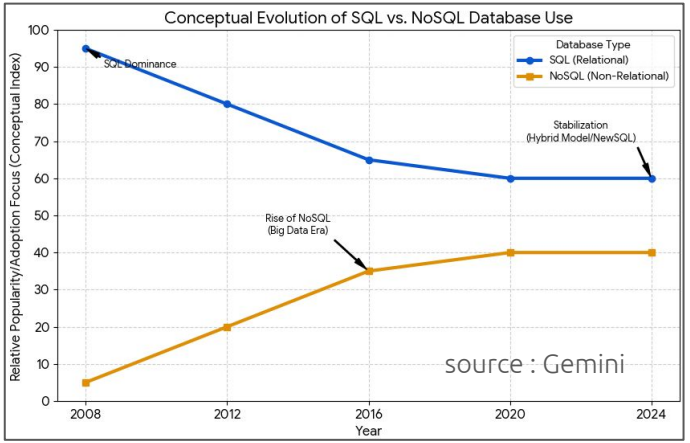
XML et bases de données NoSQL

Ahmed Laatabi
a.laatabi{at}umi.ac.ma
ENSAM - Meknès
2025-2026

NoSQL : bases de données

- **Bases de données NoSQL** (Not **Only** SQL, Not Only Relational) *vs* ***bases de données relationnelles*** (SQL, Relational).
  - Certaines BD NoSQL peuvent aussi supporter SQL à travers des APIs.
- Stocker et extraire de grandes quantités de données, sans utiliser le paradigme *table-association* (lignes, colonnes, clés).
- Structure moins stricte -- *donc plus flexible* -- que celle des SGBDR.
  - SGBDR : le schéma est défini et connu à l'avance.
  - NoSQL : généralement, pas de schéma prédéfini (le schéma peut évoluer dynamiquement avec les données).
- Évolution accélérée par le besoin de gérer de grandes quantités de données "Big Data" distribuées et hétérogènes.
  - Boom des réseaux sociaux, IA, applications web en temps réel (jeux, e-commerce, …).

- NoSQL n'a pas remplacé le modèle relationnel.
- Les SGBDR offrent plus de :
  - ***Intégrité*** : contraintes fortes sur les données (anomalies, doublons).
    - Entreprise, finance, RH, …
  - ***Complexité*** : relations avancées et jointures entre tables.
- Approche hybride :
  - Cohérence du modèle relationnel.
  - Performance de NoSQL.



NoSQL : bases de données

- Les BD NoSQL :
  - Permettent des ***requêtes rapides*** mais ***simples***.
  - Idéales pour un ***stockage non structuré*** (text, images, …).
    - Peuvent contenir des données structurées ou non, dans un même endroit.
  - Schéma flexible qui varie en fonction des données.
  - Rapidité, disponibilité, flexibilité et performance.

- Les propriétés **ACID** (modèle relationnel) d'une **transaction** ne sont pas prioritaires :
  - Atomicité : une transaction est indivisible (exécutée complètement ou pas).
  - Consistance : respect des contraintes d'intégrité après la validation d'une transaction.
  - Isolation : les transactions concurrentes doivent être exécutées séquentiellement.
  - Durabilité : une transaction validée est permanente.

    → Certaines bases de données NoSQL supportent quand même des caractéristiques ACID pour un minimum de consistance, mais généralement : **conformité ACID → SGBD relationnel**.

NoSQL : BASE

- Le modèle **BASE** (***B**asically **A**vailable, **S**oft state, **E**ventually consistent*) suit une philosophie différente :
  - **BA** : l'écriture et la lecture des données sont toujours disponibles, sans garantie de consistance (données non à jour). Les données sont disponibles grâce à la réplication sur plusieurs nœuds de la BD. Un nœud peut livrer une donnée qui n'est pas à jour.
  - **S** : pas de consistance stricte, la valeur d'une donnée peut changer au cours du temps (état temporaire ou transitoire). Le SGBD délègue la consistance à la charge du développeur (traitements externes).
  - **E** : après un certain temps, la consistance des données sera quand même atteinte grâce à la propagation des mises à jour en arrière-plan.

- Les conflits d'écriture sont résolus grâce à des mécanismes de **synchronisation** comme le *Last-write-wins*.
- Les noeuds **échangent des informations** (***gossip***) pour s'accorder sur une même donnée consistante.

→ Les données sensibles ne doivent pas être stockées en NoSQL.

→ Les **données financières** (paiements) nécessitent les propriétés ACID.

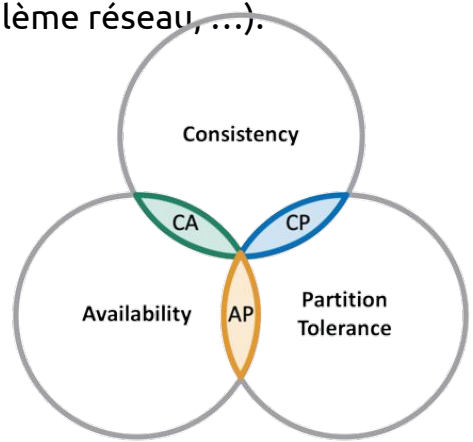
NoSQL : théorème CAP

- Le **théorème CAP** (*théorème de Brewer*) suggère qu'une source de données distribuée ne peut garantir que deux des trois propriétés suivantes :

  Consistency, Availability, and Partition tolerance

- **Consistance (**ou **cohérence**) : toutes les lectures reçoivent la même réponse (aucune donnée obsolète).
- **Disponibilité** : toutes les requêtes reçoivent une réponse (même en cas de panne partielle).
- **Tolérance au partitionnement** : disponibilité malgré des interruptions et des partitionnements du réseau.
  - Deux partitions du système ne peuvent pas communiquer (panne, problème réseau, …).

