  "mass" : 1.02413e+26,  
  "gravity" : 11.15  
}  
{  
  "name" : "Uranus",  
  "radius" : 25559,  
  "mass" : 8.6813e+25,  
  "gravity" : 8.87  
}
```

MQL

Méthodes des curseurs :

- ✓ `pretty()`
- ✓ `count()`
- ✓ `sort()`
- ✓ `limit()`
- ✓ `Skip()`



FA

Etapes :

```
{"$count" : { <Nom du champ>* }}  
{"$sort" : { <champ> : <sens du tri> }}  
{"$limit" : <integer> }  
{"$skip" : <integer>}
```

**L'intitulé du champ qui va afficher le résultat du count*

❑ Gestion mémoire liée à l'étape de tri

- *Si le tri est effectué au début du pipeline (première étape), les index seront utilisés.*
- ***Sinon** l'étape effectue un tri « in-memory », ce qui augmente significativement la consommation mémoire du serveur.*
- *L'opération de trie dans un pipeline d'agrégation est limitée à **100 megabytes de mémoire RAM par défaut.***
- *Pour trier et traiter des BD volumineuses, il est possible d'autoriser l'utilisation de la mémoire disque dur en activant une option de la méthode aggregate().*
- *Tout ce qui excède les 100 M de mémoire par défaut, sera traité automatiquement par le pipeline sur le disque.*

```
db.solarSystem.aggregate([
  { "$project":
    {
      "_id" : 0,
      "name" : 1,
      "hasMagneticField": 1,
      "numberOfMoons": 1
    }
  },
  { "$sort" : { "hasMagneticField": -1, "numberOfMoons" : -1} },],
  { "allowDiskUse" : true }
)
```

```
{ "name" : "Jupiter", "numberOfMoons" : 67, "hasMagneticField" : true }
{ "name" : "Saturn", "numberOfMoons" : 62, "hasMagneticField" : true }
{ "name" : "Uranus", "numberOfMoons" : 27, "hasMagneticField" : true }
{ "name" : "Neptune", "numberOfMoons" : 14, "hasMagneticField" : true }
{ "name" : "Earth", "numberOfMoons" : 1, "hasMagneticField" : true }
{ "name" : "Mercury", "numberOfMoons" : 0, "hasMagneticField" : true }
{ "name" : "Sun", "numberOfMoons" : 0, "hasMagneticField" : true }
{ "name" : "Mars", "numberOfMoons" : 2, "hasMagneticField" : false }
{ "name" : "Venus", "numberOfMoons" : 0, "hasMagneticField" : false }
```

```
{ $sample : { size : <(N) Nombre documents retournés> } }
```

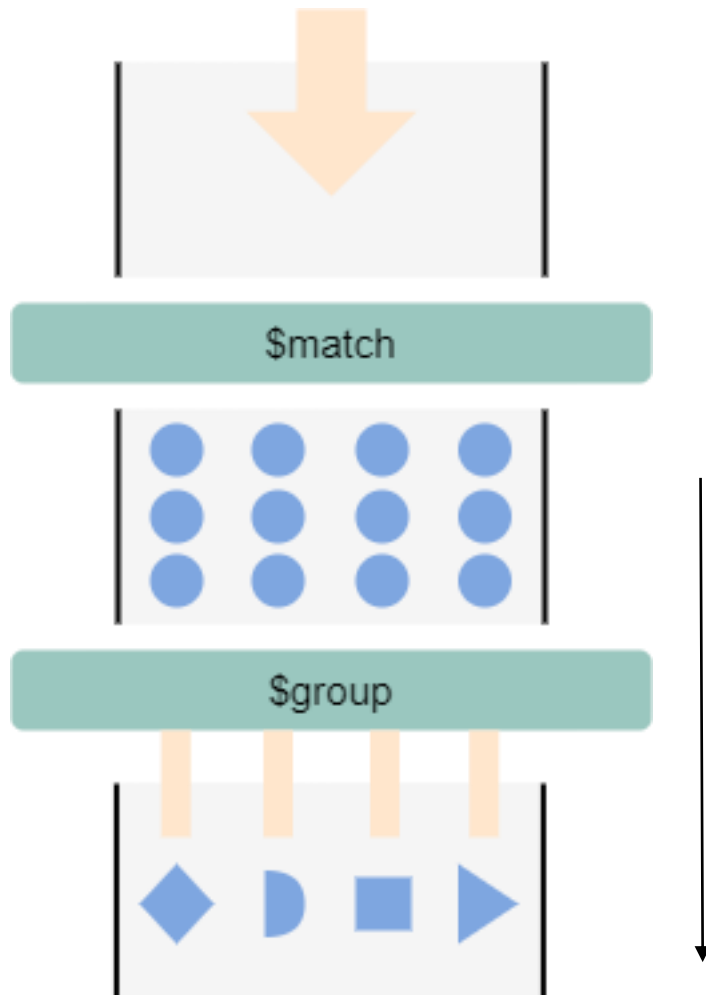
- *Sélectionner aléatoirement un sous ensemble de documents à partir d'une collection.*
- *2 façons pour sélectionner les documents :*
 - *SI* ($N \leq 5\%$ du nombre total des documents dans la collection source) **ET** (Collection contient plus de 100 documents) **ET** (\$sample est la première étape)
 - ➡ *Un pseudo-curseur sélectionne aléatoirement les documents à retourner en fonction du nombre spécifié en paramètre.*
 - *SINON*
 - ➡ *Un tri in-memory est effectué (critère aléatoire) puis retourner les **N** documents.*
NB: les mêmes restrictions mémoires de l'étape de tri {\$sort} s'appliquent dans ce cas.

```
{ $sample : { size : <(N) Nombre documents retournés> } }
```

□ **Utilité**

- *Travailler avec des **collections volumineuses** et nous souhaitons :*
 - *Traiter un nombre limité de documents,*
 - *Effectuer des analyses préliminaires sur un sous ensemble aléatoire de la collection,*
 - *Récupérer des documents de manière aléatoire pour des besoins telles que la recherche aléatoire d'utilisateurs dans une collection (tirage au sort),*
 - *Effectuer un sous-échantillonnage de la collection avec des documents choisis aléatoirement.*
 - *Avoir une idée sur la structure dynamique des documents d'une collection.*

- ❑ Grouper les données en fonction d'un critère pour créer des ensembles différents



```
{ $group :  
  {  
    _id: <expression> // Expression du regroupement  
    <field1> : { <accumulator1> : <expression1> },  
    ...  
  }  
}
```

Partie pour effectuer des analyses quantitatives

Une fois le filtrage est effectué, les étapes qui suivent ne modifient pas les données, mais travaillent avec le curseur contenant les données déjà filtrés dans les étapes précédentes du pipeline.

- Tous les documents ayant la même valeur du champ de groupement sont groupés ensemble.
- Chaque valeur unique du champ de groupement **produit un document** qui montre la valeur commune à l'ensemble de documents groupés.

➤ *Exemple: grouper les logements Airbnb par pays ?*

```
db.listingsAndReviews.aggregate([  
  {"$group" : {"_id" : "$address.country"}}  
])
```

```
> db.listingsAndReviews.aggregate([ {"$group" : {"_id" : "$address.country"}} ])  
{ "_id" : "Portugal" }  
{ "_id" : "Spain" }  
{ "_id" : "United States" }  
{ "_id" : "Brazil" }  
{ "_id" : "Turkey" }  
{ "_id" : "Australia" }  
{ "_id" : "Hong Kong" }  
{ "_id" : "China" }  
{ "_id" : "Canada" }
```

➡ **NB** : Grouper avec un seul critère revient aussi à utiliser la commande **distinct**

➡ Explorons la fonctionnalité puissante du Framework d'agrégation permettant de cumuler plusieurs expressions !

Grouper par pays et afficher le nombre d'offres de location par pays.

