

NoSQL, systèmes distribués et passage en production de projets Data

Thierry GAMEIRO MARTINS

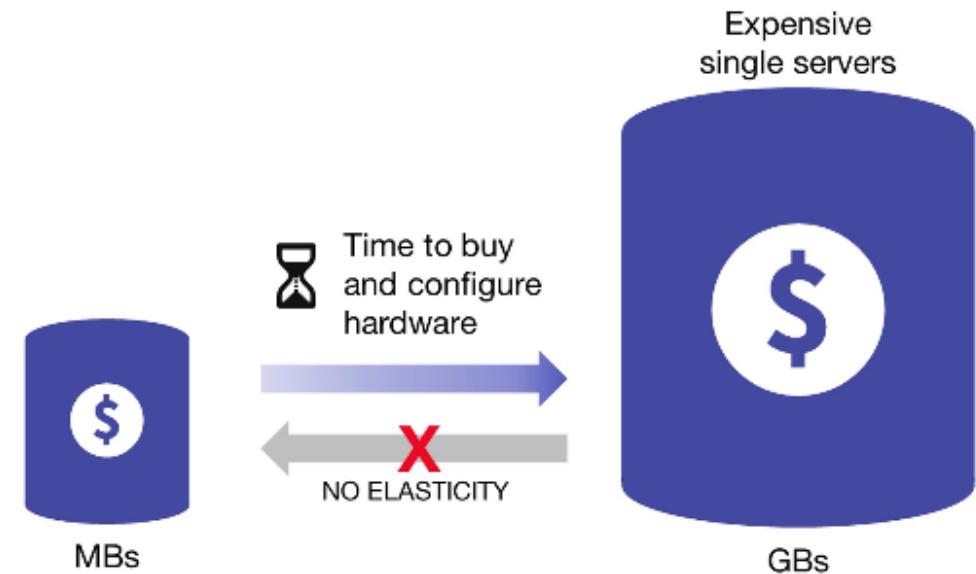
Séances

1. Introduction et prise en main d'Onyxia
- 2. Le stockage des données en NoSQL**
3. Les systèmes de stockage distribués
4. Le passage en production
5. Orchestration par Airflow et pratique DevOps
6. Déploiement conteneurisé sous Kubernetes
7. Introduction au MLOps

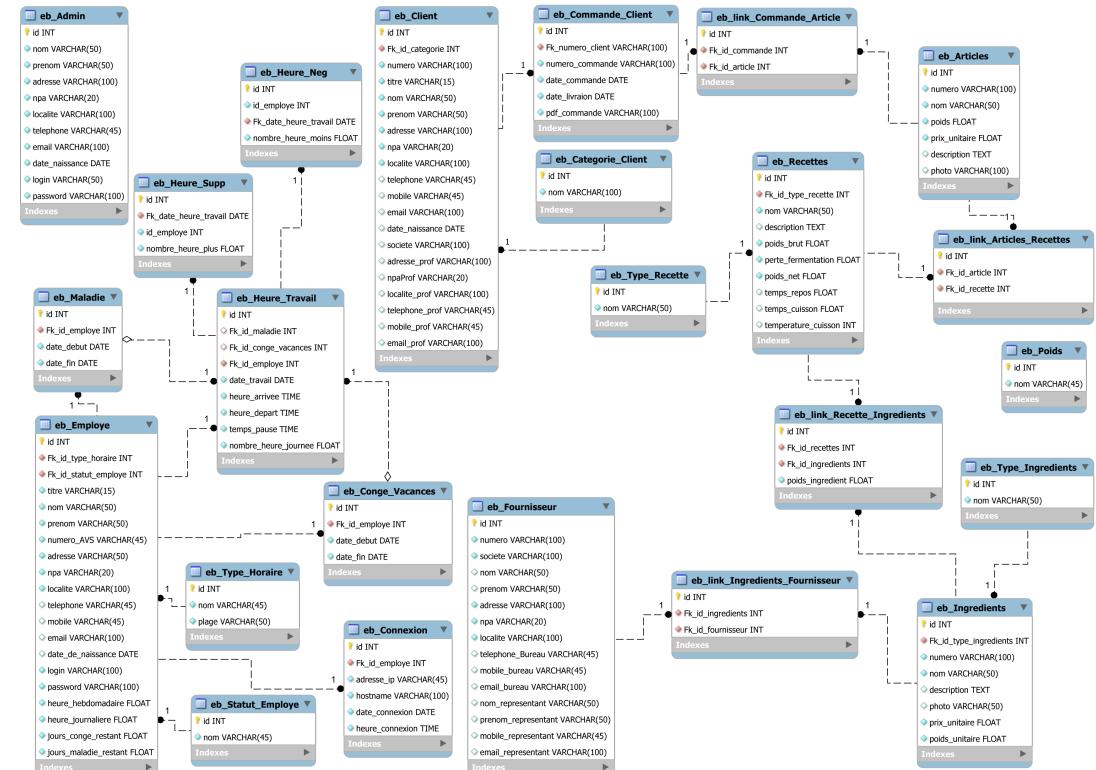
Introduction : les fondements du NoSQL

Les limites des SGBDR

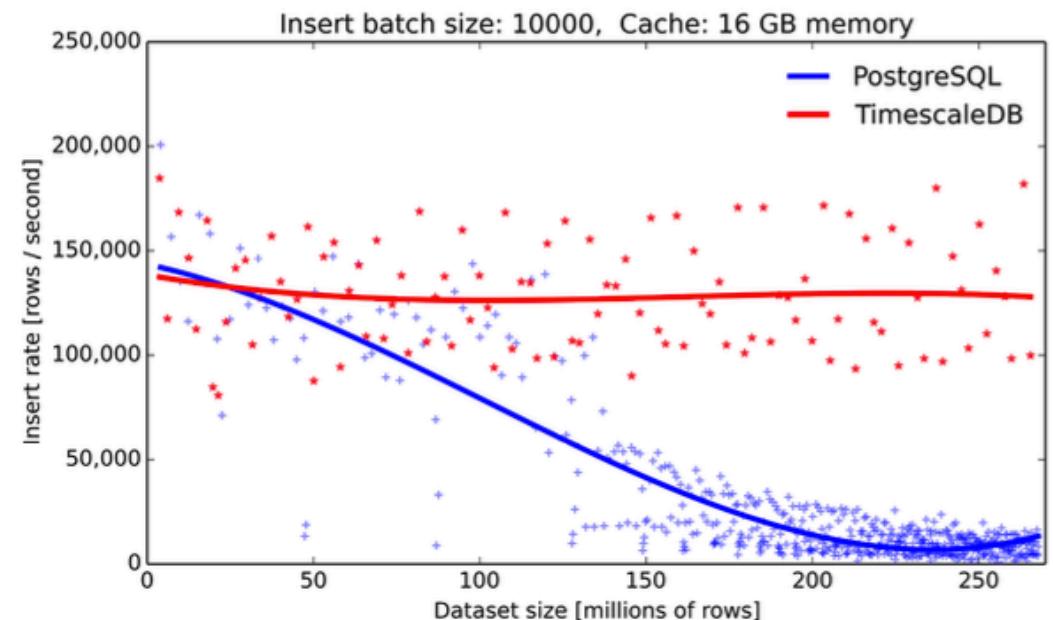
- **Scalabilité horizontale limitée :**
difficulté sur la montée en charge et
sharding complexe
- **Coût élevé :** beaucoup sont sous
licence et assez cher, nécessite des
moyens conséquents pour être résilient