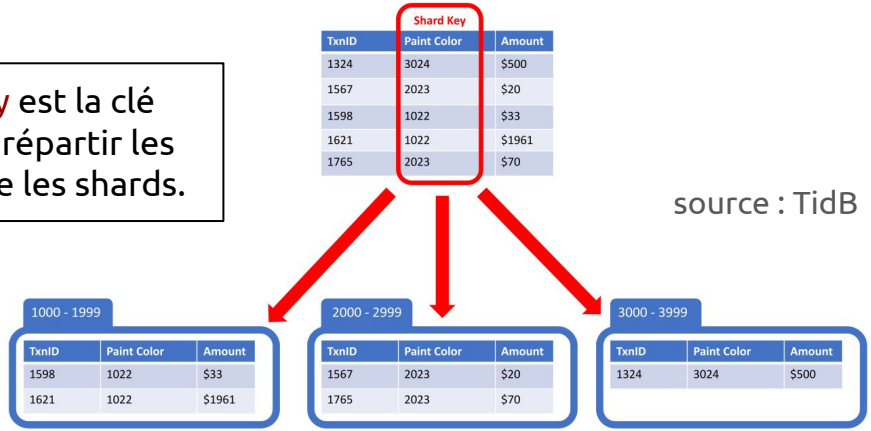
- ACID → CP
- BASE → AP
- Des bases de données **CA** n'existent pas en pratique !
  - Un SGBD distribué doit nécessairement gérer les pannes du réseau.



NoSQL : sharding

- ***Sharding*** (Évolution/scalabilité horizontale) : répartir les données sur plusieurs BD (machines) distantes.
  - *Extensibilité* : de nouveaux serveurs (***shards***) peuvent être ajoutés pour *gérer plus de données*.
  - *Rapidité* : améliore la performance en accélérant les requêtes de traitement des données.
  - *Disponibilité* : réplication des données et tolérance aux pannes sur un shard.
  - *vs* Scalabilité verticale : augmenter la capacité d'une même machine (CPU, RAM, stockage).
- Existe aussi dans le modèle relationnel, mais c'est plus une caractéristique native du modèle NoSQL.
- Sharding automatique dans les SGBD NoSQL lorsque la taille des données croît.

Le shard key est la clé utilisée pour répartir les données entre les shards.



source : TidB

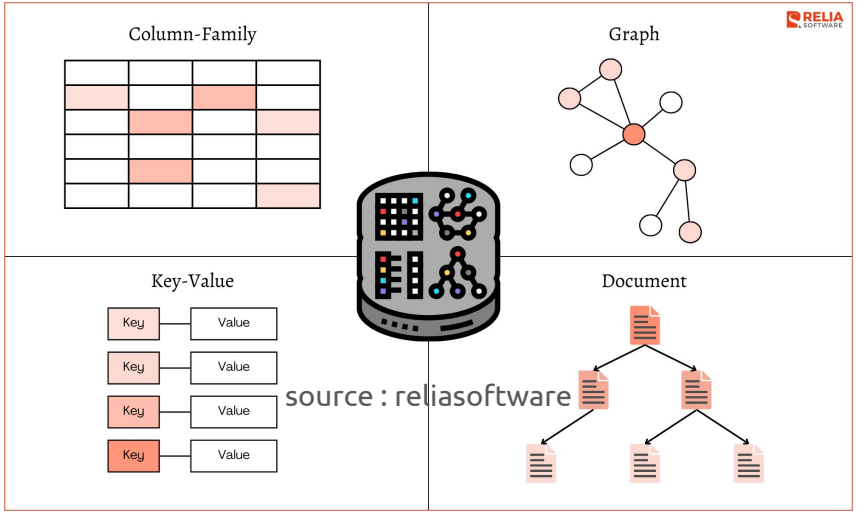
NoSQL : Types de BD

- **Clé-valeur** : les plus simples, stockage sous forme de **collections de paires clé-valeur**.

- **Orientée documents** : stocke les données sous forme de **documents semi-structurés**, généralement au format XML ou JSON.

- **Orientée graphes** : organise les données sous forme de **nœuds connectés**. Les relations entre les nœuds stockent aussi des données. Idéal pour représenter les données des réseaux complexes.

- **Orientée colonnes** : stocke les données dans des tables, ou une ligne peut avoir une ensemble de **colonnes flexibles** regroupées en familles.



NoSQL : Clé-valeur

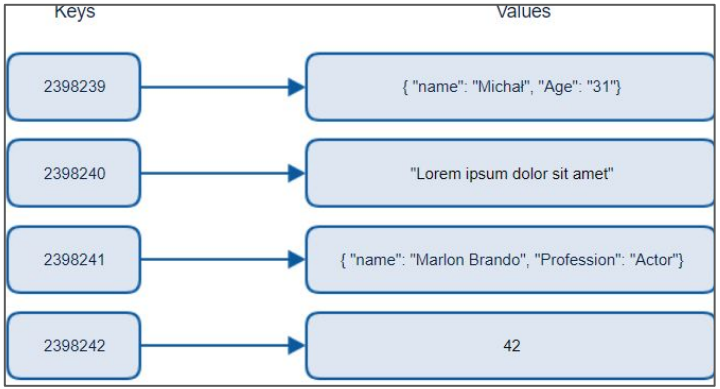
- Stocke les données dans des **collections**. Une clé unique permet d'identifier un enregistrement composé d'une ou plusieurs valeurs.
  - Clé : texte unique.
  - Valeur : nombre, texte, binaire, liste, dictionnaire, …
- Enregistrements avec des structures différentes et des données de types variés : **pas de schéma** → flexibilité.