```
db.listingsAndReviews.aggregate ([  
  
  { "$group" : {  
    "_id" : "$address.country",  
    "Number_offers" : {"$sum" : 1}  
  }  
  
})
```

```
{ "_id" : "Turkey", "Number_offers" : 661 }  
{ "_id" : "China", "Number_offers" : 19 }  
{ "_id" : "Canada", "Number_offers" : 649 }  
{ "_id" : "Brazil", "Number_offers" : 606 }  
{ "_id" : "Australia", "Number_offers" : 610 }  
{ "_id" : "United States", "Number_offers" : 1222 }  
{ "_id" : "Spain", "Number_offers" : 633 }  
{ "_id" : "Hong Kong", "Number_offers" : 600 }  
{ "_id" : "Portugal", "Number_offers" : 555 }
```

❏ Remarques :

- Les cumulateur d'expression ignorent tout document dont la valeur du champ spécifié :
 - N'est pas conforme au type de données que l'expression s'attend à trouver.
 - N'existe pas (null ou champ n'existe pas).
- Si tous les documents traités/filtrés contiennent un type de données incorrect ou une valeur manquante pour le champ spécifié, l'expression retourne **null**.
 - ➡ Il est important de vérifier les données et les nettoyer avant de les injecter dans un pipeline d'agrégation ou dans un accumulateur d'expressions.
- L'étape \$group peut être utilisée plusieurs fois dans le même pipeline.

- Décompose et duplique un document en fonction des éléments d'un Array attribué comme valeur à l'un de ses champs.

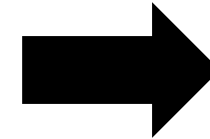


Quelle est l'utilité d'une telle transformation à votre avis ?

```
{
  "image_title" : "Summer 1988",
  "tags" : ["Travel" , "Holidays"]
}

{
  "image_title" : "Summer 2000",
  "tags" : ["Holidays" , "Travel"]
}

{
  "image_title" : "christmas 1992",
  "tags" : ["xmas" , "family" , "santa"]
}
```



```
{
  "image_title" : "Summer 1988",
  "tags" : "Travel"
}

{
  "image_title" : "Summer 1988",
  "tags" : "Holidays"
}

{
  "image_title" : "Summer 2000",
  "tags" : "Holidays"
}

{
  "image_title" : "Summer 2000",
  "tags" : "Travel"
}

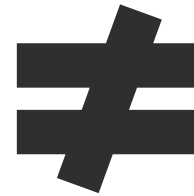
{
  "image_title" : "christmas 1992",
  "tags" : "xmas"
}

{
  "image_title" : "christmas 1992",
  "tags" : "family"
}

{
  "image_title" : "christmas 1992",
  "tags" : "santa"
}
```

❑ Exemple typique d'utilisation:

```
{  
  "image_title" : "Summer 1988",  
  "tags" : [  
    "Holidays",  
    "Travel"  
  ]  
}
```




```
{  
  "image_title" : "Summer 1988",  
  "tags" : [  
    "Travel",  
    "Holidays"  
  ]  
}
```

L'Array à gauche n'aura pas un match avec le tableau à droite même s'ils renferment les même éléments.



En faisant un groupement (\$group), les tableaux sont matchés en fonction d'une égalité stricte, pas en termes d'une équivalence.

❑ A retenir

- \$unwind opère uniquement sur les valeur d'un Array.
 - Il existe deux façons pour faire \$unwind, nous avons vu la syntaxe courte et la plus simple.
 - Utiliser \$unwind avec des collections volumineuses et avec des documents riches renfermant des grands Arrays peut engendrer des problèmes de performance.
 - Dupliquer les documents peut causer le dépassement de la limite de l'espace mémoire par défaut du Framework d'agrégation
- 

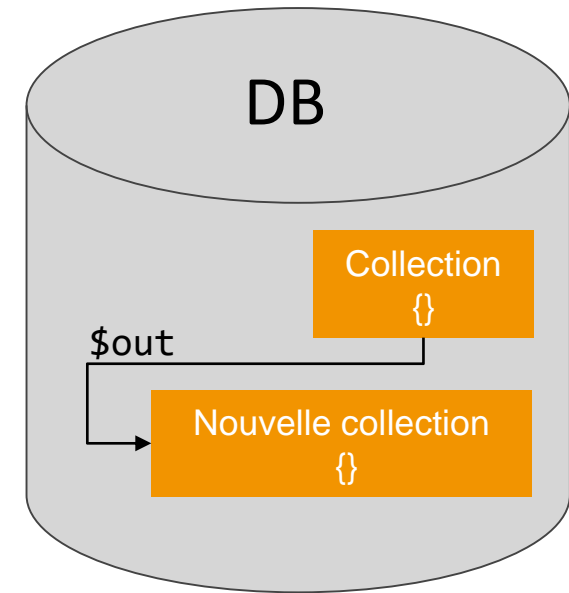
Solution ?

 - ✓ Effectuer une étape de \$match en préliminaire peut réduire les données en filtrant uniquement les documents cherchés.
 - ✓ Activer l'option permettant d'utiliser la mémoire disque.

Il existe 2 façons pour enregistrer le résultat d'un pipeline d'agrégation :

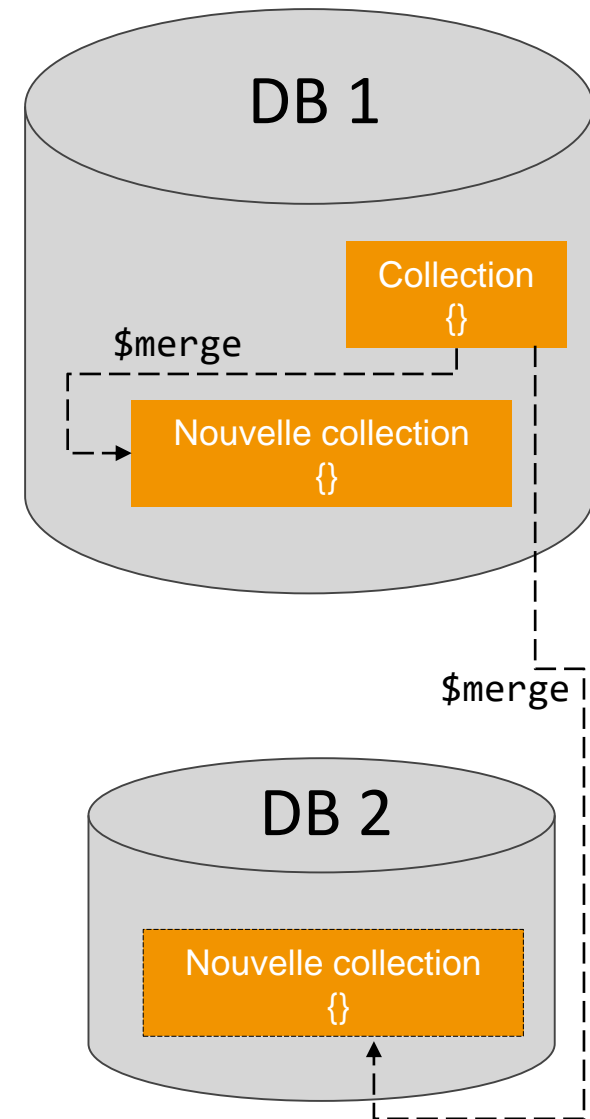
- \$out
- \$merge (nouvelle fonctionnalité apportée avec la version MongoDB 4.2)

