

# UE BIGDATA

# FISE A3 - TAF LOGIN

# INTRODUCTION



**IMT Atlantique**  
Bretagne-Pays de la Loire  
École Mines-Télécom

# DÉFINITION ?

Accueil > Grand dictionnaire terminologique

## mégadonnées

---

**Domaines :**

Informatique

Intelligence artificielle

— anglais : big data

**Auteur :**

 Office québécois de la langue française

**Dernière mise à jour :** 2020

---

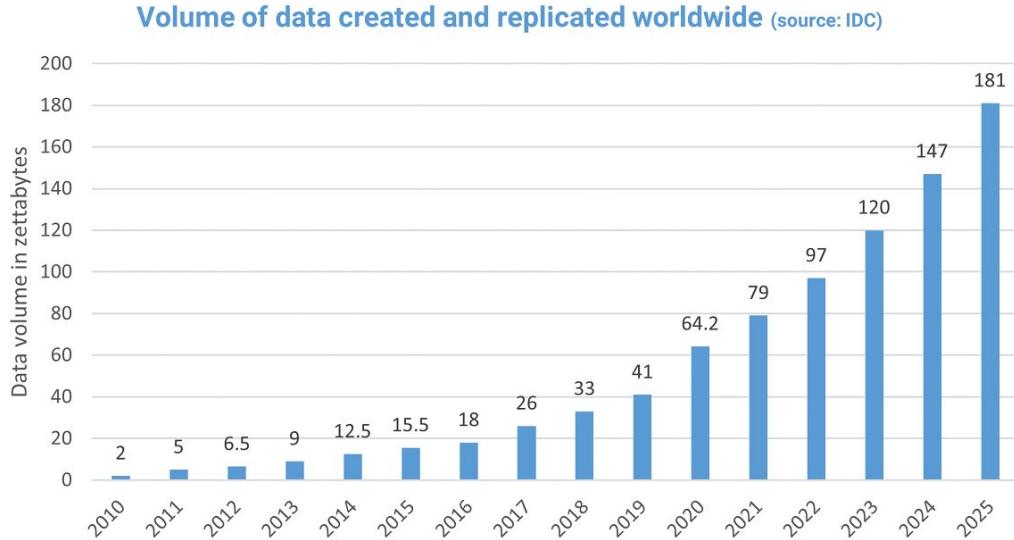
### Définition

Ensemble d'une très grande quantité de données, structurées ou non, se présentant sous différents formats et en provenance de sources multiples, qui sont collectées, stockées, traitées et analysées dans de courts délais, et qui sont impossibles à gérer avec des outils classiques de gestion de bases de données ou de gestion de l'information.

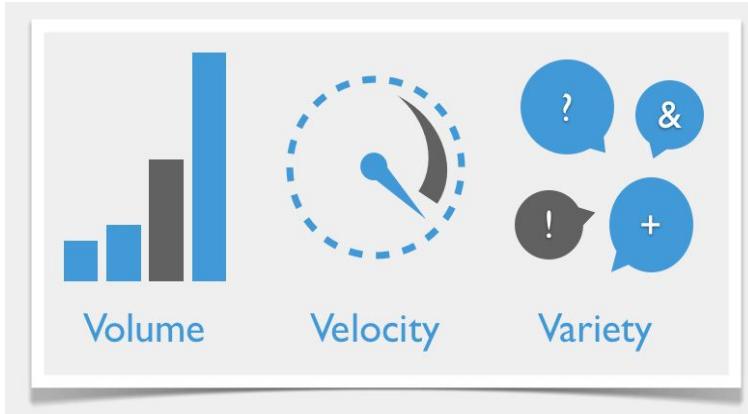
# LES ORDRES DE GRANDEUR POUR LE STOCKAGE

<b>bit (b)</b>	0 ou 1		1 mm
<b>octet (o) / byte (B)</b>	8 bits	1 caractère ASCII	1 m
<b>kilooctet (ko / KB)</b>	1000 octets, $10^3$ octets	Une page de texte (2 Ko)	1 km
<b>mégaoctet (Mo/MB)</b>	1000 ko, $10^6$ octets	Tout Shakespeare (5 Mo). Un morceau de pop (4 Mo).	$10^6$ m - Lille-Marseille
<b>gigaoctet (Go/GB)</b>	1000 Mo, $10^9$ octets	Un film (1-2 Go)	$10^9$ m - Terre-Lune (0,38)
<b>téraoctet (To/TB)</b>	1000 Go, $10^{12}$ octets	Tous les livres de la American's Library of Congress (15 To)	$10^{12}$ m - Terre-Saturne (1,3)
<b>pétaoctet (Po/PB)</b>	1000 To, $10^{15}$ octets	Google traite 1 Po en 1 heure	$10^{15}$ m - année-lumière (9,5)
<b>exaoctet (Eo/PB)</b>	1000 Po, $10^{18}$ octets	10 milliards d'exemplaires de The Economist	$10^{18}$ m - plus proche trou noir
<b>zettaoctet (Zo/ZB)</b>	1000 Eo, $10^{21}$ octets	Information présente sur Internet en 2024: 174 Zo	$10^{21}$ m
<b>yottaoctet (Yo/YB)</b>	1000 Zo, $10^{24}$ octets	Currently too big to imagine	$10^{26}$ m - la taille de l'univers

# TAILLE DES DONNÉES



# LES 3 V DU BIGDATA



- **Volume**

- La masse de données générées est de plus en plus grande

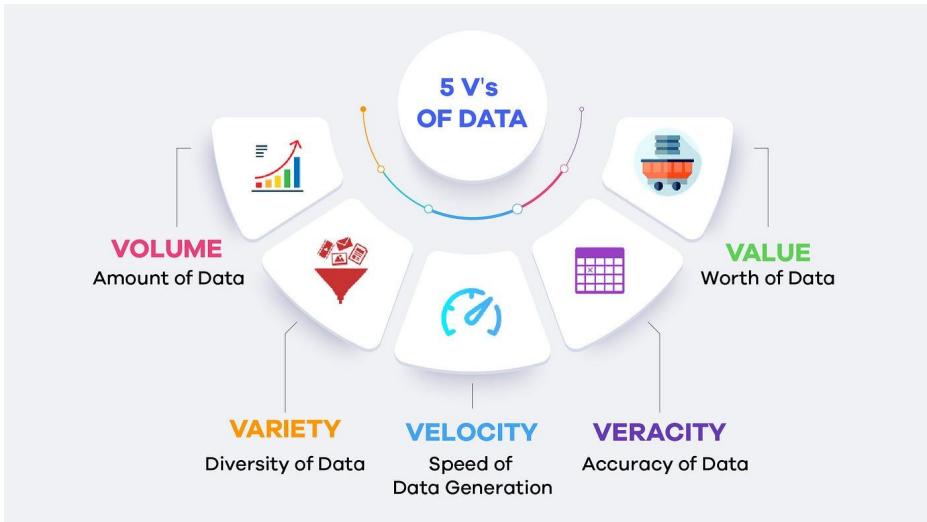
- **Vélocité**

- La fréquence à laquelle sont produites les données est de plus en plus grande

- **Variété**

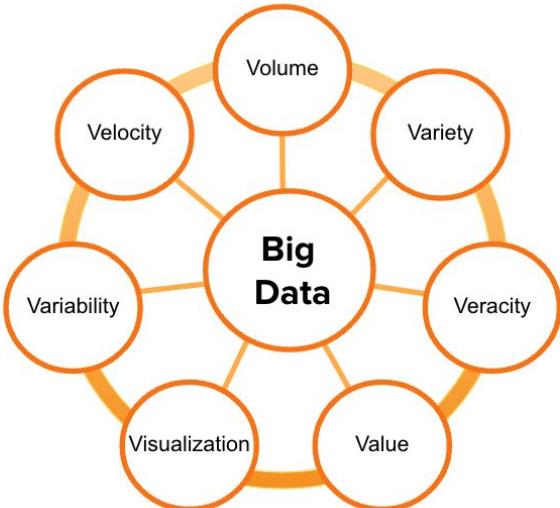
- une grande variété de données collectées

# LES 5 V DU BIGDATA



- Volume
- Vélocité
- Variété
- **Véracité**
  - Lié à la qualité de l'information, à son intégrité, à la fiabilité de la source
- **Valeur**
  - Les données brutes ont souvent peu d'intérêt, il faut réussir à créer de la valeur à partir des données

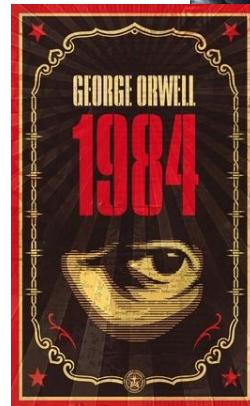
# LES 7 V DU BIGDATA



- Volume
- Vélocité
- Variété
- Véracité
- Valeur
- **Visualisation**
  - Rendre l'information exploitable par le plus grand nombre
- **Variabilité**
  - Nature changeante de la donnée dont le format ou la valeur peut varier avec le temps

# LES CÔTÉS OBSCURS

- Consommation de ressources et production de gaz à effet de serre
- Appropriation de données personnelles
- Banditisme
- Espionnage
- Manipulation de l'opinion
- Contrôle social
- etc.



# DÉROULEMENT DE L'UE

- Evaluation
  - quiz
  - oral tuto NoSQL
  - projet
- Je vous redis pour les compétences !