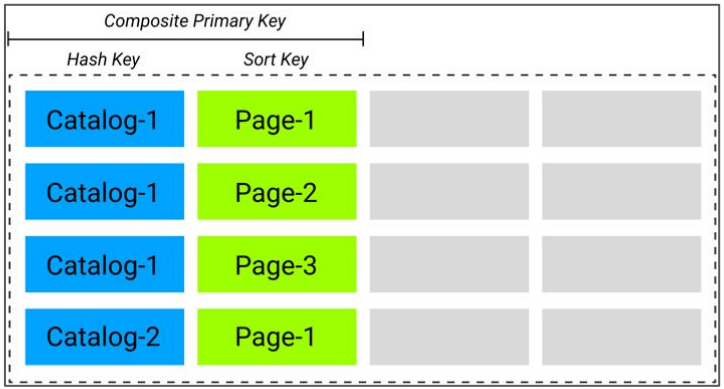
- Les valeurs peuvent être des objets simples (texte, entier, …) ou complexes (tableau, image, objet JSON, …).
- Le SGBD ne s'intéresse pas à la **structure interne** de la valeur : il la stocke comme un bloc opaque, et c'est au développeur de la traiter.
- Pas de jointures : toutes les informations nécessaires sont stockées dans la valeur.

- **Scalabilité horizontale** facile : partitionnement rapide des données entre les noeuds.
- Opérations de base simples : **SET** (clé, valeur), **GET** (clé), **DEL** (clé).

NoSQL : Clé-valeur

- Les clés sont **ordonnées** (alphabétiquement, chronologiquement, par taille de la valeur, …) afin de faciliter les **traitement**s (accès) et le **sharding** (A-D : serveur 1; E-K : serveur 2; …).
- Une clé peut être **composite** avec :
  - Une clé de partition (*hash key*) déterminant le nœud de stockage.
  - Une clé de tri (*sort* ou *range key*) : détermine l'emplacement de la donné dans le nœud.
- Utile pour les **usages en temps réel** : stockage de sessions, données de cache, applications e-commerce, …





NoSQL : Documents

- Les données sont stockées dans des **collections** de **documents** composés d'**imbrications** de **paires clé-valeur**.
- Un document possède un **identifiant unique** et peut contenir des champs différents d'un autre document de la même collection.
- Les documents sont généralement au format **JSON**, **BSON** (JSON binaire), ou **XML**.

- Les applications peuvent créer et manipuler facilement des documents JSON, directement stockables.
- JSON permet de représenter les **objets** manipulés dans les applications POO.
  - Clé-valeur : **{**"mois": 10**}**. // *les clés sont toujours des string*
  - Tableau : **{**"mois": **[**"octobre","novembre"**]}**.
  - Objet : **{**"DateDeNaissance": **{**"Jour": 20,"Mois": "Juin","Année":"2020"**}}**.

- Le documents JSON peuvent **imbriquer** librement des éléments et **évoluer** de manière flexible.
- Des **API**s permettent d'effectuer les opérations de base sur les documents : *création*, *lecture*, *MAJ*, et *suppression* (**CRUD**: *Create*, *Read*, *Update*, *Delete*).

NoSQL : Documents

- Utiles pour la **gestion de contenu**, où chaque entité (produit, catalogue, …) est stockée dans un document.
- Chaque document peut **évoluer en structure** indépendamment des autres.

Key	Document
101	{ "ID": "1001", "ItemsOrdered":[{ "ItemID": "1", "Quantity": "2", "cost": "1000", }, { "ItemID": "1001", "Quantity": "2", "cost": "1000", }], "OrderDate": "05/11/2019" }
102	"ID": "1002", "ItemsOrdered":[{ "ItemID": "2890", "Quantity": "11", "cost": "10000", erDate": "05/11/2019"

Relational

ID	first_name	last_name	cell	city	year_of_birth	location_x	location_y
1	'Mary'	'Jones'	'516-555-2048'	'Long Island'	1986	'-73.9876'	'40.7574'

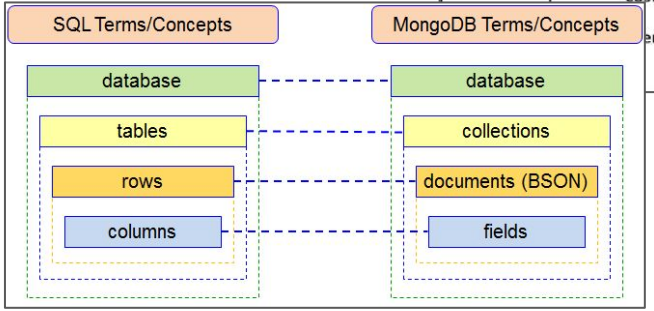
ID	user_id	profession
10	1	'Developer'
11	1	'Engineer'

ID	user_id	name	version
20	1	'MyApp'	1.0.4
21	1	'DocFinder'	2.5.7

ID	user_id	make	year
30	1	'Bentley'	1973
31	1	'Rolls Royce'	1965

MongoDB

```
first_name: "Mary",
last_name: "Jones",
cell: "516-555-2048",
city: "Long Island",
year_of_birth: 1986,
location: {
        type: "Point",
        coordinates: [-73.9876, 40.7574]
},
profession: ["Developer", "Engineer"],
apps: [
 { name: "MyApp",
   version: 1.0.4 },
 { name: "DocFinder",
   version: 2.5.7 }
],
cars: [
  { make: "Bentley",
    year: 1973 },
  { make: "Rolls Royce",
    year: 1965 }
]
```



NoSQL : JSON

- **JSON** (J*avaScript Object Notation*) : format texte dérivé de la syntaxe des objets *JavaScript*.
- Comparable à **XML** : lisible, hiérarchique et facile à analyser (parser).
  - JSON est plus léger que XML et prend en charge les listes (tableaux).
  - Pas de commentaires en JSON.