```
{"$out" : "<nom de la collection>" }
```



- Enregistre l'output (résultat) du pipeline d'agrégation dans une nouvelle collection
 - Si une collection portant le même nom qu'une collection existante :
 - ✓ Son contenu **sera écrasé.**
 - ✓ Ses **indexes** seront **préservés.**
- L'enregistrement ne sera pas effectué si une erreur s'est produite dans le pipeline.
- La collection résultante sera créée dans la même BD que la collection source, ou dans une BD différente (à partir de la version 4.4.)
- Doit être la dernière étape du pipeline.
- La nouvelle collection **doit être « unsharded »**

- Plus riche et flexible grâce à un ensemble d'options
- Possibilité d'exporter vers une collection **quelque soit son partitionnement** (i.e. avec ou sans sharding)
- Possibilité d'enregistrer dans la même BD ou dans une BD différente que celle de la BD source de la collection.
- Possibilité de **fusionner** le résultat du pipeline avec les documents d'une collection existante sans nécessairement écraser son contenu (d'où le nom « merge »)
- Possibilité de **choisir et d'ajuster la façon de fusion des documents** résultants du pipeline avec ceux de d'une collection existante
- Doit être la dernière étape du pipeline.



☐ Syntaxe générale

```
{  
  $merge : {  
    into : <collection cible>,  
    on : <clé de matching> OU [<clé de matching 1>, ... ],      //option  
    let : <variables>,                                         //option  
    whenMatched : <action>,                                   //option  
    whenNotMatched : <action>                                 //option  
  }  
}
```

```
{  
  $merge : {  
    into : <collection cible> ,  
  }  
}
```

- Syntaxe le plus simple (toutes les options par défaut):

```
{ $merge : "nouvelleCollection" }
```

- Syntaxe en utilisant une BD différente :

```
{ $merge : {  
  into : { "db" : "nomBD" , "coll" : "nouvelleCollection" }  
}
```



```
{
  $merge : {
    into : <collection cible> ,
    on : <clé de matching> OU [<clé de matching 1>, ... ],
  }
}
```

- La clé de matching est spécifiée par le paramètre **on**.
 - Par défaut le serveur utilise :
 - L'identifiant des document **_id** pour un **partitionnement sans sharding** `on : "_id"`
 - Combinaison de l'identifiant des documents et la clé du shard pour un **partitionnement avec sharding** `on : ["_id", "shardKey(s)"]`
 - Si on spécifie une clé de fusion (merging key), que ce soit un champ ou Array, **il faut qu'elle(s) soit un index unique.**

```
{  
  $merge : {  
    into : <collection cible> ,  
    on   : <clé de matching> OU [<clé de matching 1>, ... ],  
  }  
}
```

- La clé de matching peut être soit un champ unique ou un Array de champs:

```
on : [ "_id" , "date" ]
```

- La valeur associée à la clé de matching **ne doit être ni nulle ni un Array de valeurs.**
- Le pipeline ne doit pas modifier la clé de matching.
- Si la collection output n'existe pas, l'identifiant de matching **doit être (et est par défaut) l'identifiant _id.**

```
{  
  $merge : {  
    whenMatched : <action>,    //option  
    whenNotMatched : <action>  //option  
  }  
}
```

Il existe deux possibilités :

- Soit on trouve une correspondance entre le résultat du pipeline et la collection output
- Soit il n'y pas de correspondance entre le résultat du pipeline et la collection output

```
{  
  $merge : {  
    whenMatched : < replace | keepExisting | merge | fail | [pipeline] >,  
  }  
}
```

- “replace” : remplace le document existant dans la collection output par celui correspondant au pipeline d'agrégation
- “keepExisting” : garder le document existant dans la collection output
- “fail” : arrêtez et échouez l'opération d'agrégation.

NB: Les modifications déjà apportées aux à la collection ne seront pas annulées.

```
{
  $merge : {
    whenMatched : < replace | keepExisting | merge | fail | [pipeline] >,
  }
}
```

- “merge” (par défaut) : fusionner le deux documents ayant un matching
 - Si le document du pipeline contient de nouveaux champs, ils seront ajoutés dans le document existant.
 - Si le document du pipeline contient des champs communs avec celui de la collection output, les valeurs de ce dernier seront remplacées par celles du document d'agrégation.

Collection existante : { _id: 1, a: 1, b: 1 }
Résultat du pipeline : { _id: 1, b: 5, z: 1 }
} **Résultat fusion : { _id: 1, a: 1, b: 5, z: 1 }**

- [pipeline] : effectuer un traitement autre que les options précédentes, en utilisant un pipeline à définir.

```
{
  $merge : {
    whenNotMatched : < insert | discard | fail>,
  }
}
```

- “insert” (par défaut) : Insérer le document dans la collection output (scénario similaire à **upsert**).
- “discard” : Rejeter le document et ne pas l’insérer dans la collection output.
- “fail” : Arrêtez et échouez l'opération d'agrégation.

NB: Les modifications déjà apportées à la collection ne seront pas annulées.

```
{  
  $merge : {  
    let : <variables>,  
    whenMatched : <action>  
  }  
}
```

let:

- Option permettant de spécifier des variables à partir des champs des documents résultants du pipeline d'agrégation.
- Ces variables seront utilisées à priori par le pipeline **whenMatched**
- Ces variables sont accessibles en utilisant la syntaxe **\$\$new.<variable>**

❏ Remarques:

- Si l'_id n'existe pas dans les documents du pipeline d'agrégation, l'étape \$merge ajoute les champs _id avec des valeurs d'instances ObjectId générées automatiquement.