- Complexité de gestion** : gestion difficile dans de grands environnements, flexibilité très faible, et nécessite une expertise pointue
- Gestion des données non structurées** : optimisés pour les données structurée et des schémas bien définis (JSON, images, etc. sont difficiles à stocker)



- **Performance sur de la volumétrie :** performance plus faible sur de grands volumes pour les jointures ou l'insertions de données



Définition du NoSQL

- NoSQL pour *Not Only SQL*
- Utilise les propriétés **BASE** :
 - **Basically Available** : toujours disponible (*tolérance aux pannes*)
 - **Soft-state** : pas de schéma fixe (*gestion de données complexes ou non structurées*)
 - **Eventually Consistent** : la donnée finira par être cohérente (ex : *mise à jour du profil sur les réseaux sociaux*)

Avantages du NoSQL

- **Flexibilité des schémas** : les données peuvent changer au fil du temps
- **Scalabilité horizontale** : facilité à ajouter des nœuds pour augmenter la capacité
- **Performance** : optimisé pour les grandes quantités de données et les systèmes distribués
- **Adapté aux big data et aux applications modernes** : temps réel, IoT, etc.

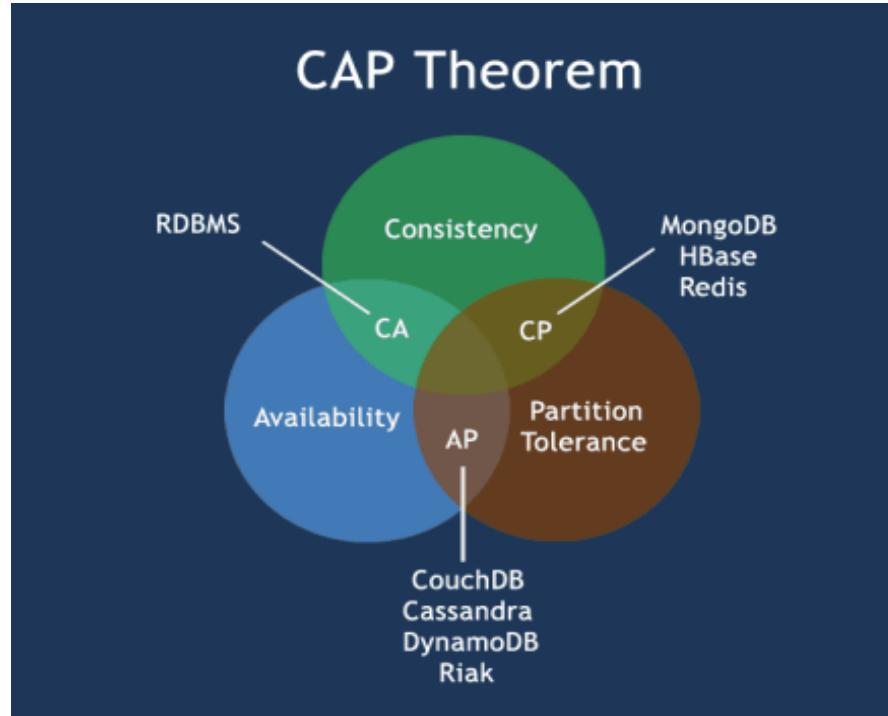
Théorème CAP

Trois propriétés définissent les bases de données

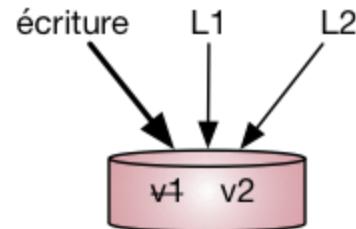
- **Cohérence** : tous les nœuds du système voient exactement les mêmes données au même moment
- **Disponibilité** : toutes les requêtes reçoivent une réponse
- **Distribution** : le système étant partitionné (ou distribué), aucune panne moins importante qu'une coupure totale ne l'empêche de répondre

Théorème de CAP (Brewer, 2000) :

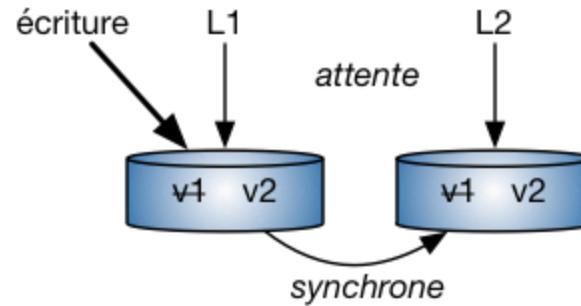
"Dans un système distribué, il est impossible d'obtenir ces trois propriétés en même temps"