- Paires de clé-valeurs → **"**Nom**"**:**"mon_nom"**

- Objet JSON **{}** peut contenir plusieurs paires → {"Nom":"**mon_nom**", "prenom":**null**, "age":**40**}
- Liste JSON **[]** peut contenir plusieurs objets :

  {"etudiants": [{"Nom":"**mon_nom**", "age":**40**}, {"Nom":"**my_name**", "age":**21**}]}

- Types : string, nombre, booléen, liste, objet JSON, null.
- Un document JSON doit avoir une racine unique : un objet **{}** ou un tableau **[]**.

NoSQL : SQL

On souhaite modéliser un réseau d'amis de type **Facebook** dans une base de données relationnelle.
Chaque utilisateur possède :

- un identifiant unique,
- un nom,
- un prénom,
- une date de naissance.

On veut aussi stocker le type et la date de début de chaque amitié.

On désire connaître par exemple :

- le nombre d'amis en commun pour chaque paire d'amis.
- la liste des amis d'amis d'un utilisateur.

→ Proposer un schéma relationnel pour représenter ces informations.
→ Expliquer les limites de ce modèle dans un contexte de grande échelle (réseau social réel).

NoSQL : SQL

- Le nombre d'amis en commun entre l'utilisateur 173 et l'utilisateur 1991 :

```
SELECT COUNT(*) AS nb_amis_communs
FROM Amitie a1
JOIN Amitie a2 ON a1.id_user2 = a2.id_user2
WHERE a1.id_user1 = 173 AND a2.id_user1 = 1991;
```

```
SELECT COUNT(*) AS nb_amis_communs
FROM Amitie
WHERE id_user1 = 173  AND id_user2 IN
(SELECT id_user2 FROM Amitie WHERE id_user1 = 1991);
```

- La liste des amis d'amis de l'utilisateur 173 :

```
SELECT DISTINCT u2.*
FROM Amitie a1
JOIN Amitie a2 ON a1.id_user2 = a2.id_user1
JOIN Utilisateur u2 ON u2.id_user = a2.id_user2
WHERE a1.id_user1 = 173 AND u2.id_user <> 173;
```

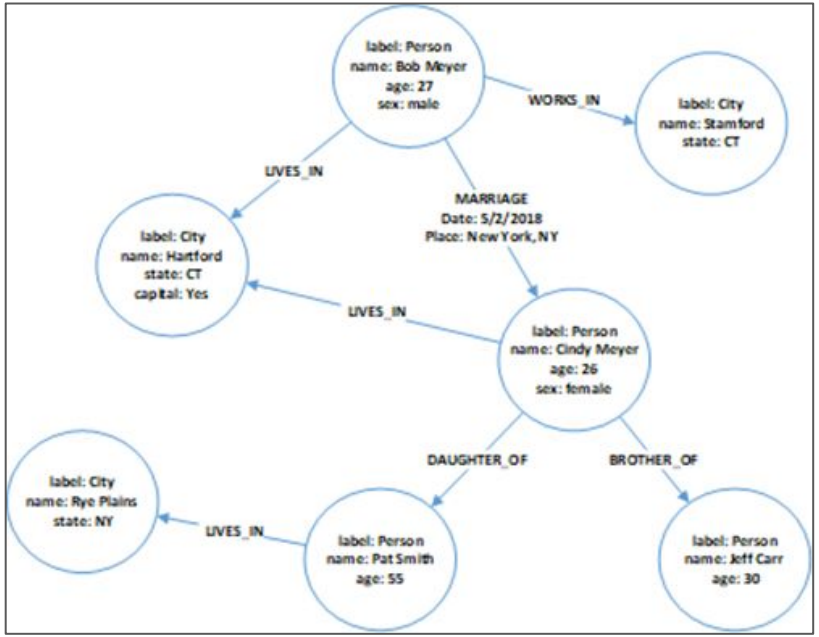
Requêtes lentes et difficiles à lire et à expliquer.
Dans cet exemple, on stocke une amitié deux fois → cela simplifie les requêtes mais augmente l'espace de stockage.

NoSQL : Graphes

- Utiles pour le stockage de **données connectées** en utilisant un **réseau** de **nœuds** et de **relations**.
- Les nœuds ont des *labels* spécifiant leur type et peuvent contenir un grand nombre de **propriétés**.
- Le type, la direction, et la force du lien entre deux nœuds sont également des données.
- La BD est manipulée en naviguant dans le graphe.

- Le schéma et la structure peuvent évoluer.
- Des langages de requêtes et des algorithmes de graphes permettent de manipuler efficacement les données.

- Les BD orientées graphes sont difficilement scalables à cause de l'interconnexion des données.

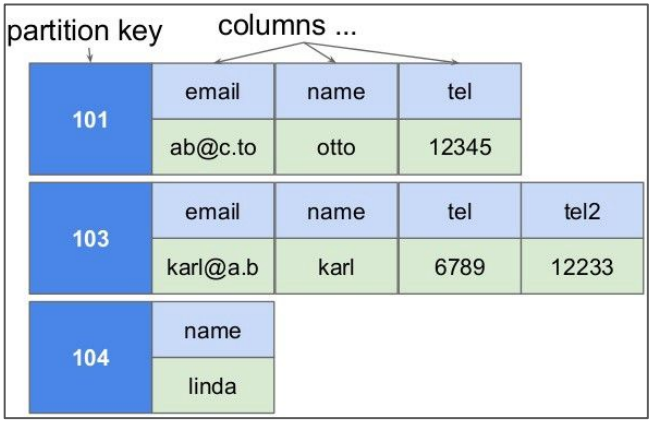


NoSQL : Colonnes larges

- Les BD en **colonnes larges** se concentrent sur la notion de **colonnes** (les BD relationnelles : *les lignes*).
- Le schéma n'est pas fixe :
  - De nouvelles colonnes peuvent être ajoutées dynamiquement.
  - Les lignes peuvent comporter un nombre, un type, et des noms de colonnes différents.