```
db.airbnb.aggregate( [  
    { "$project": { "_id": 0 } },  
    { "$merge" : { into : "nouvelleCollection" } }  
] )
```

- L'étape \$merge crée la collection output si elle n'existe pas. La collection est créée (et est visible) dès l'écriture du premier document par \$merge.
- Si le pipeline d'agrégation échoue, les modifications déjà effectuées avant l'erreur ne seront pas perdues (pas de rollback).

❏ Remarques:

- Si la collection output n'existe pas, \$merge **exige** que la clé de matching spécifiée par **on** soit le champ **_id**



Afin de pouvoir utiliser un autre champ dans **on** pour la création d'une collection non existante, il est recommandé de créer en avance la collection en question en spécifiant le champ à utiliser dans **on** comme étant un **indexe unique**.

```
db.newDailySales201905.createIndex( { salesDate: 1 }, { unique: true } )

db.sales.aggregate( [
  { "$match": { date: { $gte: new Date("2019-05-01"), $lt: new Date("2019-06-01") } } },
  { "$group": { _id: { $dateToString: { format: "%Y-%m-%d", date: "$date" } }, totalqty: { $sum: "$quantity" } } },
  { "$project": { _id: 0, salesDate: { $toDate: "$_id" }, totalqty: 1 } },
  { "$merge" : { into : "newDailySales201905", on: "salesDate" } }
] )
```

❑ Exemples 1 :

```
{ $merge: {  
  into: "commandes" ,  
  whenMatched:{"$addField": { "total" : { "$sum" : ["$total" , "$$new.total"] }}} // Exemple option : [pipeline]  
}
```

Champ existant dans
la collection output

Champ issue des documents du
pipeline d'agrégation (**\$\$new**)

❑ Exemples 2:

```
{ $merge: {  
  into: "commandes",  
  whenMatched: { "$set" : { "total" : { "$sum" : [ "$total" , "$$new.total" ] }}} // Même option : [pipeline]  
  // Même exemple mais avec $set  
}
```

Champ existant dans la
collection output

Champ issue des documents du
pipeline d'agrégation (**\$\$new**)

```
// OUTPUT PIPELINE AGREGATION  
{  
  _id : "37",  
  total : 64,  
  f1: "x"  
}
```

+

```
// COLLECTION TARGET  
{  
  _id : "37",  
  total : 245,  
  f1: "yyy"  
}
```



```
// COLLECTION TARGET APRES $MERGE  
{  
  _id : "37",  
  total : 309 ,  
  f1: "yyy"  
}
```

❑ Exemples 3:

```
{ $merge: {  
  into: "commandes" ,  
  let : {"total_intermediaire" : "$total"}  
  whenMatched:{"$set" : { "total" : {"$sum" : ["$total" , "$$total_intermediaire"] }}}  
  // MÊME EXEMPLE EN UTILISANT : LET  
}  
}
```

Vu que nous avons utilisé **let**, le champ total du pipeline est référencé par la variable `total_intermediaire` sans utiliser **\$\$new**

- MongoDB permet de créer des **vues dématérialisées** à partir d'une collection source ou une autre vue **en utilisant un pipeline d'agrégation**.
- *Read Only* : aucune opération de modification/écriture n'est autorisée.
- Hérite les mêmes indexes que la collection source et doit être dans la même BD que la collection source.
- Le pipeline de définition de la vue ne doit pas contenir les étapes \$out ou \$merge
- Permettent de faire un Slicing sur les collections sources:
 - **Slicing horizontal (en utilisant \$match)** : agit sur le nombre de documents retournés.
 - **Slicing vertical (en utilisant l'étape \$project ou autre)** : agit sur la forme des documents retournés.

☐ Restrictions:

- On ne peut pas renommer la vue.
- Pas d'opérations sur les index (update, create).
- Pas de \$text.
- Pas de MapReduce.
- Restrictions méthode find() : ne peut pas inclure les opérateurs de projection suivants
 - \$
 - \$elementMatch
 - \$meta
 - \$slice.

❑ Méthodes d'accès en lecture autorisées :

- `db.nomVue.find()`
- `db.nomVue.findOne()`
- `db.nomVue.countDocuments()`
- `db.nomVue.estimatedDocumentCount()`
- `db.nomVue.count()`
- `db.nomVue.distinct()`
- `db.nomVue.aggregate()`

! Oui ! On peut faire un pipeline d'agrégation en ayant comme input une vue.

☐ Syntaxe

En utilisant la méthode `createView()`

```
db.createView(  
    "<viewName>",  
    "<source>",  
    [<pipeline>],  
    {  
        "collation" : { <collation> }  
    }  
)
```

En utilisant la méthode `createCollection()`

```
db.createCollection(  
    "<viewName>",  
    {  
        "viewOn" : "<source>",  
        "pipeline" : [<pipeline>],  
        "collation" : { <collation> }  
    }  
)
```


❏ Exemple

Créer 3 vues contenant des données clients en fonction de leurs catégories (gold, silver, bronze)

```
db.createView("bronze_banking", "customers", [
  {
    "$match": { "accountType": "bronze" }
  },
  {
    "$project": {
      "_id": 0,
      "name": {
        "$concat": [
          { "$cond": [{ "$eq": ["$gender", "female"] }, "Miss", "Mr."] },
          " ",
          "$name.first",
          " ",
          "$name.last"
        ]
      },
      "phone": 1,
      "email": 1,
      "address": 1,
      "account_ending": { "$substr": ["$accountNumber", 7, -1] }
    }
  }
])
```

```
db.getCollectionInfos()
db.system.views.find()
```



<Exemple cours shell>

Plan

1. Introduction aux Big Data
2. Les bases de données NoSQL
3. La solution MongoDB
4. Modélisation des données Big Data
5. Manipulation des données avec MongoDB (CRUD)
6. Agrégation des données
 - 6.1. Map-Reduce avec MongoDB
 - 6.2. Framework d'agrégation (FA)
 - 6.3. Map-Reduce VS Framework d'agrégation
6. Jointures et références
7. Recherche d'information

Exemple : Calculer le montant total à payer pour chaque client

```
db.orders.aggregate([
  { $group: { _id: "$cust_id", value: { $sum: "$price" } } },
  { $out: "agg_alternative_1" }
])
```

Solution en utilisant le **pipeline d'agrégation**