Partie 1 : Bases de données pour le BigData

Partie 2 : Calculs en BigData

Partie 3 : Projet

# CONTENU DE L'UE

## Partie 1 (Hélène Coullon)

- [2h] 03/02 matin
  - Introduction, SQL, NoSQL
  - TP MySQL
  - **Affectation tutoriels NoSQL**
- [3h45] 03/02 aprem
  - TP Mongo
  - **Travail en groupe tutos NoSQL**
- [2h] 10/02 matin
  - Cours partitionnement
  - TP Mongo Shardings
  - **Travail en groupe tutos NoSQL**
- [3h45] 10/02 aprem
  - TP Garage sur Grid'5000
    - **Baptiste Jonglez - Inria**

## Partie 2 (Guillaume Rosinosky)

- [3h45] 11/02 matin
  - Spark Core
  - Spark SQL
- [3h45] 17/02 aprem
  - Spark streaming
  - Kafka
- **Projets par 4/5**
  - [3h45] 24/02 aprem
    - **Quiz NoSQL (20 min)**
  - [3h45] 03/03 aprem
  - [3h45] 10/03 aprem
  - [3h45] 17/03 aprem
    - **Quiz Spark (20 min)**
- **Oraux tutoriels NoSQL 24/03**
- **Oraux projets + TA le 27/03**

# RAPPELS SQL



**IMT Atlantique**  
Bretagne-Pays de la Loire  
École Mines-Télécom

# LES BD RELATIONNELLES ET SQL

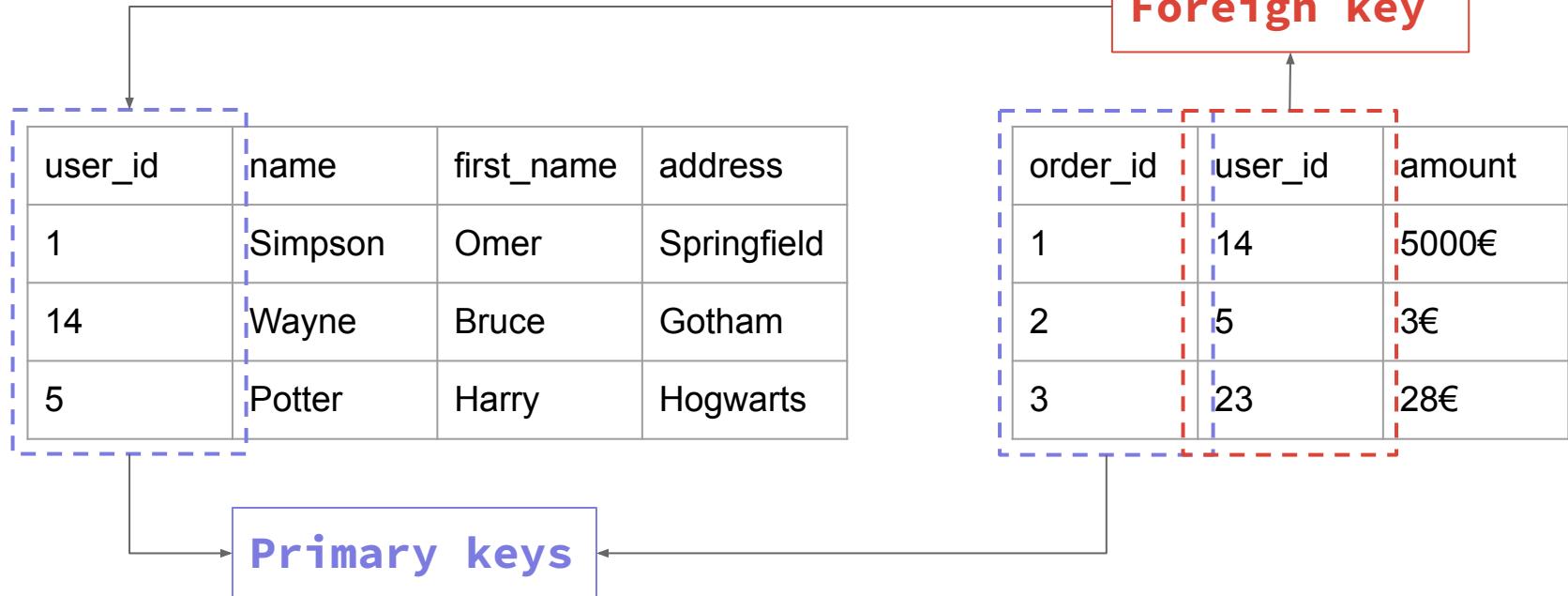
Les BD relationnelles :

- Le modèle relationnel apparaît en 1970. Il est dû à Edgar F. Codd.
- Il s'appuie sur le concept mathématique d'algèbre relationnelle

**SQL** : IBM développe un prototype de SGBD relationnel et son langage SQUARE (Specifying Queries in A Relational Environment) qui devient SEQUEL (Structured English Query Language) puis SQL suite à conflit juridique sur le nom, par la suite interprété Structured Query Language.

Cours et exemples [ici](#)

# RÉSUMÉ BD RELATIONNELLES



Données structurées / Données qui doivent valider le schéma

# SGBD RELATIONNEL

Système de gestion de bases de données

- Définition de la structure ou du **schéma** de données
- **Accès** aux données (ou **requêtes**)
- **Administration** (utilisateurs, sécurité, intégrité, concurrence, reprise sur faute, performances etc.)

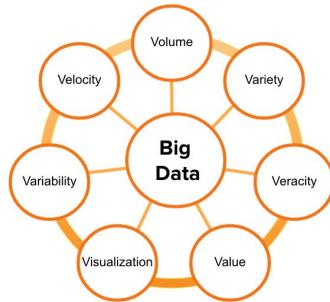
Quelques exemples de SGBD relationnels :

- SQL server (microsoft - propriétaire)
- Oracle database (Oracle corporation - propriétaire)
- **MySQL** (Oracle corporation - ouvert)
- PostgreSQL (projet collaboratif)

# BASES DE DONNÉES NoSQL



IMT Atlantique  
Bretagne-Pays de la Loire  
École Mines-Télécom



# CARACTÉRISTIQUES DES DONNÉES DU BIGDATA

- **volume**
  - il faut pouvoir **partitionner** les données
  - il faut **éviter les mécanismes qui ralentissent les requêtes**
  - il faut pouvoir faire des calculs parallèles
    - **Partie 2**
- **variété**
  - les données peuvent être **structurées semi- ou non-structurées**
  - la **nature** des données est très **variée**
- **variabilité**
  - les **schémas sont un frein** si les données évoluent dans le temps
- **vélocité**
  - on ne peut pas tout stocker, il faut traiter à la volée
    - **Partie 2**

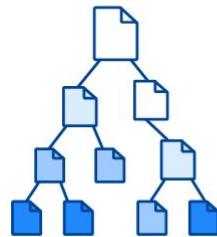
# NOT ONLY SQL (NoSQL)

Utilisent des modèles de données dont la **structure est différente** de celle du modèle relationnel en table, lignes, et colonnes

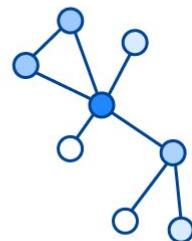
- Première apparition du terme en 1998
  - Carl Strozz - bases légères et open source
- Popularisé par les GAFAM
  - **2000** Neo4J
  - 2004 couchDB
  - **2008** Cassandra
- Adaptées au **BigData** pour différentes raisons...

# 4 TYPES PRINCIPAUX

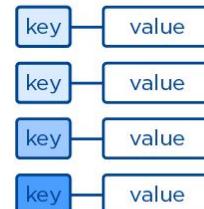
Document



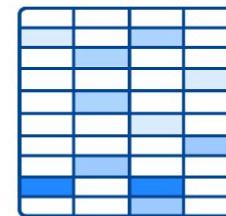
Graph



Key-Value



Wide-column



# ORIENTÉES COLONNES

- Chaque colonne est traitée séparément, et les valeurs sont stockées de façon contigüe
- Hautes performances pour les requêtes d'agrégation comme SUM, COUNT, AVG et MIN



Google  
BigTable



amazon  
REDSHIFT



cassandra

order_id	user_id	amount
1	14	5000€
2	5	3€
3	23	28€
...		
2Billions		6859€

SUM

# ORIENTÉES DOCUMENTS



- Ensemble de collections contenant des documents
- Un document est typiquement un objet JSON associé à une clé unique
- CRUD sur des documents JSON

```
{"id": "iuhd768", "type": "mobile", "name": "iPhone", "version": 15, ...}
```

```
{"id": "leo8", "type": "camera", "name": "Sony aR7", "optics": "mirrorless", ...}
```

```
{"id": "po65h", "type": "DVD", "name": "Harry Potter and the goblet of fire", "year": "2005", ...}
```

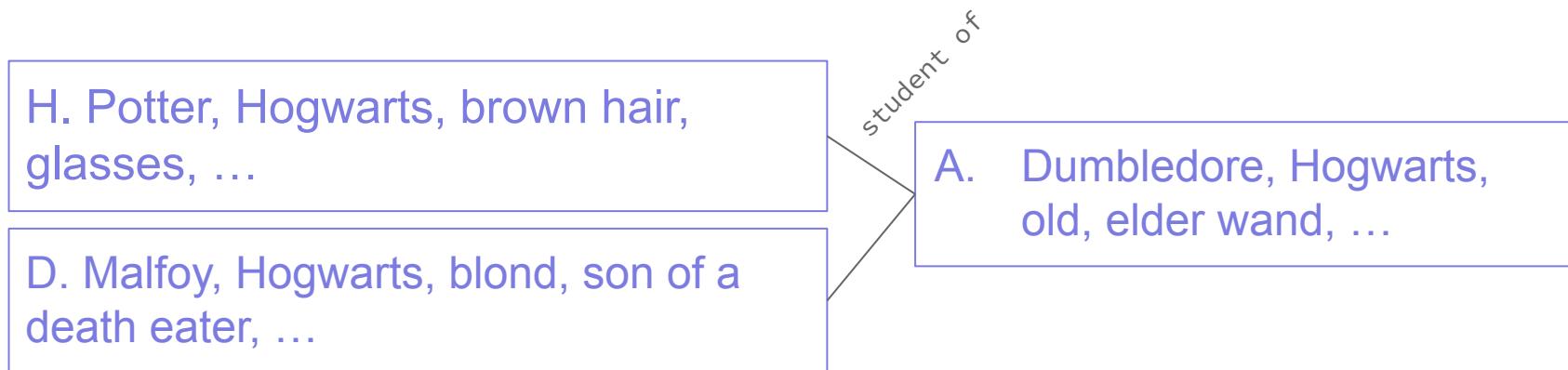
Object storage est similaire mais les objets peuvent être non-structurés



# ORIENTÉES GRAPHES



- L'entité est stockée sous forme de noeud, et les relations comme arêtes
- Facilite la visualisation des relations entre les noeuds
- On l'utilise principalement pour les réseaux sociaux, la logistique, etc.