- Le regroupement de colonnes (famille de colonnes) permet d'optimiser les requêtes en ne chargeant que les colonnes nécessaires : les colonnes en famille sont stockées ensemble.
- La structure en colonnes : plus de **flexibilité** et de **scalabilité**.

- Idéales pour les **requêtes analytiques** :
  - utilisant des agrégations et des filtres.
  - analysant de grandes quantités de données.
- Les BD relationnelles sont plus adaptées aux requêtes **transactionnelles normalisées**.



NoSQL : au-delà des limites du relationnel

1. **Scalabilité horizontale**:
   - Le relationnel est plus adapté à la scalabilité verticale.
   - Les BDR réparties (*réplication*, *fragmentation horizontale* ou *verticale*) souffrent de problèmes de synchronisation pour maintenir la consistance et la cohérence (**ACID**, **2PC** : *Two Phase Commit*) et de latence pour reconstruire la requête d'origine (UNION, JOIN) de façon transparente.
   - Le NoSQL adopte une consistance éventuelle → meilleure scalabilité horizontale.
   - Le NoSQL utilise le *sharding* (sans coordination globale) → meilleure scalabilité horizontale.

2. **Rigidité du schéma** : modifier le schéma dans le relationnel est complexe. Le NoSQL n'a pas de schéma prédéfini (ou schéma flexible), et s'adapte donc aux données.

3. **Données hiérarchiques ou non structurées** : ces types de données sont compliqués à stocker dans le relationnel. Le NoSQL (JSON, graphes) est plus adapté aux données imbriquées ou connectées.

4. **Tolérance aux pannes** : le relationnel distribué préfère rendre les données indisponibles pour préserver la cohérence (CP). Le NoSQL est plutôt AP.

NoSQL : Redis

- **Redis** (***Re**mote **Di**ctionary **S**erver*) est une BD NoSQL en mémoire, de type clé-valeur.
- Utilisé principalement pour le ***temps réel*** et le ***caching*** : accès très rapide en lecture et écriture.
- Options de durabilité (cache) et de persistance sur disque (sauvegarder et restaurer).

- **Redis** permet de :
  - trier et indexer automatiquement certaines données pour accélérer les accès.
  - traiter des requêtes regroupées pour améliorer la rapidité et réduire la charge réseau.
  - échanger des messages entre applications qui produisent (*publishers*) et qui consomment (*subscribers*) à travers des canaux de messagerie.
  - servir de cache pour des BD sur disque, par exemple *MongoDB*, afin de réduire les temps de lecture.

- Étant entièrement en mémoire, **Redis** peut ne pas être adaptée au **Big Data** si la mémoire est limitée ou en l'absence de sharding.

NoSQL : Redis

- Créer un RediSearch index : FT.CREATE idx:bicycle ON JSON …
- Ajouter les éléments JSON : JSON.SET "bicycle:0" "." {...}.

- (SQL) **SELECT** * FROM bicycles **WHERE** price >= 1000
- (Redis) **FT.SEARCH** idx:bicycle "@price:[1000 +inf]"

- (SQL) **SELECT** id, price **FROM** bicycles
- (Redis) **FT.SEARCH** idx:bicycle "*" RETURN 2 __key, price

- (SQL) **SELECT** id, price-price*0.1 **AS** discounted **FROM** bicycles
- (Redis) **FT.AGGREGATE** idx:bicycle "*" **LOAD** 2 __key price **APPLY** "@price-@price*0.1" **AS** discounted

- (SQL) **SELECT** condition, **AVG**(price) **AS** avg_price **FROM** bicycles **GROUP BY** condition
- (Redis) **FT.AGGREGATE** idx:bicycle "*" **GROUPBY** 1 @condition **REDUCE AVG** 1 @price **AS** avg_price


```
{
  "bicycle:0": {
    "pickup_zone": "POLYGON((-74.0610 40.7578, -73.9510 40.7578, -73.9510 40.6678,
      -74.0610 40.6678, -74.0610 40.7578))",
    "store_location": "-74.0060,40.7128",
    "brand": "Velorim",
    "model": "Jigger",
    "price": 270,
    "description": "Small and powerful, the Jigger is the best ride for the
      smallest of tikes! This is the tiniest kids' pedal bike on the market
      available without a coaster brake, the Jigger is the vehicle of choice for
      the rare tenacious little rider raring to go.",
    "condition": "new"
  },
  "bicycle:1": {
    "pickup_zone": "POLYGON((-118.2887 34.0972, -118.1987 34.0972, -118.1987
      33.9872, -118.2887 33.9872, -118.2887 34.0972))",
    "store_location": "-118.2437,34.0522",
    "brand": "Bicyk",
    "model": "Hillcraft",
    "price": 1200,
    "description": "Kids want to ride with as little weight as possible.
      Especially on an incline! They may be at the age when a 27.5\" wheel bike is
      just too clumsy coming off a 24\" bike. The Hillcraft 26 is just the
      solution they need!",
    "condition": "used"
  }
}
```

NoSQL : Redis

Python offre une librairie *redis-py* qui permet d'interagir avec **Redis**.

r = redis.Redis(host='localhost',port=6379,db=0)

- r.set("cle1", "première valeur")
- valeur1 = r.get("cle1")
- r.delete("cle1")

r.ft("idx:bicycle").create_index(fields=[....], prefix=["bicycle:"])

- req1 = r.ft("idx:bicycle").search("@price:[1000 +inf]")
- req2 = r.ft("idx:bicycle").search("* RETURN 2 __key price")
- req3 = r.ft("idx:bicycle").aggregate("*").load("__key", "price").apply("@price-@price*0.1", "discounted")
- req4 = r.ft("idx:bicycle").aggregate("*").group_by("@condition",reducers={"avg_price": ("AVG", "@price")})

NoSQL/SQL : Index

CREATE INDEX nom_index **ON** nom_table (NomColonne1, NomColonne2, ...);

- Un **index** est une structure de données créée sur une ou plusieurs colonnes (triées) d'une table.
  - L'indexation est aussi appliquée à des clés ou à des champs JSON.