```
var mapFunction1 = function() {
  emit(this.cust_id, this.price);
};

var reduceFunction1 = function(keyCustId, valuesPrices) {
  return Array.sum(valuesPrices);
};

db.orders.mapReduce(
  mapFunction1,
  reduceFunction1,
  { out: "map_reduce_example" }
)
```

Solution en utilisant le **Map-Reduce**

- La communauté, la documentation, et le support MongoDB recommandent d'utiliser des pipelines d'agrégation, plutôt que d'utiliser Map-Reduce.

+ Meilleure performance

+ Simplicité

Plan

1. Introduction aux Big Data
2. Les bases de données NoSQL
3. Modélisation des données Big Data
4. La solution MongoDB
5. Manipulation des données avec MongoDB (CRUD)
6. Agrégation des données
7. Jointures et références
8. Recherche d'information

- Contrairement à ce qui est dit souvent, MongoDB permet de représenter les relations entre les objets
- Une bonne conception est l'élément clé pour garantir une bonne performance du système final.
- Les questions auxquelles il faut répondre pour avoir une bonne conception sont :
 - **Fonctionnel** : Quel est le besoin du client ?
 - **Fonctionnel** : Quel besoin d'interrogation as-tu besoin pour satisfaire le besoin du client ?
 - **Conception** : Quelles sont les relations dans notre modèle Data (types et cardinalités) ?
 - **Conception** : Quelle information doit être référencée (referenced) ?
 - **Conception** : Quelle information doit être embarquée (embedded) ?
 - **Conception** : De quel côté doit-on effectuer l'imbrication ?



☐ Duplication des données

- Y a-t-il un besoin nécessitant la duplication des données ?
- Quelles sont les données à dupliquer ?

☐ Consistance de données

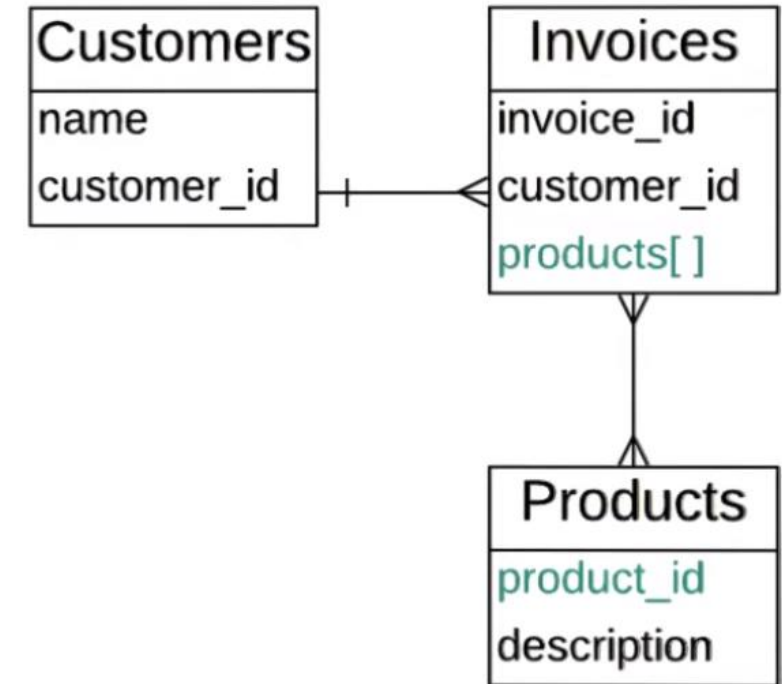
- Quel est le taux acceptable d'un point de vu métier pour le délai de consistance/cohérence des données ?
- Quelle stratégie et quel process à établir pour MAJ les données en fonction du taux de tolérance ? (exemple: MAJ en utilisant des Batch)

☐ Intégrité référentielle

- Quelles sont les données nécessitant de contrôler et MAJ l'intégrité référentielle ?
- Quel est le process à implémenter pour assurer l'intégrité référentielle ?

❑ Les différents types de relations :

- One-to-one [1-1]
- One-to-many [1-N]
- Many-to-many [N-N]
- **One-to-zillions [min-median-max] : pour BigData**



Représentation des relations en MongoDB :

1. **Imbrication** de documents [Embedding]
2. **Références** [Linking]

❑ One-to-one [1-1]:

- L'imbrication d'un attribut ou d'un champ est le choix le plus recommandé dans ce cas.
- Possibilité d'utiliser les sous-documents pour organiser les champs

users [10M]
_id: <objectId>
name: <string>
street: <string>
city: <string>
zip: <string>
shipping_street: <string>
shipping_city: <string>
shipping_zip: <string>

Ou bien



Laquelle des deux conception
est la meilleurs à votre avis ?

users [10M]				
_id: <objectId>				
name: <string>				
<table><tr><th>address</th></tr><tr><td>street: <string></td></tr><tr><td>city: <string></td></tr><tr><td>zip: <string></td></tr></table>	address	street: <string>	city: <string>	zip: <string>
address				
street: <string>				
city: <string>				
zip: <string>				
<table><tr><th>shipping_address</th></tr><tr><td>street: <string></td></tr><tr><td>city: <string></td></tr><tr><td>zip: <string></td></tr></table>	shipping_address	street: <string>	city: <string>	zip: <string>
shipping_address				
street: <string>				
city: <string>				
zip: <string>				

❑ One-to-one [1-1] :

- L'imbrication d'un attribut ou d'un champ est le choix le plus recommandé dans ce cas.
- Possibilité d'utiliser les sous-documents pour organiser les champs

users [10M]
_id: <objectId>
name: <string>
street: <string>
city: <string>
zip: <string>
shipping_street: <string>
shipping_city: <string>
shipping_zip: <string>

Ou bien
(Plus de simplicité)

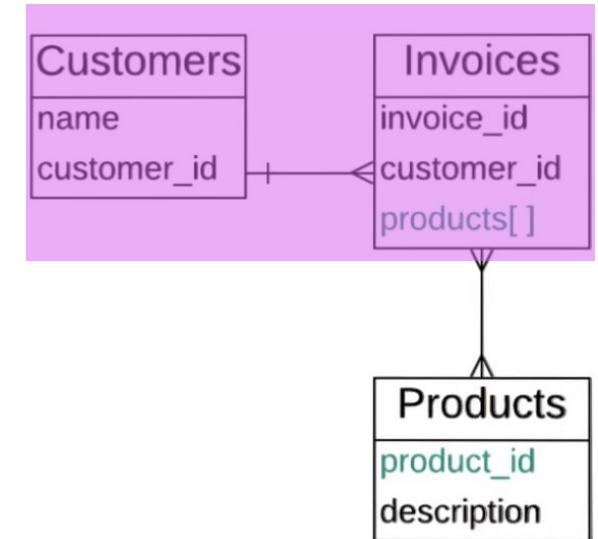