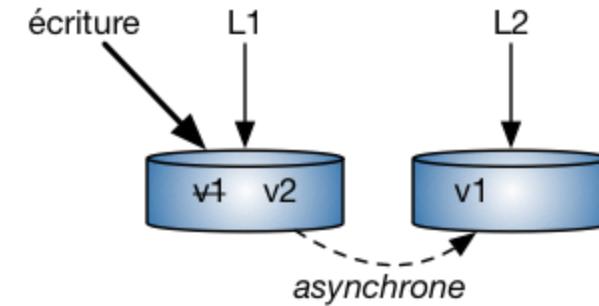
CA
Cohérence + Disponibilité



CP
Cohérence + Distribution



AP
Disponibilité + Distribution



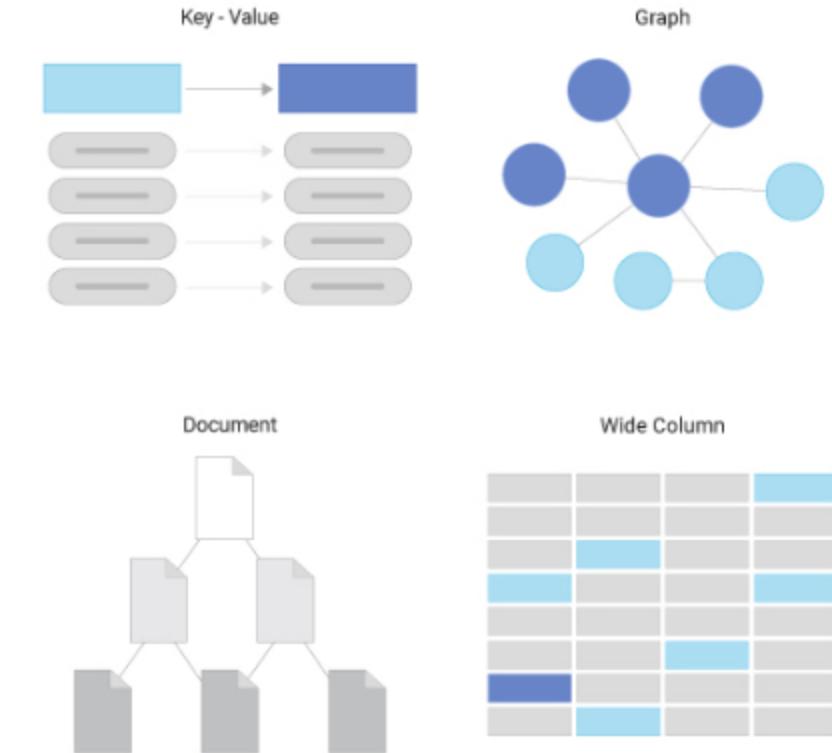
Toutes les requêtes
retournent la même donnée
sans délais

Le système est distribué et
la donnée doit être
cohérente à chaque instant,
l'accès à la donnée est plus
long à cause de la
synchronisation

La donnée est distribuée et
doit être toujours disponible.
Les synchronisations sont
asynchrone et peut conduire
à des incohérences de
données (*serveur DNS*)

Les familles de bases de données NoSQL

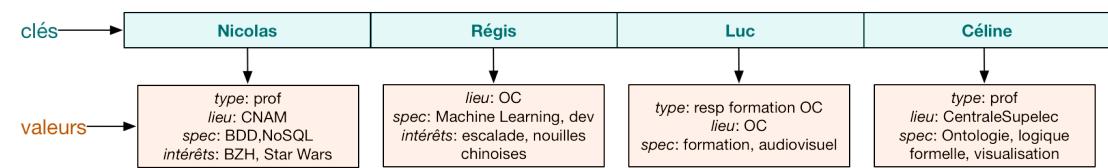
- **Clé-Valeur** (*Redis, DynamoDB*)
- **Orienté documents** (*MongoDB, ElasticSearch*)
- **Orienté colonnes** (*Cassandra, Vertica, HBase*)
- **Graphes** (*Neo4j*)



Clé-valeurs

Un système clé-valeur agit comme un dictionnaire :

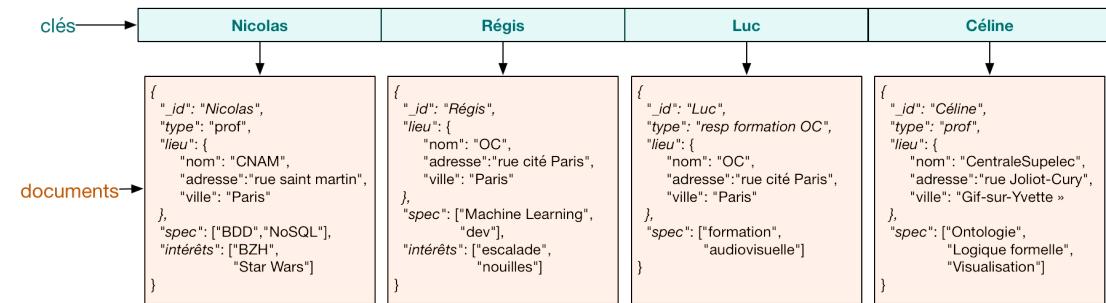
- La clé identifie la donnée de manière unique et permet d'en récupérer la valeur
- La valeur contient n'importe quel type de données
- Performance très élevée en lecture et écriture
- Limité aux opérations *CRUD* et non à son analyse



Orienté document

Manipule des documents contenant des informations avec une structure complexe (listes, dictionnaires avec imbrications).

- Structure en forme de document
(JSON)
- Approche plus structurée que la clé-valeur (possible d'imposer des schémas)
- Propose des langages d'interrogation riches permettant de faire des manipulations complexes



Orienté colonne

Système où les données sont lues par colonnes.

- Solution adaptée pour effectuer des traitements sur des colonnes (comptage, moyennes, etc.) mais plus difficile sur des lignes
- Permet une compression plus efficace, réduisant ainsi l'espace disque nécessaire

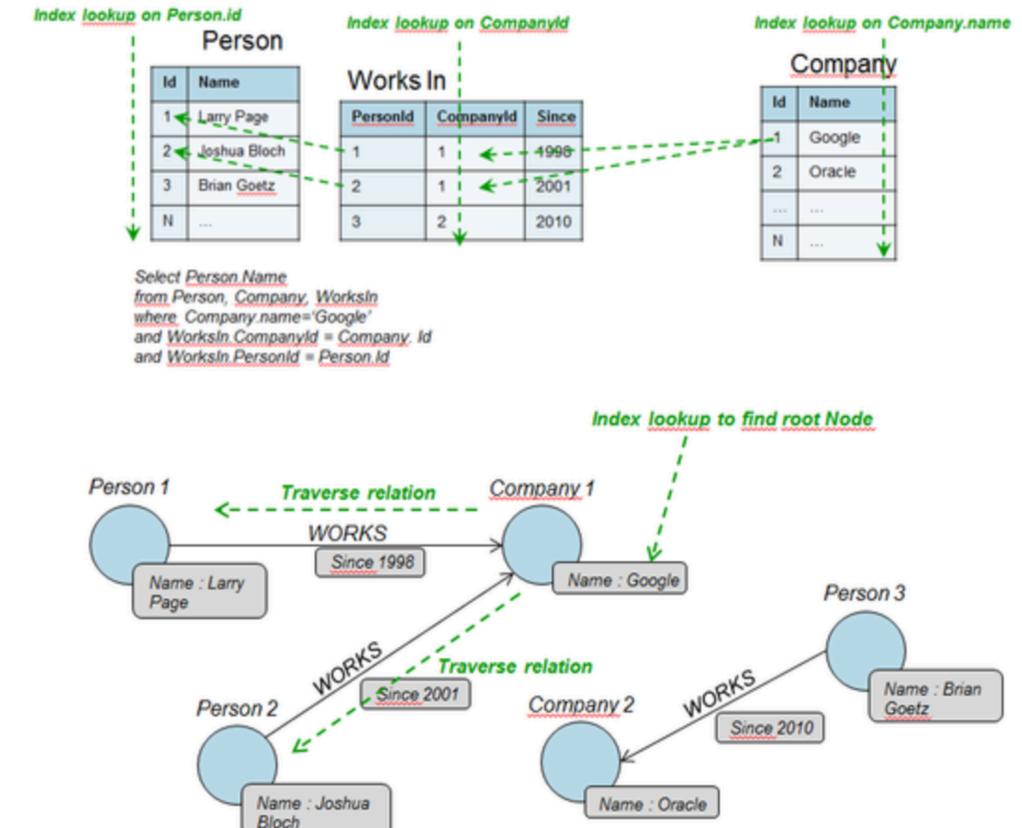
Stockage orienté lignes					Stockage orienté colonnes							
id	type	lieu	spec	intérêts	id	type	id	lieu	id	spec	id	intérêts
Nicolas	prof	CNAM	BDD, NoSQL	BZH, Star Wars	Nicolas	prof	Céline	Centrale Supélec	Nicolas	BDD	Nicolas	BZH
Régis		OC	Machine Learning, Dev	escalade, nouilles chinoises	Céline	prof	Nicolas	CNAM	Nicolas	NoSQL	Nicolas	Star Wars
Luc	resp formation OC	OC	formation, audiovisuel		Luc	resp formation OC	Régis	OC	Régis	Machine Learning	Régis	escalade
Céline	prof	CentraleSupélec	Ontologie, logique formelle, visualisation		Luc	OC	Luc	OC	Régis	Dev	Régis	nouilles chinoises