# ORIENTÉES CLÉS-VALEURS

- Les données sont stockées sous forme de paires clé / valeur (~table de hachage)
- Permet la prise en charge de larges volumes de données
- Les données sont entreposées dans un tableau de “hash” au sein duquel chaque clé est unique
- Stocker facilement des données sans schéma
- On récupère la valeur entière, pas de requêtes complexes



etcd



“dummy”	{"test": "ok", "nothing": True}
786	100111001011
“oih78zz”	42

# AUTRES TYPES

# SÉRIES TEMPORELLES



- Faites pour les événements ou mesures enregistrés dans le temps et associés à un timestamp
- Exemples : monitoring, données de capteurs IoT, enregistrement de clics, transactions financières etc.

timestamp	city (tag key)	country (tag key)	field	field value
2022-01-01T12:00:00Z	London	UK	temperature	12.1
2022-02-01T12:00:00Z	London	UK	temperature	12.5
2022-04-01T12:00:00Z	London	UK	temperature	5.4

# ORIENTÉES RECHERCHE

- Requêtes pour faire de la recherche d'information dans des objets/documents JSON (orienté Read, pas Update)
- Données semi-structurées
- Optimisé pour la recherche d'information



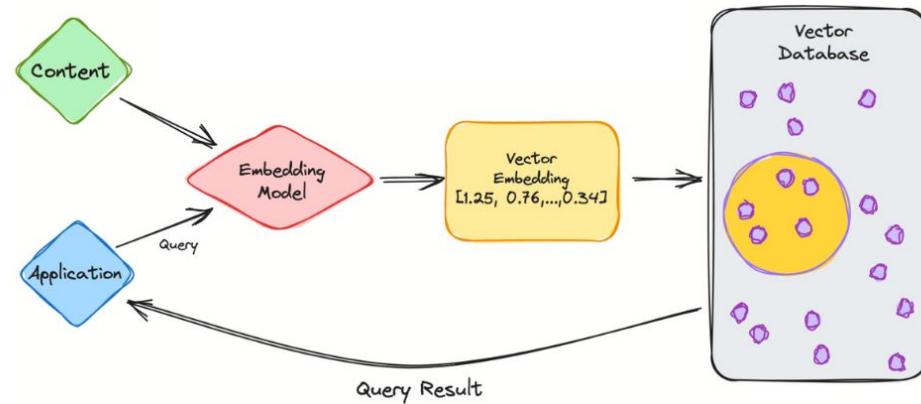
ts | @ algolia

type	ids
"mobile"	[ "iuhd768", "dkhb789" ]

*inverted index*

# VECTORIELLES

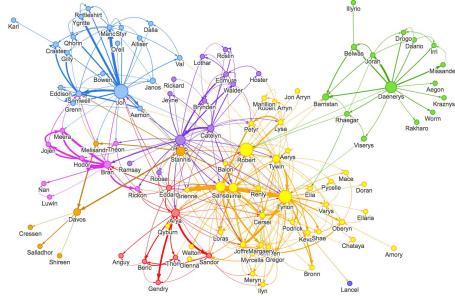
- Recherche de similarités dans des données non-structurées
- Calcul d'un vecteur représentant la donnée pour comparer des données de nature différentes par des *distances*
- Indexation des données pour accélérer les requêtes
- Retrieval Augmented Generation (RAG) : LLM utilise des bases vectorielles pour dynamiquement enrichir sa génération



# VISUALISATION DE LA DONNÉE

- Prometheus, Grafana pour les time-series
  - Elastic Kibana
  - Neo4j Browser, Neo4J Bloom, Neovis.js
  - ...

Mais il faudra découvrir par vous même, peut être dans votre tutoriel ou votre projet ?



# MONGODB

# RAPPELS JSON



IMT Atlantique  
Bretagne-Pays de la Loire  
École Mines-Télécom

# ORIENTÉES DOCUMENTS



- Ensemble de collections (~=tables) contenant des documents (~=lignes)
- Un document est typiquement un objet JSON associé à une clé unique
  - *la clé est ajoutée par Mongo à l'insertion*
- CRUD sur des documents JSON

```
{"id": "iuhd768", "type": "mobile", "name": "iPhone", "version": 15, ...}
```

```
{"id": "leo8", "type": "camera", "name": "Sony aR7", "optics": "mirrorless", ...}
```

```
{"id": "po65h", "type": "DVD", "name": "Harry Potter and the goblet of fire", "year": "2005", ...}
```

# DES ARBRES !

- **Valeurs atomiques**
  - Des chaînes de caractères
  - Des nombres
  - 3 constantes : true, false et null.
- **Paires clé/valeur**
  - La clé est un string
  - La valeur est une valeur atomique
  - Le nom et la valeur d'une paire sont séparés par un deux-points
- 2 types de valeurs **composites**
  - Une **collection de paires clé/valeur** ~= objet, dictionnaire, enregistrement, liste de paires
    - Une collection est délimitée par des accolades
    - Ses paires sont séparées par des virgules
  - Une **séquence de valeurs** ~= un tableau, un vecteur, une liste
    - Un tableau est délimité par des crochets
    - Ses éléments sont séparées par des virgules

## Exemples

TP

- Revoir SQL avec MySQL

# NOTÉ

# CONCEPTION D'UN TUTORIEL NOSQL

**6 sujets :**

- REDIS (key-value store)
- ClickHouse (columnar)
- InfluxDB (time series)
- Elasticsearch (search)
- Milvus (vector)
- Neo4J (graphes)

[Inscriptions ici](#)

**Rendus :**

- tutoriel conteneurisé
- repo GitHub et README du tuto
- utilisation d'un dataset dispo en ligne de votre choix
- visualisation des données (si vous avez le temps)

TP

- TP Mongo

# CONTENU DE L'UE

## Partie 1 (Hélène Coullon)

- [2h] 03/02 matin
  - Introduction, SQL, NoSQL
  - TP MySQL
  - **Affectation tutoriels NoSQL**
- [3h45] 03/02 aprem
  - TP Mongo
  - **Travail en groupe tutos NoSQL**
- [2h] 10/02 matin
  - Cours partitionnement
  - TP Mongo Shardings
  - **Travail en groupe tutos NoSQL**
- [3h45] 10/02 aprem
  - TP Garage sur Grid'5000
    - **Baptiste Jonglez - Inria**

## Partie 2 (Guillaume Rosinosky)

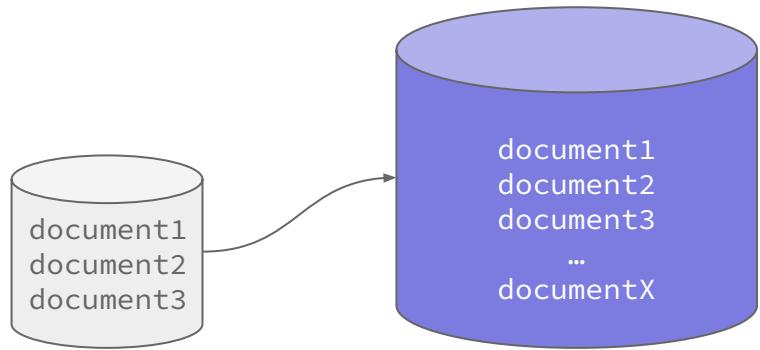
- [3h45] 11/02 matin
  - Spark Core
  - Spark SQL
- [3h45] 17/02 aprem
  - Spark streaming
  - Kafka
- **Projets par 4/5**
  - [3h45] 24/02 aprem
    - **Quiz NoSQL (20 min)**
  - [3h45] 03/03 aprem
  - [3h45] 10/03 aprem
  - [3h45] 17/03 aprem
    - **Quiz Spark (20 min)**
- **Oraux tutoriels NoSQL 24/03**
- **Oraux projets + TA le 27/03**

# SCALABILITÉ ET DISTRIBUTION



IMT Atlantique  
Bretagne-Pays de la Loire  
École Mines-Télécom

# SCALABILITÉ VERTICALE - GÉRER PLUS DE DONNÉES SUR 1 SERVEUR

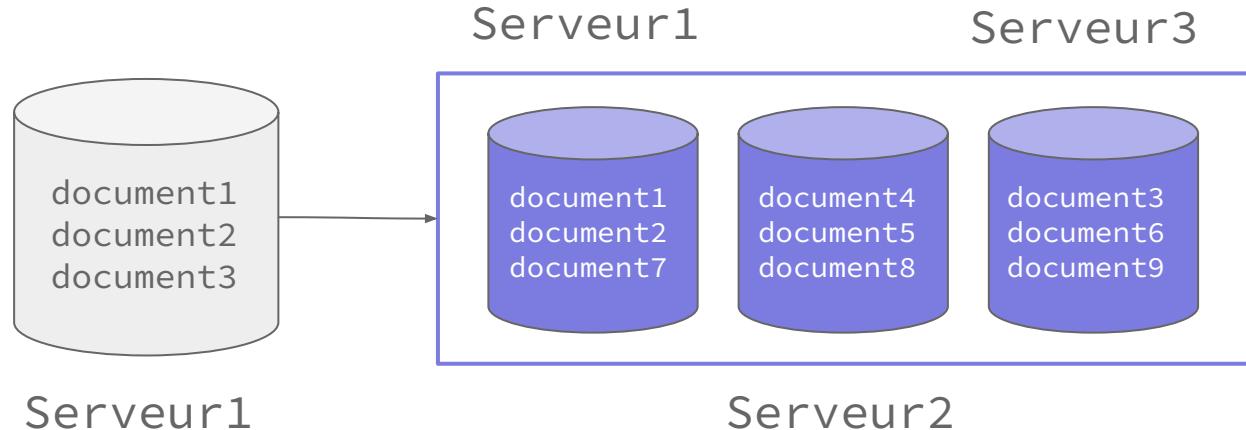


- + de CPU
- + de RAM
- + de disque

# SCALABILITÉ HORIZONTALE - PLUS DE SERVEURS

## Plus de serveurs pour

- stocker les données -> comment **partitionner** ?
  - répondre aux requêtes



**! différent de la réPLICATION**

# PARTITIONNEMENT VERTICAL

- Répartir les colonnes (les tables) d'une base de données sur les différents serveurs
- S'applique plutôt aux BD relationnelles normalisées
- Problème : les jointures demandent des échanges de messages entre les serveurs

Serveur1			
user_id	name	first_name	address
1	Simpson	Omer	Springfield
14	Wayne	Bruce	Gotham
5	Potter	Harry	Hogwarts

Serveur2		
order_id	user_id	amount
1	14	5000€
2	5	3€
3	23	28€

# PARTITIONNEMENT HORIZONTAL - SHARDING

- Répartir les lignes des tables sur les différents serveurs
- Bien sélectionner la shard key (pour bien répartir la charge)
  - range-based sharding : [A-I]→S1, [J-R]→S2, [S-Z]→S3
  - hash-based sharding : hash1→S1, hash2→S2, hash3→S3
  - directory-based sharding : red→S1, blue→S2, green→S3
  - geographical-based sharding ...
- MySQL propose aussi du sharding maintenant !