NB : La relation 1-1 est respectée dans les deux, mais la représentation avec des documents imbriqués est recommandée pour plus clarté et pour prévoir certains besoins comme la **protection des données personnelles** (e.g. anonymisation).

users [10M]				
_id: <objectId>				
name: <string>				
<table><tr><th>address</th></tr><tr><td>street: <string></td></tr><tr><td>city: <string></td></tr><tr><td>zip: <string></td></tr></table>	address	street: <string>	city: <string>	zip: <string>
address				
street: <string>				
city: <string>				
zip: <string>				
<table><tr><th>shipping_address</th></tr><tr><td>street: <string></td></tr><tr><td>city: <string></td></tr><tr><td>zip: <string></td></tr></table>	shipping_address	street: <string>	city: <string>	zip: <string>
shipping_address				
street: <string>				
city: <string>				
zip: <string>				

❑ One-to-many [1-N] :

- Plusieurs choix :
 1. **Embedding** ou **référencement** ?
 2. De quelle coté *l'embedding /référencement* doit être effectué ?



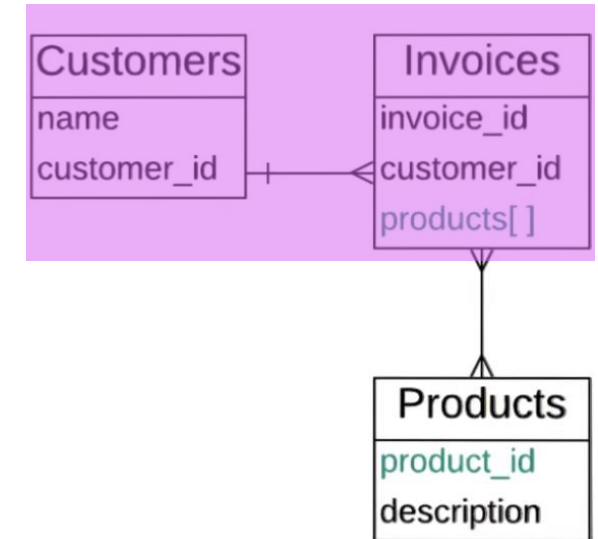
➡ L'embedding est **recommandé dans ce cas pour la simplicité**



A votre avis, de quel coté doit-on faire l'embedding ?

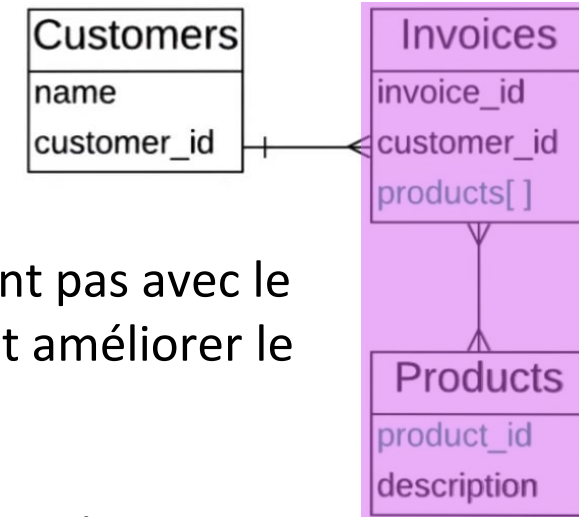
❑ One-to-many [1-N] :

- ➔ L'embedding se fait (recommandé) **dans la collection la plus interrogée**
- ➔ Le référencement peut être envisagé et recommandé **quand les informations imbriquées ne sont pas importantes pour la majorité des requêtes effectuées.**



❑ Many-to-many [N-N] :

- Il faut s'assurer que la relation ne peut pas être simplifiée
- L'embedding est recommandé dans ce cas **dans la collection la plus interrogée**



- ➔ **L'embedding est préférable** surtout **pour les informations statiques**, qui ne changent pas avec le temps (pour éviter les problèmes d'intégrité référentielle et les MAJ) et qui peuvent améliorer le modèle en les dupliquant.
- ➔ **Le référencement est préférable** si on veut éviter de gérer la duplication et la gestion des MAJ et de l'intégrité référentielle que cela peut impliquer.

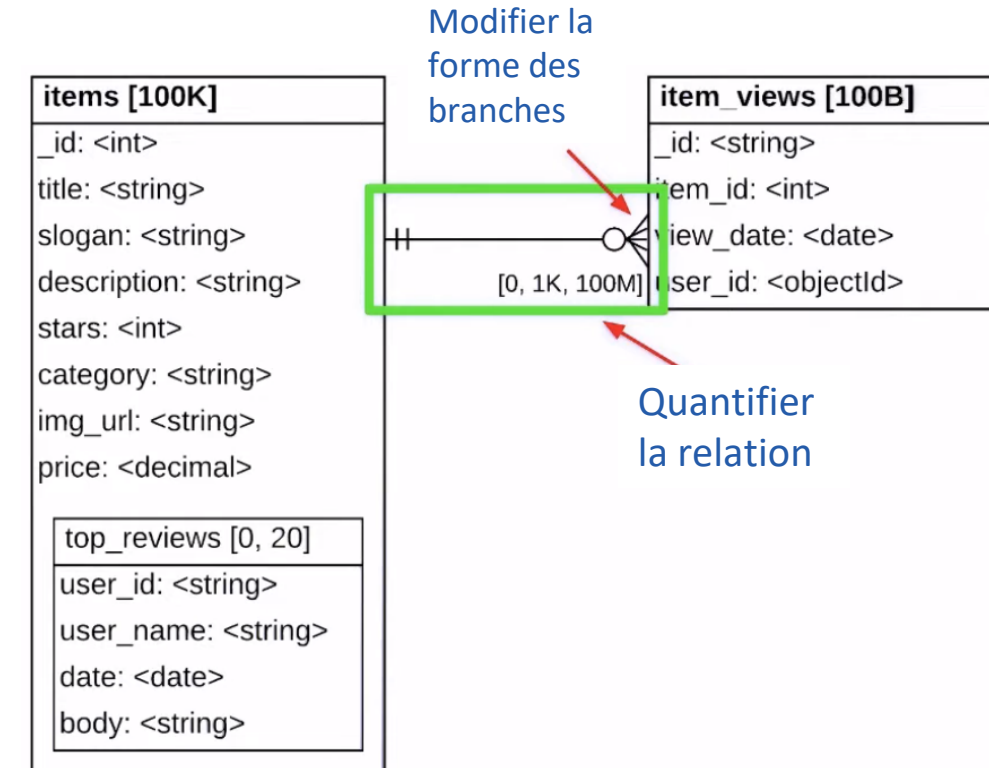
7. Jointures et références

7.2. Conception : Imbrication Versus référencement

❑ One-to-zillions

- Il s'agit d'un **cas particulier de la relation [1-N]** dans lequel la borne **Many (N)** peut atteindre plus que **10.000**.
- Il faut surtout **quantifier la cardinalité max (approximativement) du côté zellion**
- Souligner le fait qu'avoir des cardinalités larges peut **impacter fortement certains choix liés à la conception et la capacité de calcul et mémoire pour les requêtes**

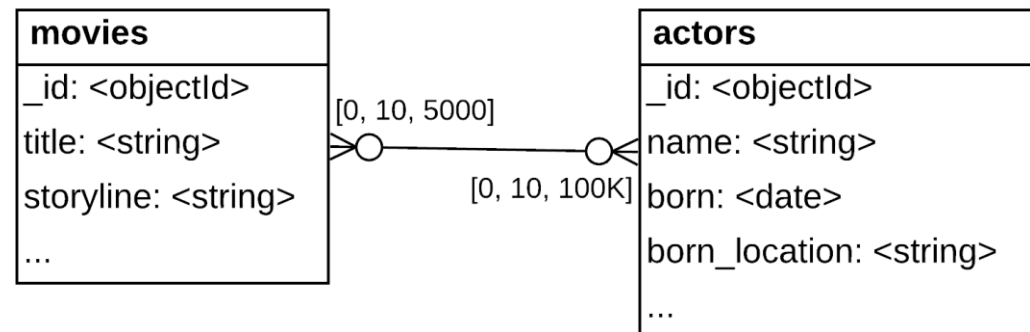
➡ Faire attention aux requêtes et au code qui traite des documents de « zillions »



 *Exemple : Un compte twitter/instagram peut suivre 1 ou plusieurs autres comptes et avoir 1 ou plusieurs voir même des millions de followers*

□ Exercice:

- ? 1. Imbriquer les acteurs dans les films implique une duplication des données des acteurs ?
- ? 2. En utilisant une collection pour les films et une autre pour les acteurs, tous les documents des films doivent contenir obligatoirement un Array pour référencer les acteurs du film en question, ET chaque document d'acteur doit contenir un Array pour référencer tous les films dans lesquels l'acteur joue ?
- ? 3. Imbriquer les acteur dans les films ne supprime pas le besoin d'avoir une collection séparée pour stocker tous les acteurs ?



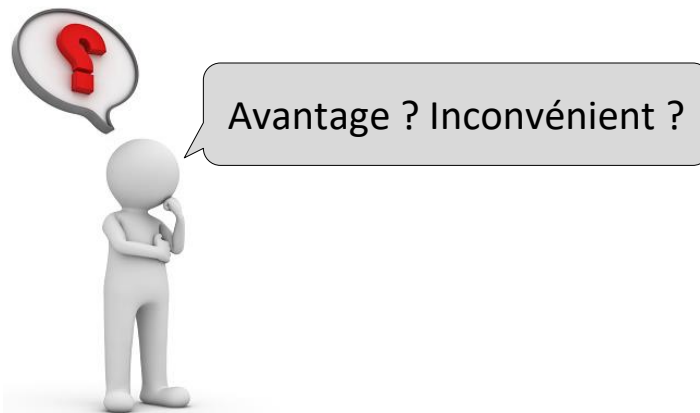
❑ Représentation des relations en MongoDB :

1. **Imbrication de documents [Embedding]** (relations de type 1-1 ou 1-N)
2. **Références [Linking]** (relations de type N-N)

❑ Il existe 3 façons en MongoDB pour référencier les données (≈ aux jointures en SQL) :

1. **Référencement manuel**
2. **DBRefs**
3. **\$lookup et \$graphLookup (Pipeline d'agrégation)** : à partir de MongoDB 3.2

- Placer l'identifiant (`_id`) d'un document dans un autre.
- Faire une requête pour extraire le document référencé.