Graphe

Stockent des données sous formes de :

- **nœuds** : représente une entité
- **liens** : la relation entre les entités
- **propriété** : attributs sur les liens

Utile pour faire de la représentation de liens entre des données (*connexion entre utilisateurs sur les réseaux sociaux, relations suspectes entre comptes bancaires, etc.*)



Redis



Connexion avec le client python

```
import redis
```

```
r = redis.Redis(host='redis', port=6379,  
decode_responses=True)
```

- REmote DIctionnary Server

(<https://redis.io>)

- Vitesse d'écriture et lecture très élevée
- Base de données en mémoire (rapide)
- Accès par une table *hashage*

Quand l'utiliser ?

- Système de stockage de cache
- Information de sessions (les profils, préférences d'utilisateurs)
- File d'attente

Stockage de clés en python

The screenshot shows a code editor interface with tabs for Redis CLI, C#, Java, Node.js, and Python. The Python tab is active, displaying the following code:

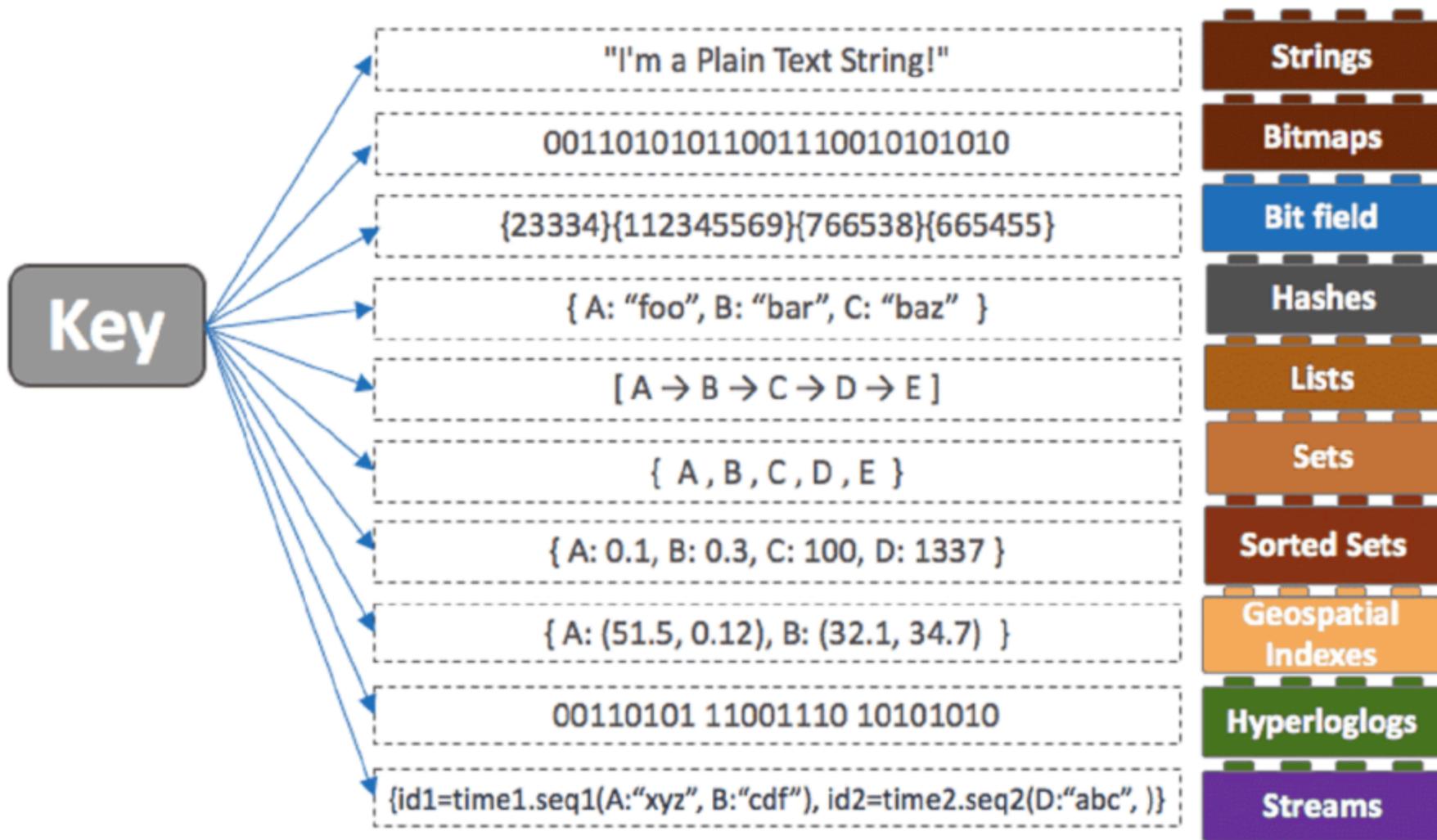
```
>_ Redis CLI      C#      Java      Node.js      Python

res1 = r.hset(
    "bike:1",
    mapping={
        "model": "Deimos",
        "brand": "Ergonom",
        "type": "Enduro bikes",
        "price": 4972,
    },
)
print(res1)
# >>> 4

res2 = r.hget("bike:1", "model")
print(res2)
# >>> 'Deimos'

res3 = r.hget("bike:1", "price")
print(res3)
# >>> '4972'

res4 = r.hgetall("bike:1")
print(res4)
# >>> {'model': 'Deimos', 'brand': 'Ergonom', 'type': 'Enduro bikes', 'price': '4972'}
```



Structure des données

Strings : les données sont stockées sous toutes formes de données (nombres, chaîne, image, etc.)