Shard1

user_id	name	first_name	address
1	Simpson	Omer	Springfield
14	Wayne	Bruce	Gotham

Shard2

user_id	name	first_name	address
5	Potter	Harry	Hogwarts

# DÉNORMALISATION

Un modèle de données **dénormalisé** embarque **toutes les données associées dans un seul document** au lieu de répartir l'information dans différentes collections et documents.

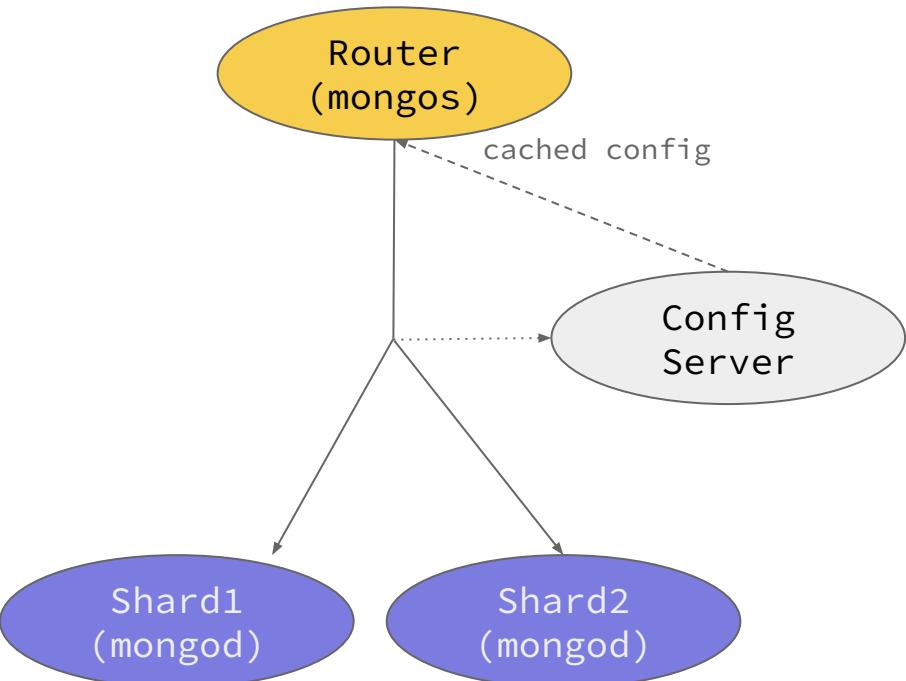
*(Concerne principalement les BD NoSQL qui gère des documents)*

## Objectifs :

- éviter au maximum les jointures qui ralentissent les requêtes surtout quand on a de très gros volumes de données
- faciliter le partitionnement horizontal (sharding)

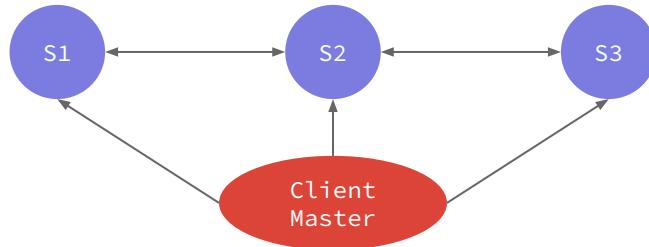
Une **réPLICATION** de l'information est parfois nécessaire entre différents documents. Attention à la **cohérence** des données !

# EN PRATIQUE AVEC MONGO



- **Router (mongos)**
  - The mongo server acts as a query router, providing an interface between client applications and the sharded cluster
- **Config server**
  - Config server stores metadata and configuration settings for the cluster
- **Shards (mongod)**
  - each shard contains a subset of the sharded data

# RÉPLICATION ET BD DISTRIBUÉES



**RéPLICATION pour la tolérance aux pannes** (ne pas perdre de données)

- données disponibles à plusieurs endroits
- gestion de la consistance des données ?
- tolérance aux partitionnements réseau ?

**On ne peut pas tout avoir à la fois !**

# THÉORÈME CAP

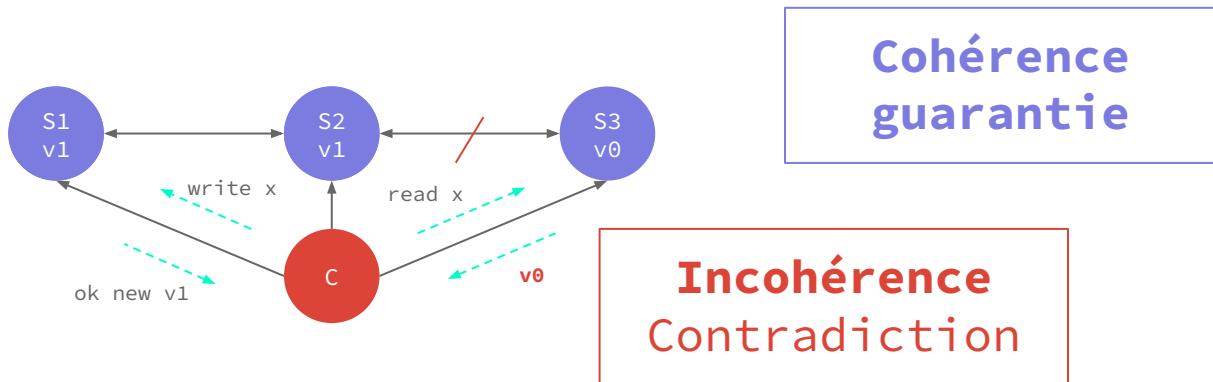
3 propriétés souhaitables sur les bases de données distribuées :

- La **cohérence** (consistency) signifie que la donnée est la même sur tous les noeuds (version la plus récente)
- La **disponibilité** (availability) signifie que toute requête obtient une réponse, même si un ou plusieurs nœuds sont en panne
- La **tolérance au partitionnement réseau** signifie que le cluster doit continuer à fonctionner malgré un nombre quelconque de pannes de communication entre les nœuds du système

**Seulement 2 de ces 3 propriétés peuvent être vérifiées à la fois**

# INTUITION DE LA PREUVE

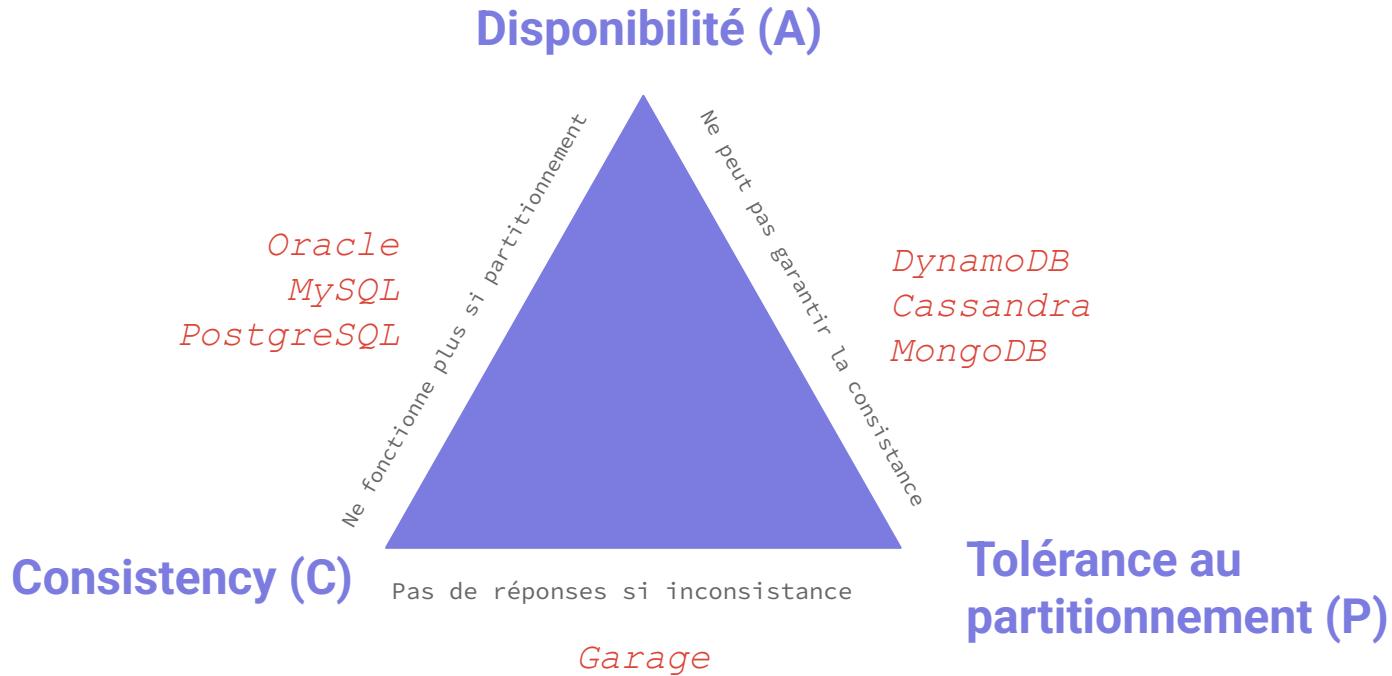
Par l'absurde : imaginons que les 3 propriétés sont validées