```
nouveau_id = ObjectId()

db.adresses.insert({
  "_id": nouveau_id,
  "name": "Domicile",
  "rue": "22 Rue de la Pompe",
  "ville" : "Paris",
  "code_postale" : 75016
})

db.personnes.insert({
  "nom": "BERNARD",
  "age" : 32
  "adresse": [nouveau_id, nouveau_id_2 ]
})
```

- Placer l'identifiant (`_id`) d'un document dans un autre.
- Faire une requête pour extraire le document référencé.

 Simplicité

 Aucune information sur la collection ou la BD du document référencé

```
nouveau_id = ObjectId()

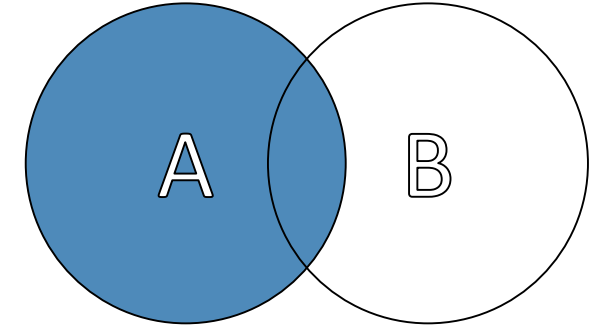
db.adresses.insert({
  "_id": nouveau_id,
  "name": "Domicile",
  "rue": "22 Rue de la Pompe",
  "ville" : "Paris",
  "code_postale" : 75016
})

db.personnes.insert({
  "nom": "BERNARD",
  "age" : 32
  "adresse": [nouveau_id, nouveau_id_2 ]
})
```

- Il s'agit d'une **convention** pour représenter un document, plutôt qu'un type de références spécifique.
- L'ordre des champs BDRefs (\$ref, \$id, \$db) doit être respecté tel qu'illustré.
- Utile surtout si on veut référencier à la fois plusieurs documents appartenant à plusieurs BDs à partir d'un document donné.

```
{
  "_id" : ObjectId("5126bbf64aed4daf9e2ab771"),
  // .. champs du document
  "adresse" : {
    "$ref" : "<nom collection du document référencié>",
    "$id" : <identifiant doc. référencié>,
    // Exemple : ObjectId("5126bc054aed4daf9e2ab772")
    "$db" : "<nom BD du document référencié>"
  }
}
```

- Permet de combiner les données de deux collections différentes.
- En SQL, \$lookup est équivalente à un LEFT JOIN (A LEFT JOIN B)



❑ Syntaxe

```
db.A.aggregate([
  {
    "$lookup": {
      "from": <collection avec laquelle on effectue la jointure (B)>,
      "localField": <champ du document courant (A)>,
      "foreignField": <champ de la collection à joindre (B)>,
      "as": <nom du champ (Array) contenant les documents résultants de la jointure>
    }
  }
])
```



<Exemple cours shell>

❑ A retenir

- La collection **from** doit être dans la même base de données.
- La collection **from** ne doit pas avoir un partitionnement avec sharding.
- Les valeurs de **localField** et **foreignField** peuvent être soit des Array ou autre, et sont matchés en fonction d'une égalité stricte.
- On peut choisir n'importe quel nom pour **as**, toutefois **si un champ portant ce même nom existe déjà** dans le document courant, **il sera remplacé** par le résultat de la jointure.
- En cas de correspondance, les documents qui matchent la jointure seront ajoutés dans la collection invoquant l'agrégation (ajout temporaire dans le pipeline sans aucune modification de la collection source).

Plan

1. Introduction aux Big Data
2. Les bases de données NoSQL
3. Modélisation des données Big Data
4. La solution MongoDB
5. Manipulation des données avec MongoDB (CRUD)
6. Agrégation des données
7. Jointures et références
8. Recherche d'information

- Il s'agit d'étudier la manière de retrouver des informations dans un corpus dans le but de répondre à une requête utilisateur.
- Corpus : composé de documents ou d'une ou plusieurs bases de données.
- Type de données : texte, images, son, vidéo, etc.
- Exemples :
 - Moteurs de recherche (le plus populaire)
 - Systèmes de recommandation (e.g. Amazon)

Les clients ont également acheté ces appareils Amazon