- `SET` : associe une clé à une valeur
- `GET` : récupère la valeur associée
- `DEL` : supprime la clé
- `GETDEL` : lit et supprime la clé ensuite

```
SET user:01 "hello"
GET user:01
"hello"
DEL user:01
GET user:01
(nil)
```

Possibilité de faire des enregistrements multiples

```
MSET foo1 "bar1" foo2 "bar2" foo3 "bar3"
MGET foo1 foo2 foo3
1) "bar1"
2) "bar2"
3) "bar3"
```

Redis permet une recherche sous la forme `GET user:*`

`TYPE` commande qui permet de récupérer le type d'une clé

Hashage : clés associées à un ensemble de sous-clés et de valeurs accessibles

- `HSET` , `HDEL` , `HGET` : ajoute, récupère ou supprime une sous-clé et une valeur à un objet
- `HGETALL` : récupère toutes les sous-clés et valeurs associées
- `HKEYS` : affiche toutes les clés
- `HEXISTS` : vérifie si une sous-clé existe

```
HSET user:1001 name "Alice" age "30"  
HGETALL user:1001  
"Alice" "30"
```

Liste : collections d'éléments selon leur ordre saisie

- `LPUSH` , `LP0P` : ajoute ou supprime un élément à la tête de la liste
- `LRANGE` : récupère les éléments de la liste.
- `LINDEX` : récupère la valeur d'une liste à un index

```
LPUSH tasks "task1"  
LPUSH tasks "task2"  
LRANGE tasks 0 -1  
1) "task1"  
2) "task2"
```

Set : collections unique de valeurs non ordonnées, utile pour des comparaisons

- `SADD` , `SREM` , `SGET` : ajoute, supprime ou récupère un élément d'un set
- `SINTER` , `SDIFF` : récupère les valeurs communes ou différentes entre deux sets
- `SISMEMBER` : vérifie pour une clé si la valeur existe

```
SADD abo:01 lundi mardi mercredi jeudi  
SADD abo:02 lundi dimanche  
SINTER abo:01 abo:02  
"lundi"
```

Sorted Set : collections d'éléments où chaque élément est associé à un score

- `ZADD` , `ZREM` : ajoute ou supprime un élément à la tête de la liste
- `ZRANK` , `ZREVRANK` : obtient le rang associé à une valeur par rapport à son score (ascendant ou descendant)
- `ZRANGE` : permet de générer un top par rapport aux scores des valeurs entre deux index (-1 pour la dernière valeur)

```
ZADD gagnants user1 90 user2 30 user3 50  
ZRANK gagnants user2  
"2"
```

Autres options

Incrémantation automatique

- `INCR` : augmente de 1 la valeur de la clé
- `DECR` : diminue de 1 la valeur de la clé

```
SET test 1
INCR test
"2"
DECR test
"1"
```

Expiration de clés

- `EXPIRE` : supprime la clé au bout de n secondes
- `TTL` : affiche le *time to live* de la clé
- `SETEX` : crée une clé et lui associe un TTL

```
EXPIRE test 60
TTL test
20
SETEX foo 100 "bar"
```

Options d'expiration

- **NX** : Applique l'expiration uniquement si aucune n'existe
- **XX** : Applique l'expiration uniquement si la clé possède déjà une expiration
- **GT** : Applique l'expiration uniquement si la nouvelle est plus élevée que celle existante
- **LT** : Applique l'expiration uniquement si la nouvelle est moins élevée que celle existante

```
EXPIRE foo 10 NX
```

Modification de clé

- **RENAME** : renome une clé existante
- **APPEND** : ajoute à la clé la valeur associé, si n'existe pas la crée

```
SET foo "Hello"
APPEND foo " World"
GET foo
"Hello World"
RENAME foo bar
GET foo
(nil)
GET bar
"Hello World"
```

Gérer les données geospatiales

Ajouter des données geospatiales

```
GEOADD Paris 2.229307 48.896676 "La Defense Arena"  
GEOADD Paris 2.321951 48.842138 "Tour Montparnasse"  
GEOADD Paris 2.294481 48.858370 "Tour Eiffel"
```

Recherche d'éléments les plus proches selon la localisation

```
GEOSEARCH Paris FROMLONG 2.322214 FROMLAT 48.865840  
BYRADIUS 5 Km WITHDIST  
1) 1) "Tour Eiffel"  
    2) "2.1928"  
2) 1) "Tour Montparnasse"  
    2) "2.6362"
```

Recherche de distance à partir d'un membre

```
GEOSEARCH Paris FROMMEMBER "La Defense Arena"  
BYRADIUS 10 km WITHDIST  
1) 1) "La Defense Arena"  
    2) "0.0000"  
2) 1) "Tour Eiffel"  
    2) "6.3942"  
3) 1) "Tour Montparnasse"  
    2) "9.0963"
```

Administration

Gestion de la configuration

Il est possible de récupérer la configuration de `redis` et de la modifier

```
config get *
config get dbfilename
config set dbfilename "dump2.rdb"
```

```
redis:6379> config get *
1) "lazyfree-lazy-server-del"
2) "no"
3) "lazyfree-lazy-user-flush"
4) "no"
5) "oom-score-adj"
6) "no"
7) "replica-lazy-flush"
8) "no"
9) "repl-backlog-size"
10) "1048576"
11) "no-appendfsync-on-rewrite"
12) "no"
13) "hz"
14) "10"
15) "cluster-link-sendbuf-limit"
16) "0"
```

Sauvegarde de la base

- **RDB (Redis Database File) :**
Sauvegardes périodiques des données (toutes les 60 secondes si 10 modifications ont été effectuées)

```
save 60 10
```

bgsave permet de lancer des sauvegardes en arrière plan car save est bloquant

- **AOF (Append Only File) :**
Journalisation de chaque opération

```
config set appendonly yes
```

Abonnement

Pub/Sub permet à Redis de fonctionner comme un système de messagerie.

- Publier un message sur un canal
- S'abonner à un canal pour recevoir des messages

```
PUBLISH mychannel "Hello"  
SUBSCRIBE mychannel  
UNSUBSCRIBE mychannel
```

Quand ?