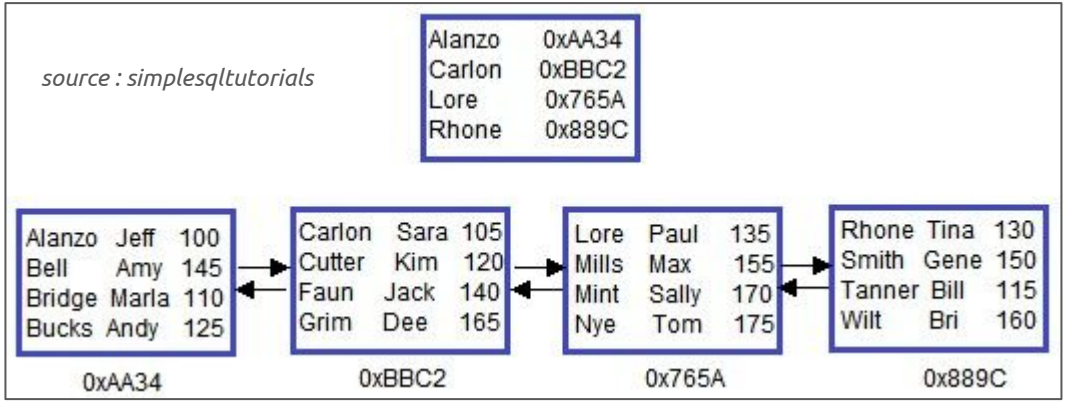
- Il permet d'optimiser les requêtes (WHERE, GROUP BY, ORDER BY):
  - Accélérer les recherches et les tris : localiser rapidement les données dans la table sans la parcourir entièrement (éviter le *full table scan* ou *sequential scan*).
  - L'**index d'un livre** (le catalogue) : sur quelles pages (lignes) se trouve le terme (la donnée) **X**.

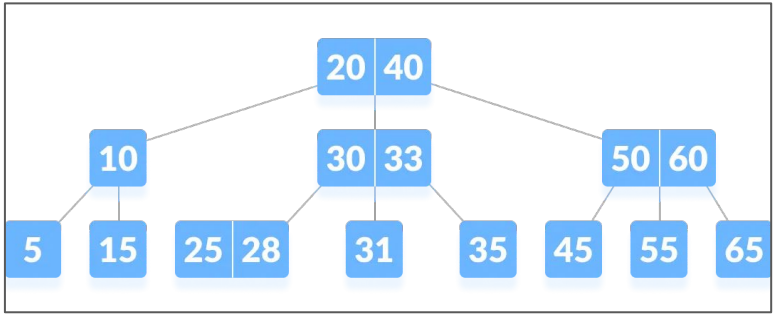
- Les SGBD créent automatiquement un index sur toute clé primaire.
- L'index ralentit les opérations d'écriture : il doit être mis-à-jour à chaque modification des données.
- Les index consomment l'**espace de stockage**.
  - L'index créé une autre structure de données (le **B-Tree**) contenant les colonnes indexées.

NoSQL/SQL : B-Tree

- **Balanced-Tree** : arbre de recherche équilibré, où chaque noeud peut contenir plusieurs clés ordonnées et des pointeurs vers les sous-arbres (**arbre moins profond**).
  - Les données stockées dans les feuilles sont triées en ordre croissant.
  - Tous les chemins de la racine aux feuilles ont la même longueur (feuilles au même niveau).
  - Opérations en $O(\log n)$: accès rapide qui minimise les accès disque.
  - Les feuilles sont en liste chaînée (simple ou double) pour des parcours ordonnés sans remonter.

- **Etudiant** (Nom, Prenom, ID)
- On crée un index sur la colonne “Nom”.





NoSQL : MongoDB

- **MongoDB** est une BD open-source basée sur les documents au format **JSON-like**, stockés en BSON (*Binary JSON*).
  - BSON est plus rapide à lire et à écrire.
  - BSON Prend en charge des types supplémentaires : ObjectId, Date, Binary, …

- Un enregistrement dans MongoDB est appelé **document**. Les documents sont regroupés en **collections**.
- Pour chaque document, MongoDB crée automatiquement un champ "**_id**" de type **ObjectId** si aucun champs "_id" n'est spécifié.

- Le serveur MongoDB :
  - Se charge de la traduction JSON/BSON à chaque interaction avec l'utilisateur.
  - Peut être installé localement (***Community Server***) ou utilisé via le service cloud ***MongoDB Atlas***.
- **MongoDB Compass** offre une interface graphique pour interagir avec le serveur.
- On peut également utiliser l'invite de commandes ***mongosh***.