```
{
  $text:
  {
    $search: <string>,
    $language: <string>,           //option
    $caseSensitive: <boolean>,    //option
    $diacriticSensitive: <boolean> //option
  }
}
```

- ❑ Permet d'effectuer une recherche textuelle sur le contenu des champs **contenant un index textuel**
 - `$search` : contient la chaîne de caractères que MongoDB analyse et utilise pour interroger l'index de texte.
 - `$language` : préciser le langage pour permettre à MongoDB de déterminer la liste de *stop words* et les règles de racinisation (*tokenisation/stemming*).
 - `$caseSensitive` : sensibilité à la casse (faux par défaut).



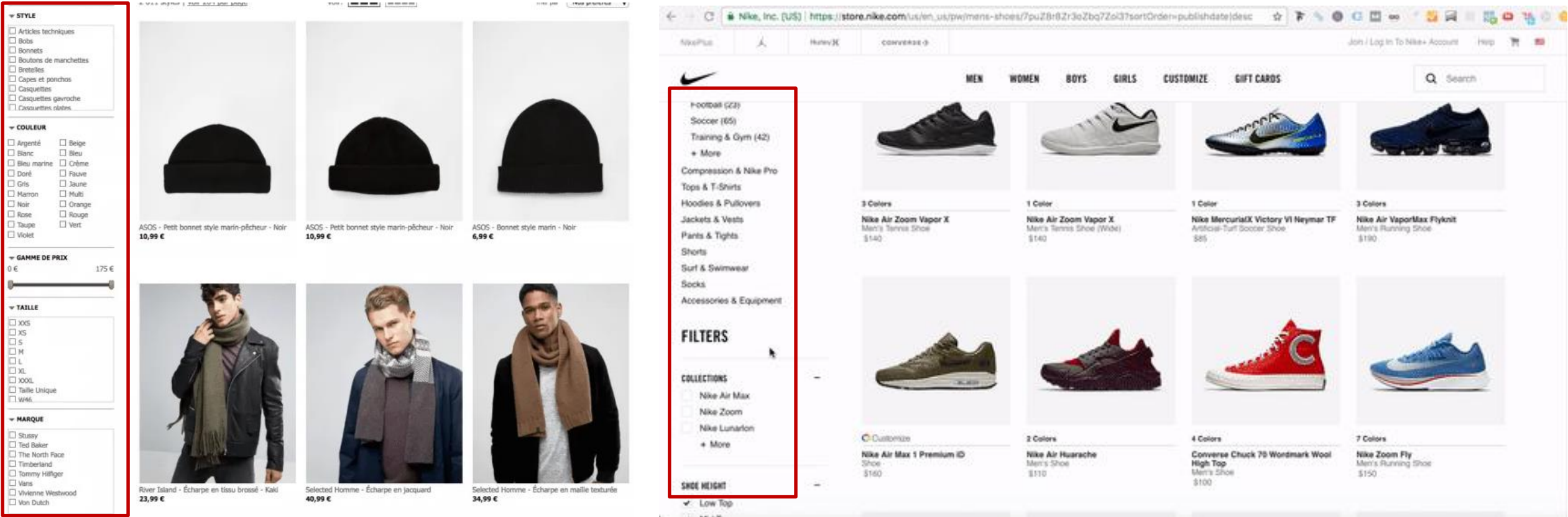
☐ Restrictions :

- L'étape \$match contenant l'opérateur \$text doit obligatoirement être la première.
- Doit être utilisée une seule fois (pas plus) dans une étape d'agrégation.
- Ne peut pas être combinée avec des expressions \$nor ou \$not.
- A ne pas combiner dans une même requête avec des opérateurs impliquant d'autres index d'autres types de données (exemple : combiner avec l'opérateur géolocalisation \$near)

8. Recherche d'information

8.3. Recherche / Navigation à facettes

- Technique de recherche d'information basée principalement sur une classification à facettes.
- Très populaire dans les sites e-commerce pour **faciliter la recherche** des produits et la **navigation** de l'utilisateur.
- Offre à l'utilisateur les moyens de filtrer une collection de données en choisissant un ou plusieurs critères (les facettes).



❑ 2 types de facettes:

- **Les facettes simples** à une seule dimension/requête (Single Query Facets) sont assurées par la nouvelle étape du pipeline d'agrégation :

```
{"sortByCount": "$<champ>"}
```



- Les facettes simples à plages de valeurs liées à des **dimension** :



➤ Manuelles (Manual Buckets)

```
{"bucket": {  
  "groupBy": "$<champ>",  
  "boundaries": [<val1>, <val2>, .., Infinity]  
}}
```



NB:

- Toutes les limites des intervalles doivent avoir le même type de données, sinon une erreur est déclenchée.
- Si la valeur du <champ> d'un document n'est pas dans la plage de valeur indiquée dans « *boundaries* », une erreur sera levée ➡ ajouter l'option « default ».

➤ Automatique (Auto Buckets)



```
{ "$bucket": {  
  "groupBy": "$<champ>",  
  "buckets": <Nombre d'intervalle>  
}}
```



Question : Comment est ce que MongoDB détermine les intervalles ?

❑ Facettes Multiples



```
{ "$facet": {  
  "Nom_Facette_1": [{ <Def_Facet_1> }],  
  "Nom_Facette_2": [{ <Def_Facet_2> }],  
  ...  
  "Nom_Facette_N": [{ <Def_Facet_N> }]  
}}
```



▼ STYLE

- ☐ Articles techniques
- ☐ Bobos
- ☐ Bonnets
- ☐ Boutons de manchettes
- ☐ Bretelles
- ☐ Capes et ponchos
- ☐ Casquettes
- ☐ Casquettes givrées
- ☐ Casquettes plates

▼ COULEUR

<input type="checkbox"/> Argenté	<input type="checkbox"/> Beige
<input type="checkbox"/> Blanc	<input type="checkbox"/> Bleu
<input type="checkbox"/> Bleu marine	<input type="checkbox"/> Crème
<input type="checkbox"/> Doré	<input type="checkbox"/> Fauve
<input type="checkbox"/> Gris	<input type="checkbox"/> Jaune
<input type="checkbox"/> Marron	<input type="checkbox"/> Multi
<input type="checkbox"/> Noir	<input type="checkbox"/> Orange
<input type="checkbox"/> Rose	<input type="checkbox"/> Rouge
<input type="checkbox"/> Taupe	<input type="checkbox"/> Vert
<input type="checkbox"/> Violet	

▼ GAMME DE PRIX

0 € 175 €

▼ TAILLE

- ☐ XXS
- ☐ XS
- ☐ S
- ☐ M
- ☐ L
- ☐ XL
- ☐ XXL
- ☐ Taille Unique
- ☐ Vêtements

▼ MARQUE

- ☐ Stussy
- ☐ Ted Baker
- ☐ The North Face
- ☐ Timberland
- ☐ Tommy Hilfiger
- ☐ Vans
- ☐ Vivienne Westwood
- ☐ Von Dutch

- L'étape `$facet` a comme input les documents filtrés/traités par les étapes précédentes du pipeline.
- Chaque facette définie dans `$facet` aura le **même input** (i.e. documents) que les autres facettes.

! - Les documents ne sont **ni filtrés ni traités** entre les différentes facettes.
- Chaque facette traite les documents d'une façon indépendante des autres facettes, et à partir du même input (ensemble de documents) qui est commun à toutes les facettes de l'étape `$facet`.