- Notifications en temps réel
- Coordination entre services
- Diffusion d'événements système

The image shows three terminal windows side-by-side, each with a dark background and light-colored text. The top window shows a publisher publishing two messages: 'first' and 'second'. The middle window shows a subscriber subscribing to the channel and receiving both messages. The bottom window shows another subscriber receiving the same two messages. All windows have a title bar 'titigrmr — com.docker.cli - docker exec -it 6b27e2c7672 bash — 114x24' and a status bar at the bottom.

```
[127.0.0.1:6379> PUBLISH first OK  
(integer) 2  
[127.0.0.1:6379> PUBLISH second OK  
(integer) 1  
127.0.0.1:6379>  
  
[127.0.0.1:6379> SUBSCRIBE first  
1) "subscribe"  
2) "first"  
3) (integer) 1  
Reading messages... (press Ctrl-C to quit or any key to  
Reading messages... (press Ctrl-C to quit or any key to  
Reading messages... (press Ctrl-C to quit or any key to  
Reading messages... (press Ctrl-C to quit or any key to  
Reading messages... (press Ctrl-C to quit or any key to  
1) "message"  
2) "first"  
3) "OK"  
Reading messages... (press Ctrl-C to quit or any key to  
Reading messages... (press Ctrl-C to quit or any key to  
Reading messages... (press Ctrl-C to quit or any key to  
Reading messages... (press Ctrl-C to quit or any key to  
1) "message"  
2) "second"  
3) "OK"  
Reading messages... (press Ctrl-C to quit or any key to  
Reading messages... (press Ctrl-C to quit or any key to  
Reading messages... (press Ctrl-C to quit or any key to  
1) "message"  
2) "second"  
3) "OK"  
Reading messages... (press Ctrl-C to quit or any key to  
Reading messages... (press Ctrl-C to quit or any key to  
Reading messages... (press Ctrl-C to quit or any key to  
1) "message"
```

MongoDB

- Stocke les données sous forme de documents JSON (semi-structuré) dans des collections
- Modèle flexible sans schéma prédéfini
- Scalabilité horizontale
- Stockage compact Binary JSON (BSON)

Quand l'utiliser ?

- Applications nécessitant une flexibilité des schémas
- Applications en temps réel nécessitant des opérations rapides

Connexion avec le client Python

```
from pymongo import MongoClient  
  
client = MongoClient('mongodb', 27017)  
  
db = client['mydatabase']
```



Collection

La dénormalisation des données

Les données métiers sont souvent stockées dans un format relationnel ou en `csv`

Principe de la normalisation

- Évite la duplication de données
- Assure une meilleure fiabilité de la donnée



À éviter dans une base document

orders

order_id	name	store_id
0	alex	0
1	alice	1
2	tom	0

store

store_id	store_city	store_size
0	Paris	Small
1	Marseille	Large

Dénormalisation

Combiner l'ensemble des données au sein d'un même document plutôt que d'utiliser des jointures entre documents

Pourquoi ?

- Stockage à faible coût aujourd'hui
- Référence à plusieurs documents augmente le temps de réponse
- Simplification des requêtes

orders

```
{  
  "_id": 1,  
  "name": "alex"  
  "store": {  
    {"store_id": 0, "store_city": "Paris", "store_size": "Small"}  
}
```

Les types de données MongoDB

Le type ObjectId

- **Imbrication de documents** : idéal pour les relation 1-1
- **Référence partielle** : répliquer une partie des informations couramment utilisée et référencer les détails via un id
- **Risque des modifications** : couteux si une modification implique de modifier tous les documents
- **Vérifier la taille des document** : dans le cas de relation 1-n, peut alourdir la taille du document (max 16Mo)

```
{
  "_id": ObjectId("6149eae5a1c2f1001e5e8a23"),
  "title": "How to scale MongoDB",
  "author": {
    "name": "John Doe",
    "job": "Data Scientist",
    "user_id": ObjectId("6149eae5a1c2f1001e5e8b24")
    "address": {
      "city": "Paris",
      "state": "France",
      "zip": "75014"
    }
  },
  "comments": [
    {
      "comment_id": ObjectId("6149eae5a1c2f1001e5e8a25"),
      "text": "Great article!",
      "commenter": {
        "name": "Jane Smith",
        "user_id": ObjectId("6149eae5a1c2f1001e5e8b26")
      },
      "created_at": "2023-09-21T14:15:22Z"
    },
    {
      "comment_id": ObjectId("6149eae5a1c2f1001e5e8a26"),
      "text": "I found it very helpful!",
      "commenter": {
        "name": "Alice Brown",
        "user_id": ObjectId("6149eae5a1c2f1001e5e8b27")
      },
      "created_at": "2023-09-21T14:20:15Z"
    }
  ]
}
```

Insertion de données

Ajouter un document

```
db.collection.insertOne({"name": "Alice", "age": 25})
```

Insérer plusieurs documents

```
db.collection.insertMany([{"name": "Bob", "age": 30}, {"name": "Charlie", "age": 35}], { ordered: false })
```

`ordered` : n'impose pas d'ordre à l'insertion et tente l'ensemble
`writeConcern` : gestion de la vérification de l'écriture (`majority` , `0` , `1`)

Utilisation de `mongoimport` pour de l'import en masse (fichiers CSV, JSON, etc.).

```
mongoimport --db mydatabase --collection mycollection --file data.json --jsonArray
```

- Très efficace pour de l'import massif
- Supporte différents formats (JSON, CSV)

MongoDB crée automatiquement les collections lors de la première insertion

Recherche de données

Trouver un document

```
db.collection.findOne({ "name": "Alice" })
```

Trouver plusieurs documents

```
db.collection.find({ "age": { "$gt": 30 } })
```

Filtrage

Valeur exacte : { "name": "Alice" }

Sous-clé : { "address.city": "Metz" }

Regex : { "name": { \$regex: "^A" } }

Paramètres

- **Premier paramètre** : critère de recherche
- **Deuxième paramètre** : projection (champ à inclure/exclure) dans la sortie

```
db.collection.find(  
  { "age": { $gt: 30 } }, // critère  
  { "name": 1, "age": 1 } // projection  
)
```

{ "_id": 0} pour exclure l'id

Comparaisons

- `$gt` et `$lt` : supérieur ou inférieur
`{ "age": { $gt: 30 } }`
- `$gte` et `$lte` : supérieur/inférieur ou égal à
`{ "age": { $gte: 30 } }`
- `$and` ou `$or` : combinaison
`{"age": {"and": [{"$lte": 30}, {"$gte": 40}] }}`
- `$ne` : non égal à
`{ "age": { $ne: 25 } }`

Inclusions/exclusions

- `$in` : inclus dans une liste
`{ "status": { $in: ["Active", "Pending"] } }`
- `$nin` : non inclus dans une liste
`{ "status": { $nin: ["Banned"] } }`
- `$exists` : si la clé existe
`{ "age": { $exists: 1 } }`