Tolérance au partitionnement  
Le système continue de fonctionner

Disponible  
Le système répond

# PROBLÈME CAP

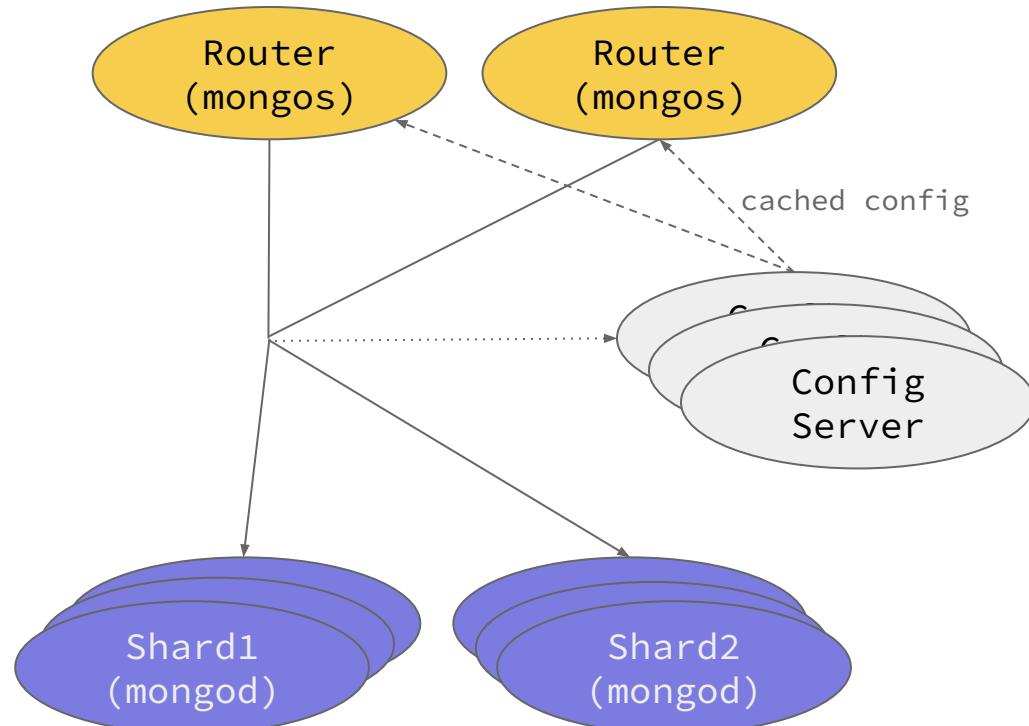


Modèles de consistance faible peuvent rendre CAP un peu plus “possible”

# EN PRATIQUE AVEC MONGO (BIS)

En pratique il faut coupler le sharding avec la réPLICATION !

- scalabilité
- tolérance aux pannes



# TP

- TP Sharded Mongo
- TP BD distribuée Garage
  - Baptiste Jonglez

# CALCULS PARALLÈLES

## MAP-REDUCE



IMT Atlantique  
Bretagne-Pays de la Loire  
École Mines-Télécom

# MOTIVATION

Début des années 2000, Google automatise l'exploration et l'indexation du web

- Plus de 1 milliard de pages de 20 Ko = 20 To
- Performance d'un disque dur : ~ 20 Mo/s, 10 Go
- Il faut :
  - pour enregistrer le web : 2000 disques durs
  - pour traiter le web :
    - 1 mois pour “lire” le web avec une machine
    - 8 minutes avec 2000 machines

Il faut **paralléliser** les calculs ! -> **Map-Reduce**

# CALCULS PARALLÈLES

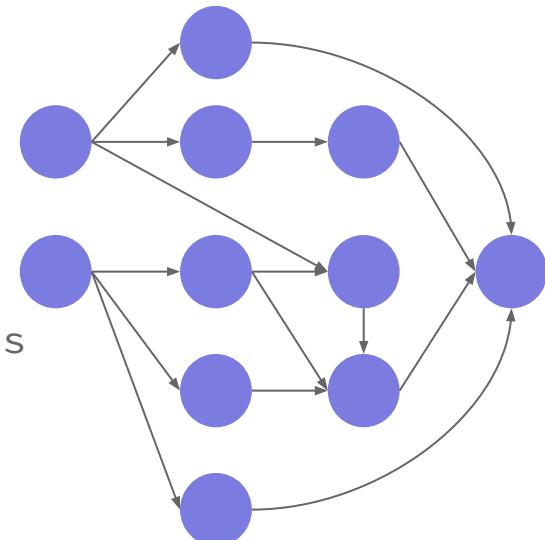
Le calcul parallèle ça date de bien longtemps ! ~Années 1960

On parallélise pour

- aller plus vite
- traiter de plus gros problèmes
- traiter plus de données

Différents types de parallélisme :

- Parallélisme de **tâches**
  - exécution de plusieurs tâches en même temps
  - ordre partiel de tâches
  - partage des données
- Parallélisme de **données**
  - division des données
  - application du même calcul sur les différentes données



# MAP-REDUCE ?

- Parallélisme de **données**
- Calculs “simples” sur beaucoup de données
  - indexes inversés
  - liens entre les pages web
  - page rank etc.
- Aller plus vite et traiter plus de données
- Ne cherche pas à optimiser au maximum (pas HPC)
- Mais cherche à **abstraire la complexité**
  - un **modèle de programmation simple**
  - parallélisme **automatique / distribution** des données
  - les **communications** et la **coordination** des calculs
  - la tolérance aux pannes (fault tolerance)
  - l'équilibrage de charges (load balancing)

# GOOGLE MAPREDUCE (2000)

Map et Reduce sont **deux fonctions classiques d'ordre supérieur de la programmation fonctionnelle**

- Une bibliothèque (C++)
  - fournit les primitives de programmation sous forme d'appels à la bibliothèque
  - gère les "détails" précédents une fois pour toute

Les étapes :

1. **Lecture** des données
2. **Map** : extrait de chacune des données un résultat intermédiaire
3. **Shuffle** (GroupByKey, Sort) : réarrange et trie les résultats intermédiaires
4. **Reduce** : filtre, agrège, résume, transforme... les résultats intermédiaires
5. **Ecriture** du résultat

Seules les étapes Map et Reduce dépendent du problème.

# MAP

Soit une liste `l = list(e1, ..., en)`

- `map(f, l)` applique `f` à chacun des éléments de la liste
- `map(f, l) = list(f(e1), ..., f(en))`
- Le principe s'étend aux collections.

# REDUCE

Soit une liste `l = list(e1, ..., en)`

- `reduce(op, l)` où `op` est un opérateur binaire **associatif**
- effectue le calcul `e1 op e2... op en-1 op en`

Dans les cas limite ( $n \leq 1$ )

- la réduction retourne un résultat optionnel (`e1` pour  $n = 1$ )
- un 3ème paramètre définit une valeur à associer à la liste vide

# HADOOP

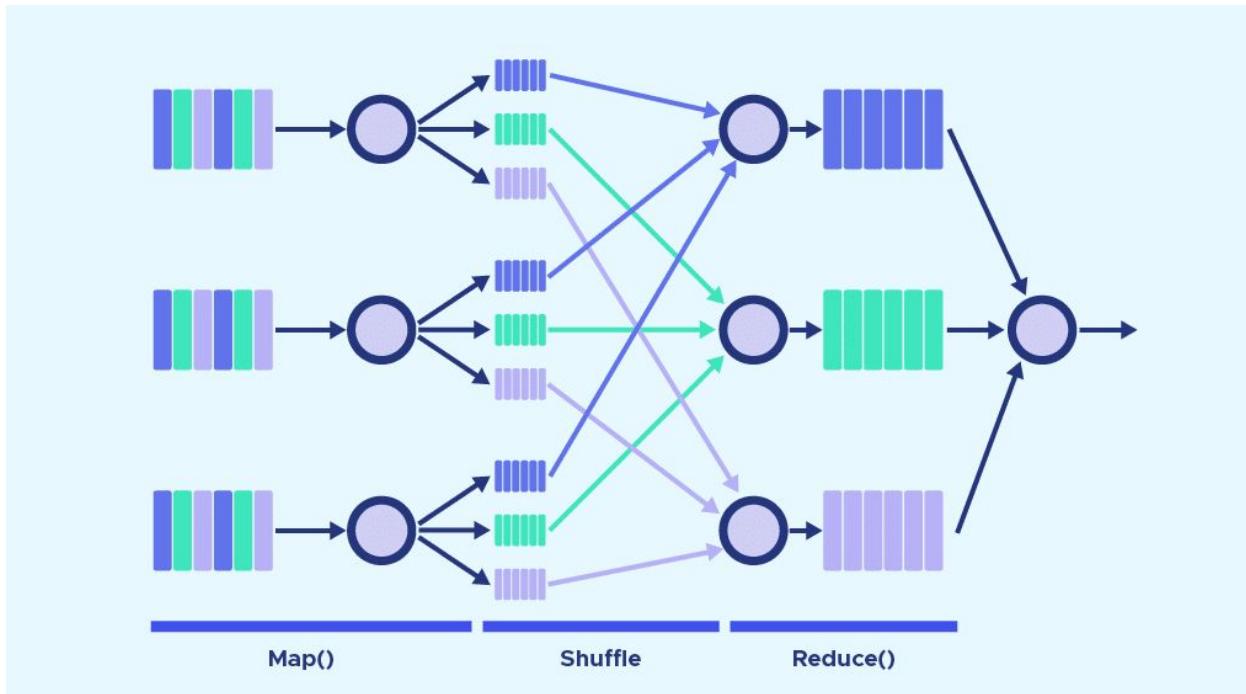
Projet Apache depuis 2006 (basé sur des développements antérieurs notamment de Doug Cutting, et inspirés par GFS et Google MapReduce)

- Écrit en **Java**
- Fournit le modèle de programmation **MapReduce** au dessus de
  - HDFS (Hadoop Distributed File System)
  - YARN (Yet Another Resource Negotiator) qui gère l'allocation des ressources et l'ordonnancement des tâches (depuis 2012)

En voie de désuétude... mais le modèle de programmation reste d'actualité !