NoSQL : MongoDB


```
{
  "bicycle:0": {
    "pickup_zone": "POLYGON((-74.0610 40.7578, -73.9510 40.7578, -73.9510 40.6678,
      -74.0610 40.6678, -74.0610 40.7578))",
    "store_location": "-74.0060,40.7128",
    "brand": "Velorim",
    "model": "Jigger",
    "price": 270,
    "description": "Small and powerful, the Jigger is the best ride for the
      smallest of tikes! This is the tiniest kids' pedal bike on the market
      available without a coaster brake, the Jigger is the vehicle of choice for
      the rare tenacious little rider raring to go.",
    "condition": "new"
  },
  "bicycle:1": {
    "pickup_zone": "POLYGON((-118.2887 34.0972, -118.1987 34.0972, -118.1987
      33.9872, -118.2887 33.9872, -118.2887 34.0972))",
    "store_location": "-118.2437,34.0522",
    "brand": "Bicyk",
    "model": "Hillcraft",
    "price": 1200,
    "description": "Kids want to ride with as little weight as possible.
      Especially on an incline! They may be at the age when a 27.5\" wheel bike is
      just too clumsy coming off a 24\" bike. The Hillcraft 26 is just the
      solution they need!",
    "condition": "used"
  }
}
```

NoSQL : MongoDB

- SELECT * FROM bicycles; ⇔ db.bicycles.find()
- SELECT * FROM bicycles LIMIT 5; ⇔ db.bicycles.find().limit(5) ⇔ db.bicycles.aggregate([{"$limit": 5}])
- SELECT * FROM bicycles WHERE price > 500; ⇔ db.bicycles.find({ price: { $gt: 500 } })
- SELECT COUNT(*) FROM bicycles; ⇔
  db.bicycles.countDocuments() ⇔ db.bicycles.aggregate([{ $count: "total" }])
- SELECT condition, AVG(price) FROM bicycles GROUP BY condition; ⇔
  db.bicycles.aggregate([{ $group: { _id: "$condition", avgPrice: { $avg: "$price" } } }])
- SELECT * FROM bicycles ORDER BY price DESC; ⇔ db.bicycles.find().sort({ price: -1 })
- INSERT INTO bicycles (brand, price, condition) VALUES ('Vélo', 100, 'old'); ⇔
  db.bicycles.insertOne({ brand: "Vélo", price: 100, condition: "old" })
- UPDATE bicycles SET price = 150 WHERE brand = 'Vélo'; ⇔
  db.bicycles.updateOne({ brand: "Vélo" }, { $set: { price: 150 } })
- DELETE FROM bicycles WHERE price < 200; ⇔ db.bicycles.deleteMany({ price: { $lt: 200 } })
- SELECT brand, COUNT(*) FROM bicycles GROUP BY brand HAVING COUNT(*) > 2; ⇔
  db.bicycles.aggregate([{ $group: { _id: "$brand", total: { $sum: 1 } } }, { $match: { total: { $gt: 2 } } }])

NoSQL : MongoDB

- use transports
- **db**.createCollection("bicycles")
- **db**.**bicycles**.insertOne({...})
- **db**.**bicycles**.insertMany([{...}])

- **db**["**bicycles**"].**find**({"condition" : "new"}) → **db**.**bicycles**.**find**({"condition" : "new"})
- **db**.**bicycles**.**findOne**({"condition" : "new"})
- **db**.**bicycles**.**find**({"condition": "new", "model": "Secto" })
- **db**.**bicycles**.**find**({"condition" : "new"},{"id_":1}) *// le "id_" est inclus par défaut dans find, l'exclure avec :0*
- **db**.**bicycles**.**find**({}, {"_id":0, "condition":1}) *// exclure "id_" et inclure "condition" seulement*
- **db**.**bicycles**.**find**({}, {"_id":0, "condition":0}) *// exclure "id_" et "condition" et inclure tout le reste*
- **db**.**bicycles**.**find**({"condition" : "new"},{"_id":1}).**sort**({"price":1})

- Opérateurs logiques : $and, $or, $nor, $not.
- Opérateurs de comparaison : $eq, $ne, $gt, $gte, $lt, $lte, $in (une valeur dans une liste), $nin.

NoSQL : MongoDB

- **db**.**bicycles**.**updateOne**({"condition": "refurbished"},{"**$set**": {"price": 815}})
- **db**.**bicycles**.**updateOne**({"brand": "Vélo"},{
  "**$set**": {"brand": "Vélo", "model": "Bicyclette", "price": 100, "condition": "old"}
  }, {**upsert**: true})
- **db**.**bicycles**.**updateMany**({"condition": "refurbished"},{"**$set**": {"price": 815}})
- **db**.**bicycles**.**updateMany**({"condition": "refurbished"},{"**$inc**": {"price": 5}})
- **db**.**bicycles**.**updateOne**({"condition": "refurbished"},{"**$rename**": {"price": "prix"}})
- **db**.**bicycles**.**updateMany**({"condition": "used"},{"**$unset**": {"condition": ""}})

- **db**.**bicycles**.**deleteOne**({"condition": "old"})
- **db**.**bicycles**.**deleteMany**({"condition": "old"})

- **db**.**bicycles**.**find**({"condition" : "new"}) → **db**.**bicycles**.**find**({"condition" : {"$eq":"new"}})
- **db**.**bicycles**.**find**({"condition" : {"$ne":"new"}})

NoSQL : MongoDB

- **db**.**bicycles**.**find**({"condition" : {"$nin": ["new","used"]}}, {"brand":1})
- **db**.**bicycles**.**find**({"condition" : {"$in": ["new"]}, "price": {"$gt": 500} },
  {"condition":1, "price": 1})

  ⇔

- **db**.**bicycles**.**find**({ "$and":
  [{ "condition" : {"$in": ["new"]} } , { "price": {"$gt": 500} }]
  }, {"condition":1, "price": 1})

- **db**.**bicycles**.**find**({ "$nor":
  [{ "condition" : {"$in": ["new", "used"]} } , { "price": {"$lt": 500} }]
  }, {"condition":1, "price": 1})
- **db**.**bicycles**.**find**({ "$and" : [
  { "$or": [{"condition" : "new"}, {"condition" : "used"}] },
  { "price": {"$lt": 500} }
  ] }, {"condition":1, "price": 1})

Simplifier les requêtes ?