Suppression de données

- `deleteOne` : supprime un seul document correspondant au critère
- `deleteMany` : supprime tous les documents correspondant au critère
- Pour supprimer tous les documents d'une collection

```
db.collection.deleteOne({ name: "Alice" })
```

```
db.collection.deleteMany({ age: { $lt: 30 } })
```

```
db.collection.deleteMany({})
```



Vérifier le critère avant toute suppression

Mise à jours des données

- `updateOne` : met à jour un document correspondant au critère
 - `updateMany` : met à jour tous les documents correspondants au critère (ou tous)
 - `replaceOne` : remplace entièrement un document correspondant au critère
 - `replaceMany` : remplace entièrement tous les documents correspondants au critère
- **Premier paramètre** : critère de recherche (quel document modifier)
 - **Second paramètre** : transformation à appliquer (quelles modifications apporter)

```
db.customers.updateOne(  
  { "name": "Alex" },  
  { $set: { "email": "newemail@example.com" } }  
)
```

\$set : définit une nouvelle valeur pour un champ spécifique

```
{ $set: { "email": "updated_email@example.com" } }
```

\$inc : incrément ou décrément une valeur numérique

```
{ $inc: { "age": 1 } } // Ajoute 1 à l'âge
```

\$rename : renomme une clé existante

```
{ $rename: { "oldFieldName": "newFieldName" } }
```

\$unset : supprime un champ

```
{ $unset: { "address": "" } } // Supprime le champ
```

\$mul : multiplie la valeur par un nombre spécifié

```
{ $mul: { "salary": 1.1 } } // Augmente de 10%
```

le paramètre optionnel `{ upsert: true }` permet de créer un nouveau document si aucun n'est trouvé avec le critère de recherche

Pipeline de données

- Ensemble d'étapes qui traitent les documents
- Chaque étape passe les résultats à la suivante

```
db.orders.aggregate([
  { $match: { status: "completed" } },
  { $group: { _id: "$customerId", totalAmount: { $sum: "$amount" } } }
]);
```

- `merge` : permet d'écrire le résultat dans un document (toujours en dernière étape)

```
{ $merge: { into: "myOutput", on: "_id",
whenMatched: "replace",
whenNotMatched: "insert" } }
```

- `$match` : permet de filtrer les documents afin de les passer à l'étape suivante
- `$project` : permet de sélectionner les champs à passer dans la prochaine étape d'une pipeline
- `$group` : permet de regrouper des données et d'y appliquer une expression à chaque groupe

D'autres étapes sont disponibles [ici](#)

- `uwind` : aplatit une liste à afin de produire une clé pour chaque élément dont la valeur est un élément de la liste
- `$lookup` : permet de faire correspondre des colonnes d'un document avec la colonne d'un autre document

inventory

```
{ prodId: 100, price: 20, quantity: 125 },
{ prodId: 101, price: 10, quantity: 234 },
{ prodId: 102, price: 15, quantity: 432 },
```

orders

```
{ orderId: 201, custid: 301, prodId: 100, numPurchased: 20 },
{ orderId: 202, custid: 302, prodId: 101, numPurchased: 10 },
{ orderId: 203, custid: 303, prodId: 102, numPurchased: 5 },
```

Création d'une vue `sales` à partir de la collection `orders`

```
db.createView( "sales", "orders", [
  {
    $lookup:
      {
        from: "inventory", localField: "prodId",
        foreignField: "prodId",
        as: "inventoryDocs"
      }
  },
  {
    $project:
      {
        _id: 0, prodId: 1,
        orderId: 1,
        numPurchased: 1,
        price: "$inventoryDocs.price"
      }
  },
  { $unwind: "$price" }
] )
```

sales

```
{ orderId: 201, prodId: 100, numPurchased: 20, price: 20 },
{ orderId: 202, prodId: 101, numPurchased: 10, price: 10 },
{ orderId: 203, prodId: 102, numPurchased: 5, price: 15 },
{ orderId: 204, prodId: 103, numPurchased: 15, price: 17 },
```

Comparaison SQL et MongoDB

- Une page de la documentation est dédié à la comparaison aux équivalences entre MongoDB et le SQL :

<https://www.mongodb.com/docs/manual/reference/sql-comparison/>

<pre>SELECT * FROM people</pre>	<pre>db.people.find()</pre>
<pre>SELECT id, user_id, status FROM people</pre>	<pre>db.people.find({ }, { user_id: 1, status: 1 })</pre>
<pre>SELECT user_id, status FROM people</pre>	<pre>db.people.find({ }, { user_id: 1, status: 1, _id: 0 })</pre>
<pre>SELECT * FROM people WHERE status = "A"</pre>	<pre>db.people.find({ status: "A" })</pre>
<pre>SELECT user_id, status FROM people WHERE status = "A"</pre>	<pre>db.people.find({ status: "A" }, { user_id: 1, status: 1, _id: 0 })</pre>
<pre>SELECT * FROM people WHERE status != "A"</pre>	<pre>db.people.find({ status: { \$ne: "A" } })</pre>
<pre>SELECT * FROM people WHERE status = "A" AND age = 50</pre>	<pre>db.people.find({ status: "A", age: 50 })</pre>

Exercice

1. Lancer un service `MongoDB` sur Onyxia
2. Insérer les données des deux précédents `csv` dans une collection chacune
3. Mettre en place une dénormalisation des données (dans une nouvelle collection) afin d'éviter la recherche par jointure
4. Effectuer les différentes requêtes d'analyses suivantes :
 - récupérer les données
 - effectuer un comptage pour
 - ...

Optimisation des performances : indexation

Un index permet d'éviter de scanner l'ensemble des documents lors d'une recherche

- ✓ Améliore le temps de lecture
- ✗ Diminue le temps d'écriture

Lister les index

```
db.collection.getIndexes()
```

Supprimer un index

```
db.collection.dropIndex("indexName")
```

Plusieurs types d'index :

- Index par défaut (`_id`)
- Index simple (sur un seul champ)

```
db.collection.createIndex({  
    field: 1  
})
```

- Index composés (plusieurs champs)

```
db.collection.createIndex({  
    field1: 1, field2: -1  
})
```