# VISION DE HAUT NIVEAU



# MACHINES PARALLÈLES

HPC, Cloud, clusters



# CLOUD VS HPC

- La **Data** c'est plutôt le monde du **Cloud**
  - orienté “services”
  - disponibilité
  - élasticité
  - **la donnée c'est du pouvoir**
- Le **calcul parallèle** c'est plutôt le monde du **HPC**
  - calculer **le plus vite possible** (Calcul Haute Performance)
  - résoudre des problèmes nouveaux grâce à la puissance de calcul (décodage du génome humain, IA, forages, astronomie, météo et climat, etc.)
  - plutôt orienté “science”
  - **le calcul c'est du pouvoir**

**Le Cloud/Data/IA comme le HPC/IA sont désormais des éléments géo-politiques. Et les deux mondes sont de plus en plus proches**

Le BigData est un peu au milieu mais un peu plus proche du Cloud

# PARLONS MACHINES HPC

- Top500 ... La course à la puissance de calcul
  - Et la Chine ?
- Green500 ... “Green” mmoui

**FLOPS** (FLoating point Operations Per Second) : la performance s'exprime en opérations (additions ou multiplications) à virgule flottante par seconde



# CLUSTER DE MACHINES

## Définition

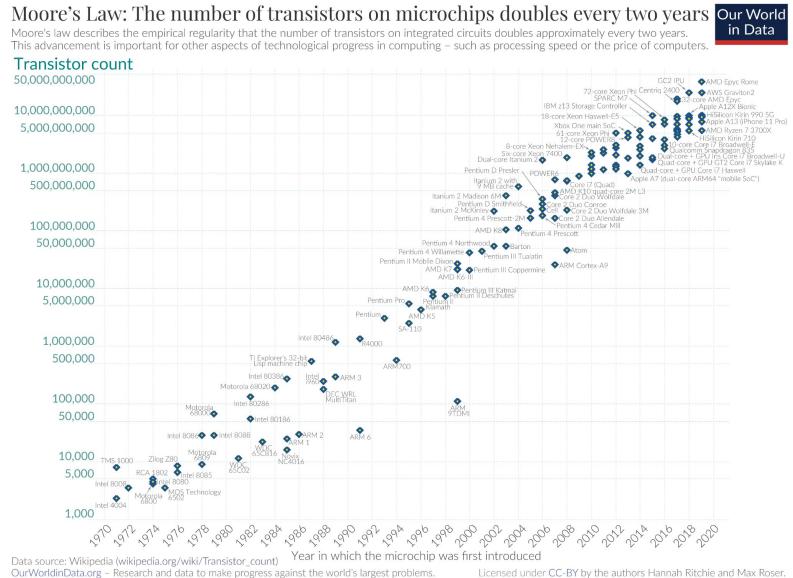
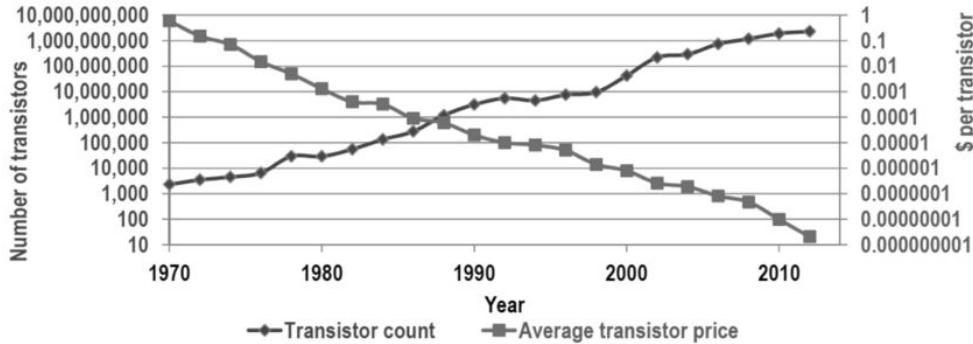
- A **grand nombre de serveurs** très **proches** les uns des autres
- Interconnectés par un **réseau haute performance**

## Remarques

- Une machine HPC est un très gros cluster de machines
- Les centres de calculs HPC sont constitués de plusieurs clusters
- Les clouds sont aussi un ensemble de clusters divisés en régions
- Hadoop ou Spark sont typiquement déployés sur des clusters
- Il est bon de noter qu'on peut aussi faire des clusters de machines virtuelles

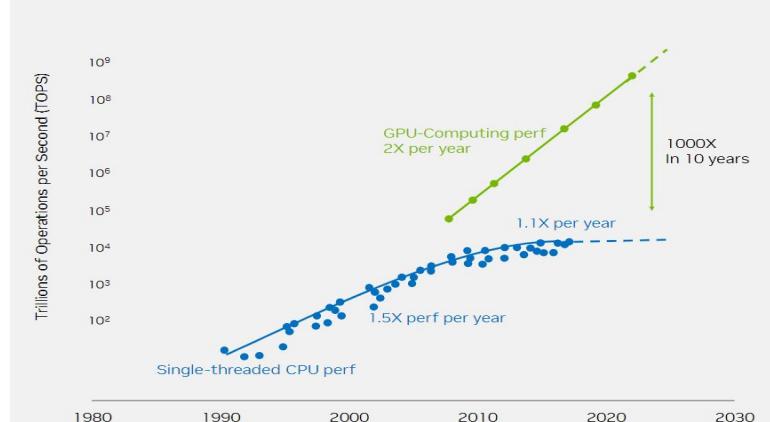
# LOI DE MOORE ET ARCHITECTURES

- **Loi empirique de 1965** (revue en 1975)
  - Doublement du nombre de transistors sur une puce CPU tous les deux ans à coût constant

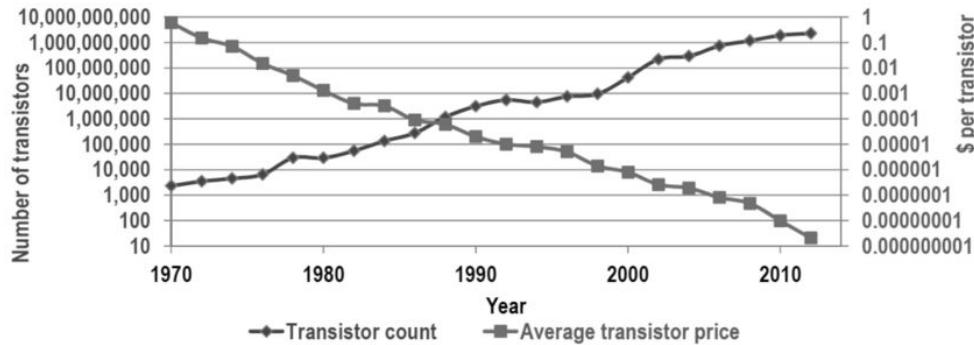


# LOI DE MOORE ET ARCHITECTURES

- Mauvaise interprétation – fréquence des processeurs
  - stagne depuis 2004 à 5GHz
  - record de 500 GHz par IBM
    - transistor équipé d'une puce à base de silicium-germanium
    - refroidi à -269 °C à l'hélium liquide
- Les architectures se sont complexifiées
  - la loi reste vrai
  - **multi-core**
  - **accélérateurs graphiques**



# PRIX DU MATÉRIEL

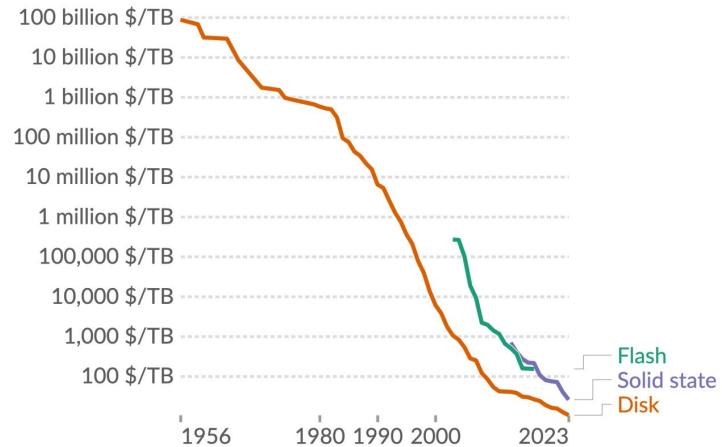


## Historical price of computer storage

Our World  
in Data

Expressed in US dollars per terabyte (TB), adjusted for inflation.

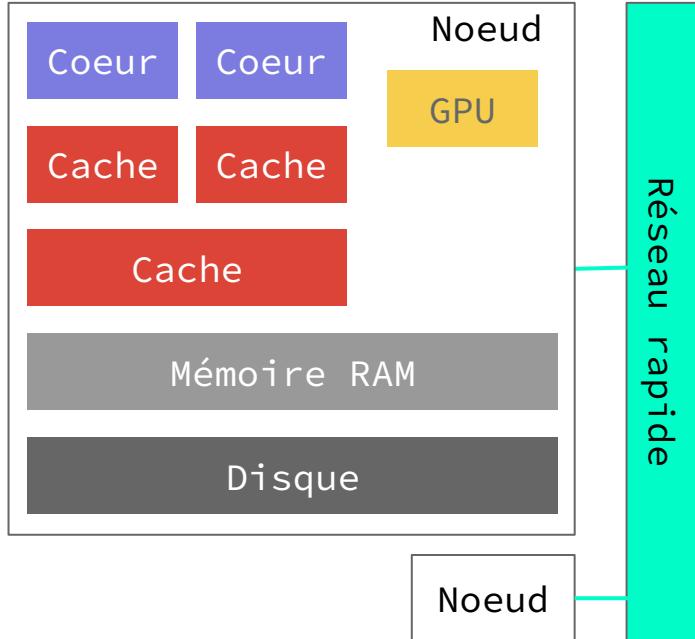
"Disk" refers to magnetic storage, "flash" to memory used for rapid data access and rewriting, and "solid state" to solid-state drives (SSDs).



Data source: John C. McCallum (2023)

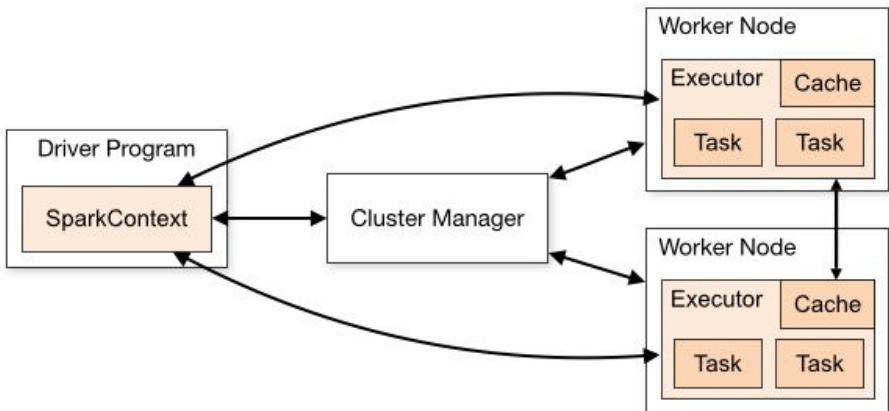
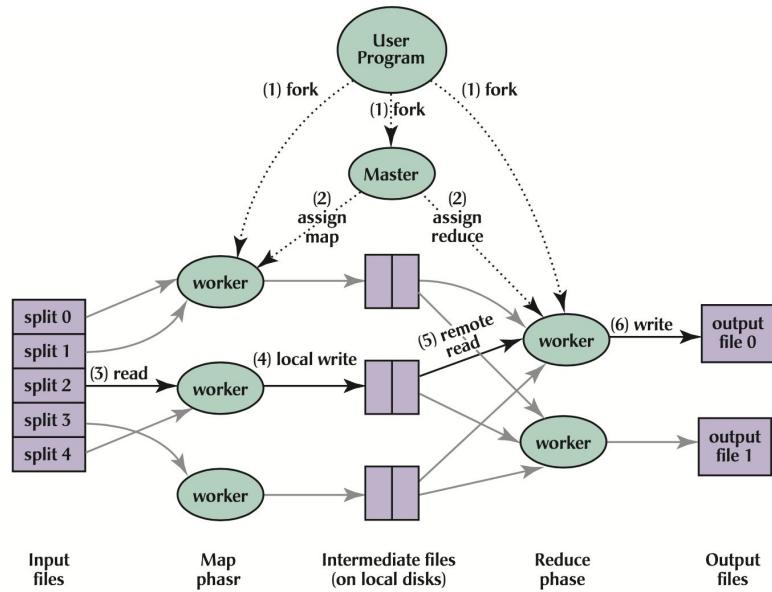
CC BY