NoSQL : MongoDB

- L'**agrégation** permet de transformer les documents d'une collection à travers un **ensemble** (liste **[]**) d'**étapes** (***pipeline***) pour analyser et produire des données filtrées, regroupées, et reformulées.
- Les étapes de l'agrégation sont ordonnées, et chacune travaille sur les données issues (*outputs*) de l' étape précédente.
- **Récupérer les documents (filtre)** :

```
db.bicycles.aggregate ([{"$match": {"price":{"$gt": 500}}}])
```

// Plus léger : si l'on a pas besoin d'un pipeline d'agrégation, il est préférable d'utiliser find()

⇔ **db.bicycles.find**({"price": {"$gt":500}})

- **Réaliser des regroupements par catégories** :

```
db.bicycles.aggregate ([
{ "$match": { "price":{"$gt": 500} } },
{ "$group": { _id : "$condition" , "prix_moyen": {"$avg" : "$price"} } }
])
```

- À ne pas confondre **_id** (l'**identifiant** d'un document) et **_id** (la **clé du regroupement** dans $group).

NoSQL : MongoDB

- **$match** utilise une logique de requêtes classiques (comme **find**) : les noms de champs n'ont pas besoin de **$**. Après l'opérateur **$group**, il faut utiliser **$** devant les noms des champs.

- **db**.bicycles.**aggregate([** { "**$match**": { "price": { "$gt": 500 } } },
  { "**$group**": { _id: "$condition", "prix_total": {"$sum" : "$price" } } } **])**

- **db**.bicycles.**aggregate([** { "**$group**": { _id: null, "prix_total": {"$sum" : "$price" } } } **])**

- Pour compter le nombre d'éléments par groupe, il ne faut **pas** utiliser $count :
  db.bicycles.**aggregate([** { "**$match**": { "price": { "$gt": 500 } } },
  { "**$group**": { _id: "$condition", "nombre": { "$sum": **1** } } } **])**

- L'opérateur $count (ne prend pas d'arguments) est la dernière étape du pipeline, utilisée pour compter le nombre total de documents en sortie d'une requête :
  db.bicycles.**aggregate([** { "**$match**": { "price": { "$gt": 500 } } },
  { "**$count**": "total" } **])**

NoSQL : MongoDB

- **db**.bicycles.**aggregate** ([**{** "**$match**": { "price":{"$gt": 500} } **}**,
  { "**$group**": { _id : "**$**condition" } **}**]) *// lister tous les valeurs distinctes du champ **condition***

- **db**.bicycles.**aggregate** ([**{** "**$match**": { "price":{"$gt": 500} } **}**,
  { "**$group**": { _id : ["**$**condition","$model"] } **}**])

- **db**.bicycles.**aggregate** ([**{** "**$match**": { "price":{"$gt": 500} } **}**,
  { "**$group**": { _id : ["**$**condition","$model"]} **}**,
  {"**$limit**": 3**}**])

- **db**.bicycles.**aggregate** ([**{** "**$match**": { "price":{"$gt": 500} } **}**,
  { "**$project**": { "condition": 1, "price": 1 } **}**]) *// même projection que dans find()*

  ⇔

  db.bicycles.**find**({"price" : {"$gt" : 500}}, {"condition" : 1, "price" : 1})

NoSQL : MongoDB

- **db**.**bicycles**.**aggregate** ([{ "**$project**": { "condition": 1, "price": 1 } }])
- **db**.**bicycles**.**aggregate** ([{ "**$match**": { "price":{"$gt": 500} } },
  { "**$sort**": { "price": -1 } } , { "**$project**": {"price": 1} }])
- **db**.**bicycles**.**aggregate** ([{ "**$match**": { "price":{"$gt": 500} } },
  { "**$group**": {_id: "$condition", "prix_moyen": {"$avg" : "**$**price" } } },
  { "**$sort**": { "prix_moyen": -1 } } , { "**$project**": {"prix_moyen": 1} } ,
  { "**$limit**": 2 }])
- **db**.**bicycles**.**aggregate** ([{ "**$addFields**": { "my_cond": "$condition"} }])

→ L'**ordre des opérations est important**, car chaque étape agit uniquement sur les documents produits par l'étape précédente. Placer un filtre (**$match**) au début du pipeline est donc recommandé pour réduire le nombre de documents (si c'est possible !) sur lesquels les étapes suivantes vont opérer.

NoSQL : MongoDB

- L'opérateur **$lookup** permet de faire une **jointure** de type "*left outer*" avec une autre collection de la même base de données.

- **from** : nom de l'autre collection.
- **localField** et **foreignField** : champs de la jointure.
- **as** : nom du champ qui contiendra **le** ou **les** documents correspondants de l'autre collection.

- **db**.**bicycles**.**aggregate** ([{"**$lookup**": {
  from: "conditions",
  localField: "condition", **foreignField**: "condition",
  as: "la_condition"
  } }])

NoSQL : MongoDB

- **$lookup** renvoie le résultat sous forme de tableau (liste).
- Si un seul résultat est attendu, **$unwind** permet de décomposer le tableau et récupérer directement l'objet.

```
db.bicycles.aggregate ([ {"$lookup": {
        from: "conditions", localField: "condition", foreignField: "condition", as: "la_condition" } },
            { "$unwind": "$la_condition" }, {"$limit": 1}  ])
```

```
db.bicycles.aggregate ([ {"$lookup": {
        from: "conditions", localField: "condition", foreignField: "condition", as: "la_condition" } },
                    { "$unwind": "$la_condition" }, {"$project" : {"la_condition.description" : 1}  ])
```

- Plusieurs opérations **$lookup** peuvent être combinées entre elles pour réaliser des jointures entre plusieurs collections.
- **$lookup** peut être combinée également avec les autres opérations du pipeline d'agrégation comme $match, $project, $group, et $sort.