⚠ Vérifier que votre index tient en RAM

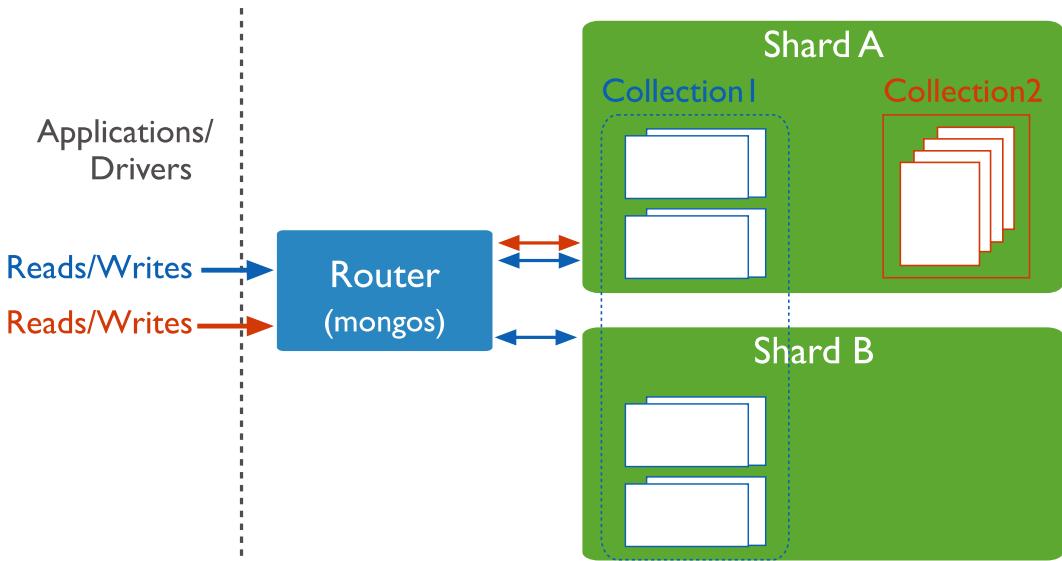
```
db.collection.totalIndexSize()
```

Optimisation des performances : sharding

Principe de partitionnement horizontal

pour distribuer les données à travers plusieurs serveurs

- shards : chaque serveur shard contient une partie des données (chunk).
- mongos : agit comme un routeur de requêtes selon les données et les configurations de préférences



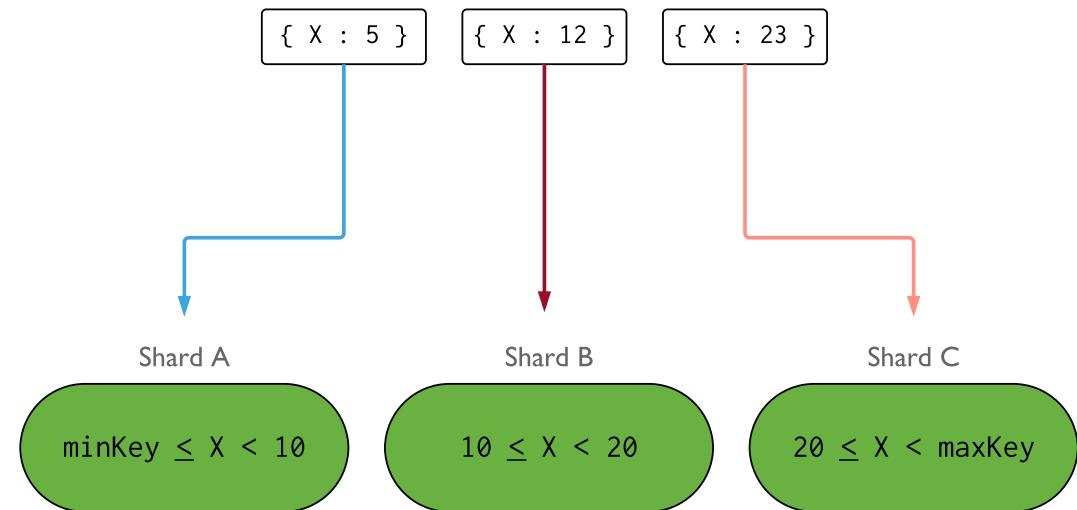
```
sh.enableSharding("mydb")
```

Shard Key : clé de répartition de la donnée

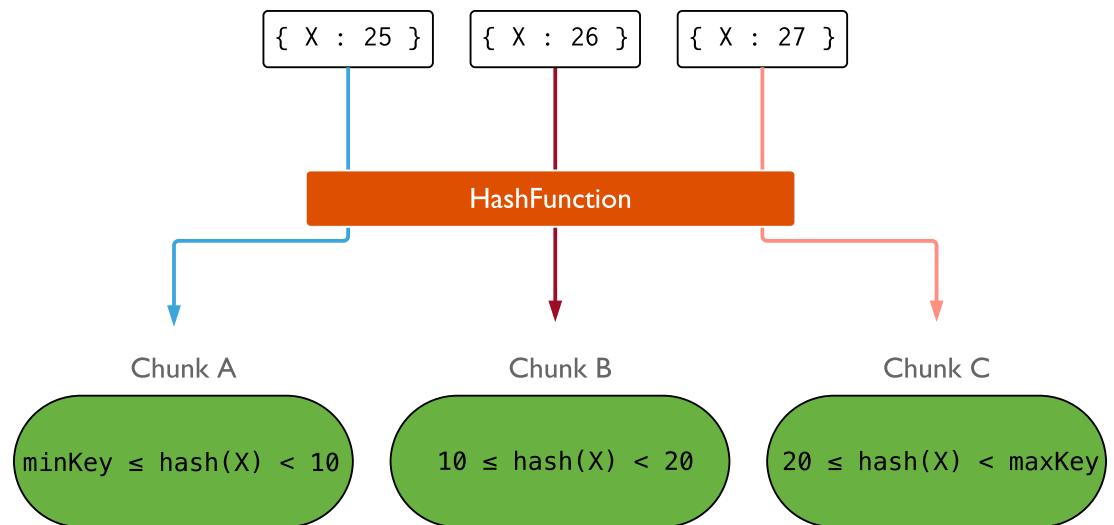
- Cardinalité large (nombre de chunks)
- Fréquence faible (uniformément répartie)

Deux type de *sharding* :

- **Hashed sharding** : fonction appliquée à la clé
 - Accès aléatoires
 - Scalabilité plus simple
- **Ranged sharding** : organisation par plage de valeurs
 - ! déséquilibre possible
 - Idéal pour les accès par plage de valeurs (*logs, transactions*)



```
sh.shardCollection("mydb.collection2", { shardKey: "hashed" })
```



```
sh.shardCollection("mydb.collection", { shardKey: 1 })
```

- **Replica Sets** : ensemble de `shards` qui maintiennent la même données (pour la redondance et disponibilité)
 - Serveur primaire (écriture) et secondaire (lecture)
 - RéPLICATION asynchrone
 - Principe d'élection avec *l'arbiteur* (externe) et *Read Preference*
- **Balancer** : processus qui gère la répartition des `chunks` entre les `shards`