# ARCHITECTURE D'UNE MACHINE ET D'UN CLUSTER



- **Cœur** : ressource de calcul
- **Processeur** / CPU : composé de 1 ou plusieurs coeurs
- **Mémoire** : organisée de façon hiérarchique
- **Noeud** : plusieurs processeurs qui partagent une même mémoire (NUMA)
- **Réseau rapide** : relie les noeuds entre eux
- **Système de fichier** : entrées/sorties sur disques

# MAPREDUCE, HADOOP ET SPARK



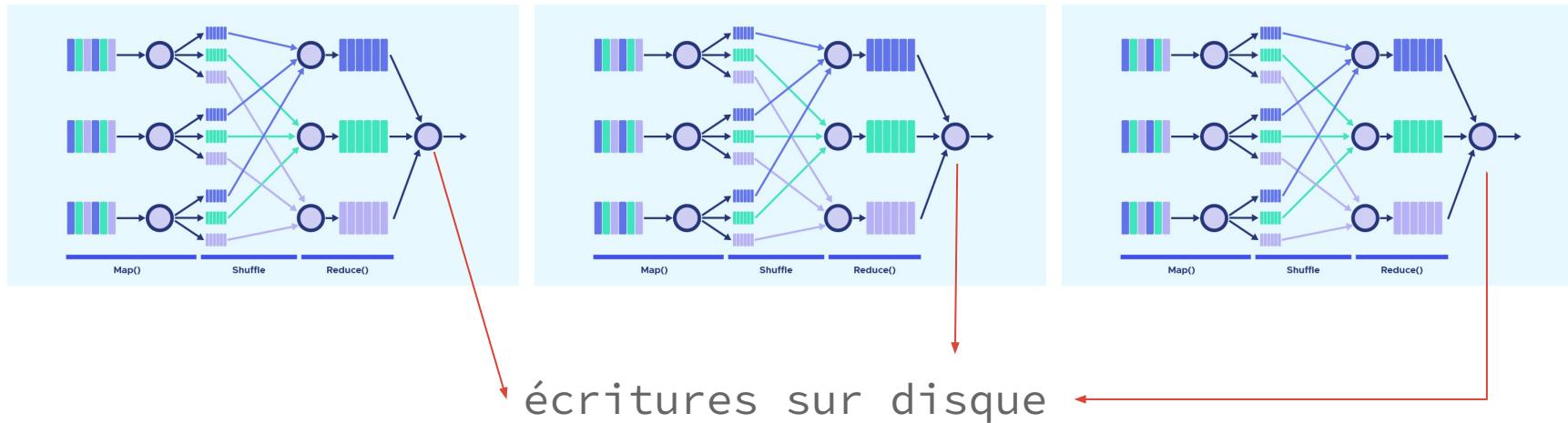
SPARK



**IMT Atlantique**  
Bretagne-Pays de la Loire  
École Mines-Télécom

# PROBLÈME D'ÉCRITURES ET DE LECTURES SUR DISQUE

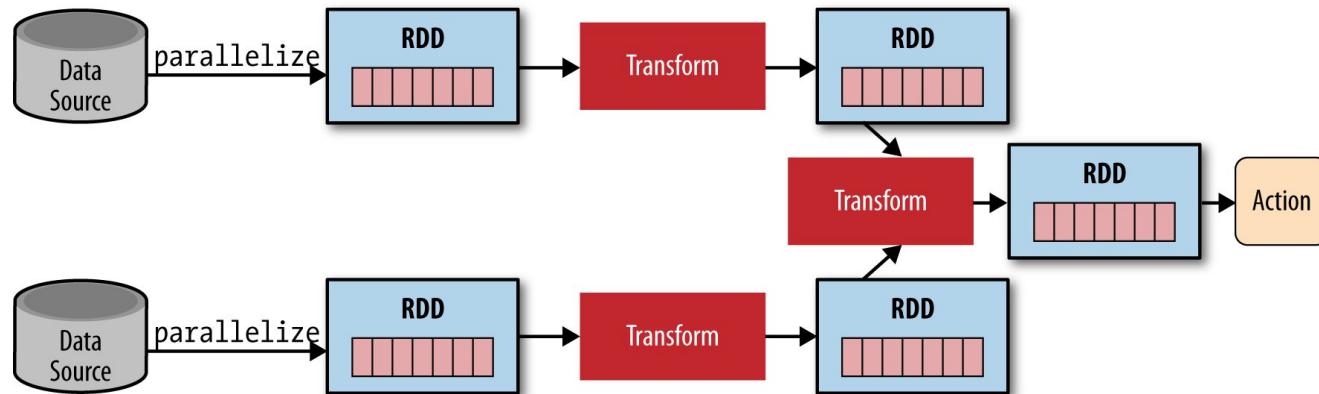
Avec Google MapReduce / Hadoop



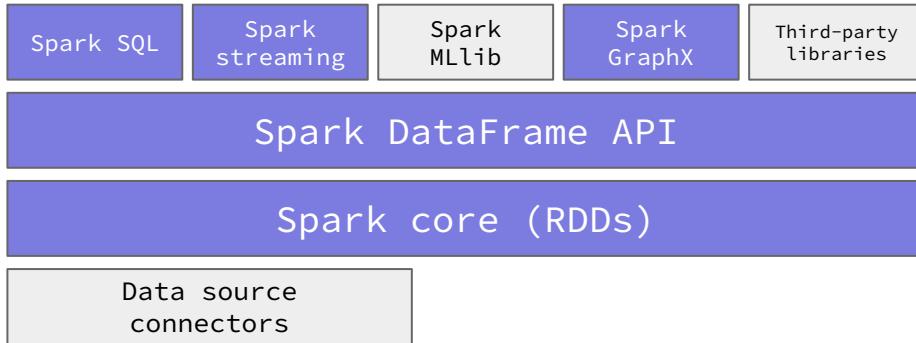
*Aussi ne pas faire nécessairement un reduce après le map ? plusieurs map, des filtres, et autres opérations*

# IN-MEMORY APPROACH

- Ne plus faire d'écritures intermédiaires
- Pouvoir enchaîner des **transformations** sur des **données en mémoire** et déjà **partitionnées**
- Écrire seulement lorsqu'une **action** est demandée (write, reduce etc.)

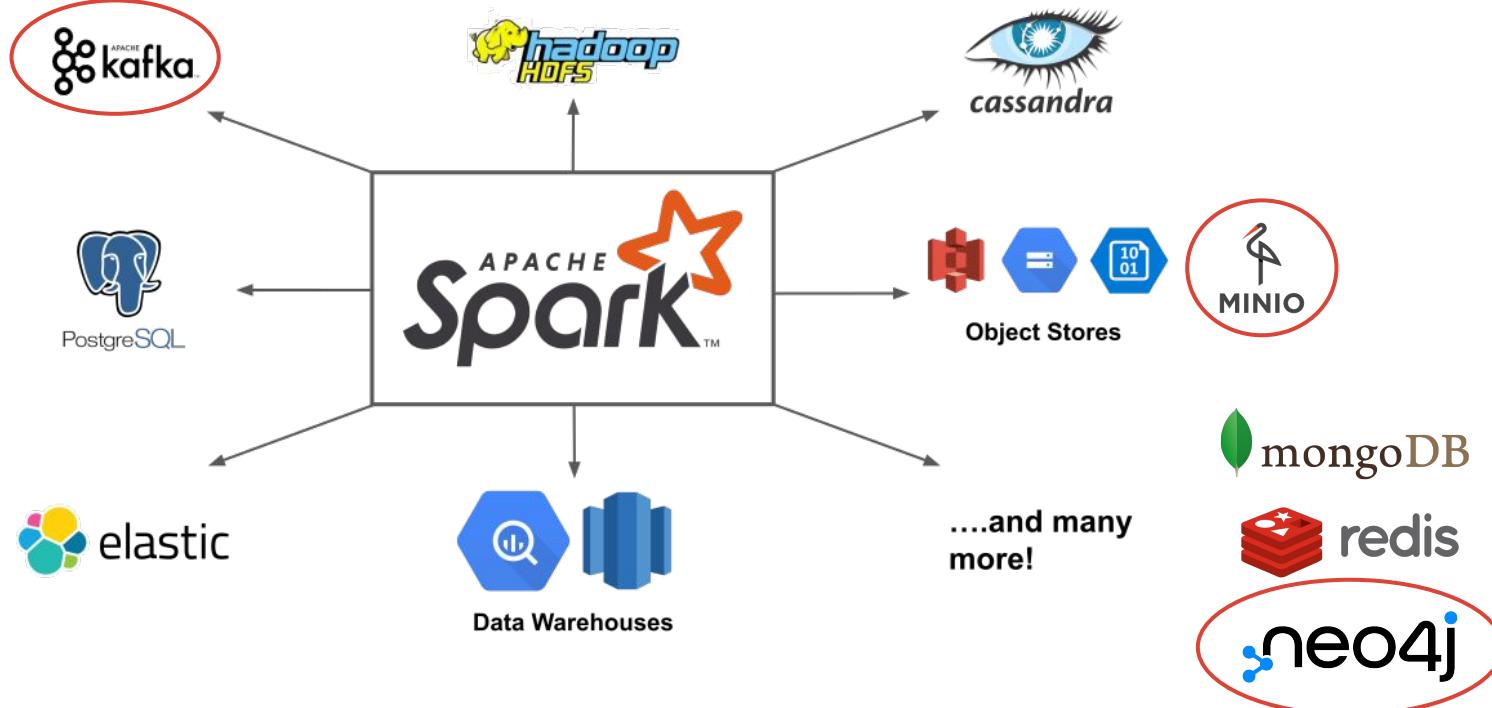


# SPARK



- **Unified analytics engine** for large-scale data processing
- High-level APIs in Java, Scala, Python and R
- Rich set of **higher-level tools** including
  - **Spark SQL** for SQL and structured data processing
  - MLLib for machine learning
  - GraphX for graph processing
  - **Streaming** for incremental computation and stream processing

# SPARK DATA SOURCES



# SPARK CORE - RDD

- The main abstraction Spark provides is a **Resilient Distributed Dataset** (RDD)
- An RDD is a **Collection of elements partitioned** across the nodes of the Spark cluster
- RDDs can be operated on in parallel
- **2 types** of computations on RDDs
  - **Transformations** create a new dataset from an existing one
  - **Actions** return a value to the *driver* program after running a computation on the dataset

# SPARK CORE - TRANSFORMATIONS ET ACTIONS

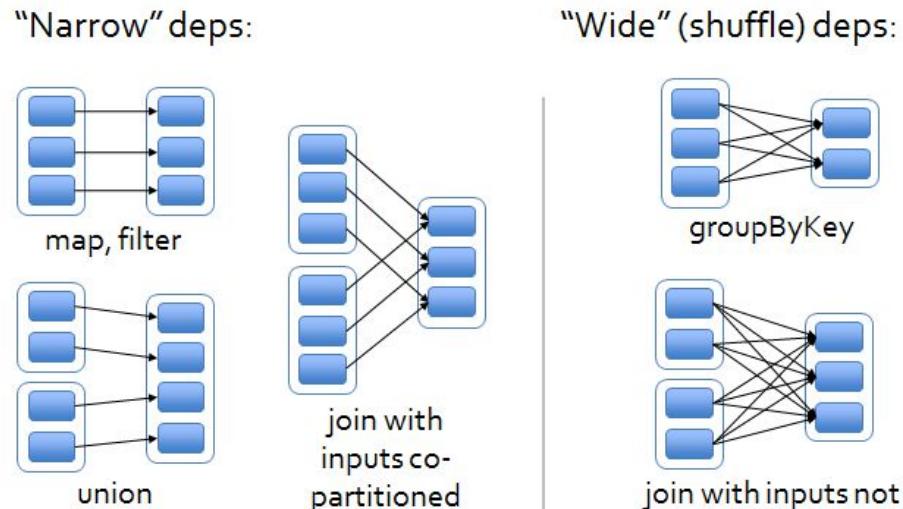
Création initiale d'un RDD : Fonction parallelize

## Transformations

- Map
- Filter
- GroupBy
- etc.

## Actions

- Collect
- Reduce
- Save
- etc.

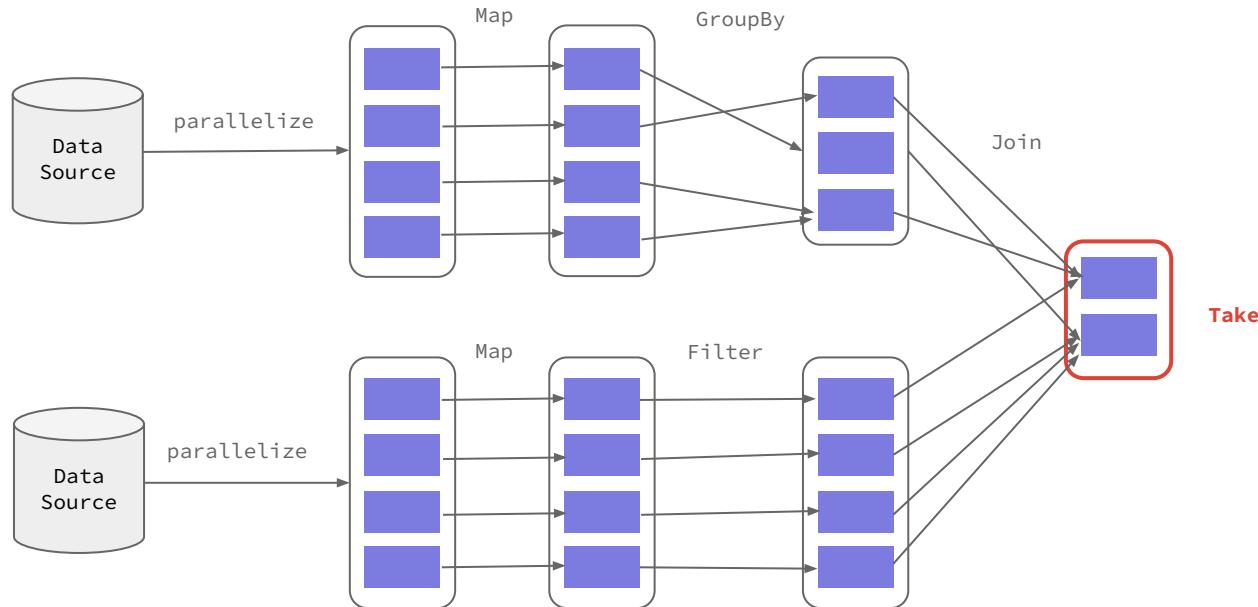


# SPARK CORE - TRANSFORMATIONS ET ACTIONS

<b>Transformations</b>	$map(f : T \Rightarrow U)$ : $\text{RDD}[T] \Rightarrow \text{RDD}[U]$ $filter(f : T \Rightarrow \text{Bool})$ : $\text{RDD}[T] \Rightarrow \text{RDD}[T]$ $flatMap(f : T \Rightarrow \text{Seq}[U])$ : $\text{RDD}[T] \Rightarrow \text{RDD}[U]$ $sample(\text{fraction} : \text{Float})$ : $\text{RDD}[T] \Rightarrow \text{RDD}[T]$ (Deterministic sampling) $groupByKey()$ : $\text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, \text{Seq}[V])]$ $reduceByKey(f : (V, V) \Rightarrow V)$ : $\text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)]$ $union()$ : $(\text{RDD}[T], \text{RDD}[T]) \Rightarrow \text{RDD}[T]$ $join()$ : $(\text{RDD}[(K, V)], \text{RDD}[(K, W)]) \Rightarrow \text{RDD}[(K, (V, W))]$ $cogroup()$ : $(\text{RDD}[(K, V)], \text{RDD}[(K, W)]) \Rightarrow \text{RDD}[(K, (\text{Seq}[V], \text{Seq}[W]))]$ $crossProduct()$ : $(\text{RDD}[T], \text{RDD}[U]) \Rightarrow \text{RDD}[(T, U)]$ $mapValues(f : V \Rightarrow W)$ : $\text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, W)]$ (Preserves partitioning) $sort(c : \text{Comparator}[K])$ : $\text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)]$ $partitionBy(p : \text{Partitioner}[K])$ : $\text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)]$
<b>Actions</b>	$count()$ : $\text{RDD}[T] \Rightarrow \text{Long}$ $collect()$ : $\text{RDD}[T] \Rightarrow \text{Seq}[T]$ $reduce(f : (T, T) \Rightarrow T)$ : $\text{RDD}[T] \Rightarrow T$ $lookup(k : K)$ : $\text{RDD}[(K, V)] \Rightarrow \text{Seq}[V]$ (On hash/range partitioned RDDs) $save(path : \text{String})$ : Outputs RDD to a storage system, e.g., HDFS

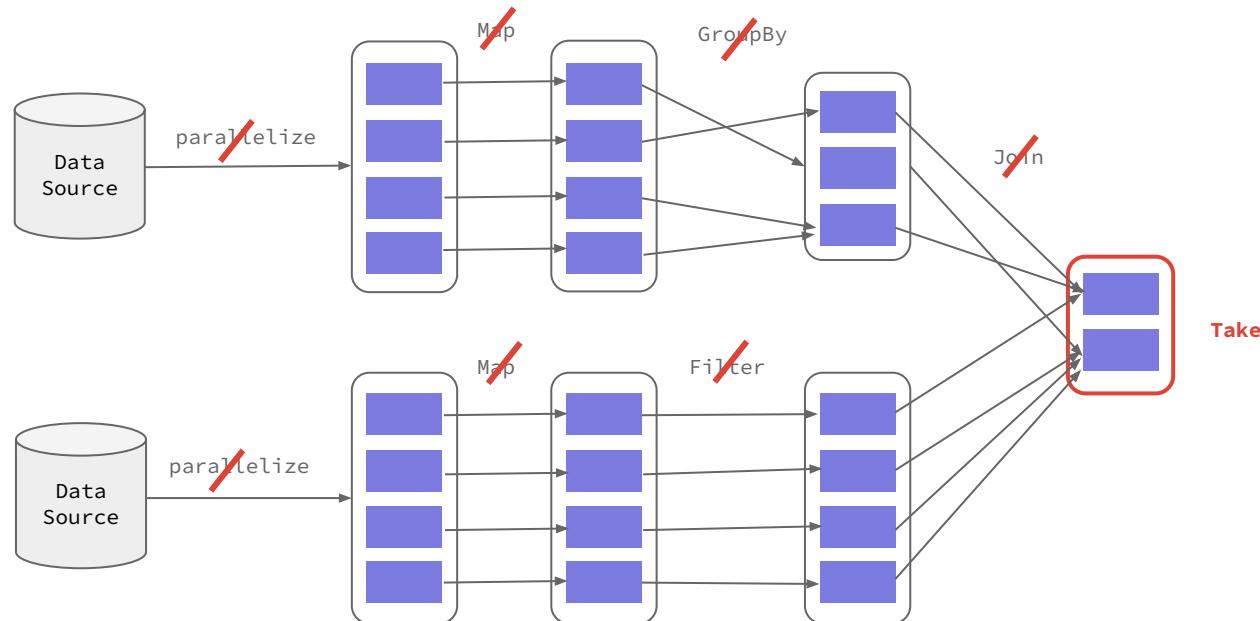
Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

# SPARK CORE - EXAMPLE



# LAZY EVALUATION

- Nothing is computed on RDDs until an action is called!
- Each transformed RDD may be recomputed each time you run an action on it
- Can also persist an RDD in memory using the `persist` (or `cache`) method



# ARCHITECTURE DE SPARK

