

# Chapitre 1: Introduction au BIG DATA

## 1/Qu'est-ce que le Big Data ?

- Le **Big Data** désigne l'ensemble des **données dynamiques, volumineuses et disparates (provenant de sources variées et sous des formats différents)** produites en continu par les individus, les entreprises, les objets connectés et les systèmes informatiques. Ces données se caractérisent par leur **volume important, leur vitesse élevée et leur variété**. Pour exploiter efficacement ces informations, des technologies innovantes sont nécessaires afin de les **collecter, stocker, traiter et analyser en temps réel**.

## 2/Les principales évolutions du Big Data

1. **Changement en volume** 
  - La quantité de données générées est en forte croissance.
  - Exemples :
    - Les réseaux sociaux (Facebook, Twitter, Instagram) produisent des **pétaoctets** de données chaque jour.
    - Les objets connectés (IoT) génèrent en permanence des informations (capteurs, montres intelligentes, etc.).
2. **Changement d'origine des données** 
  - Les données ne proviennent plus uniquement des bases de données classiques mais de **sources variées** :
    - Réseaux sociaux, capteurs IoT, logs de serveurs, vidéos, etc.
    - Exemple : Un simple smartphone collecte et transmet en permanence des données (localisation, navigation web, utilisation d'applications, etc.).
3. **Changement de structure des données** 
  - Les données sont de **différentes formes** :
    - **Structurées** : Bases de données relationnelles (ex. SQL).
    - **Semi-structurées** : Données avec un certain format mais non fixes (JSON, XML, logs).

Un **log** est un enregistrement d'événements et d'activités d'un système ou d'une application, utilisé pour le suivi, le diagnostic et la sécurité.

XML (**eXtensible Markup Language**) est un langage de balisage utilisé pour structurer des données avec des **métadonnées** décrivant leur contenu. Il est couramment employé pour l'échange de données entre systèmes hétérogènes, car il est **flexible et lisible**.

Contrairement au **modèle relationnel**, XML permet de gérer des données hiérarchiques et semi-structurées sans schéma rigide, ce qui le rend plus adapté aux documents complexes et aux échanges inter-systèmes.

- **Non-structurées** : Images, vidéos, fichiers audio, emails, etc.

- L'analyse de ces nouvelles structures nécessite des **technologies adaptées** comme les bases de données NoSQL et le Machine Learning.

#### 4. Capacités de stockage accrues (Cloud Computing)

- Le stockage traditionnel sur **serveurs physiques** est remplacé par des solutions plus flexibles et évolutives comme le **Cloud Computing**.
- **Le Cloud Computing, c'est quoi ?**
  - C'est un modèle permettant d'accéder à des ressources informatiques (serveurs, bases de données, logiciels) via Internet au lieu de les stocker localement.
  - **Avantages :**
    - Évolutivité (ajout de ressources en fonction des besoins).
    - Réduction des coûts (pas besoin d'acheter des infrastructures physiques).
    - Accessibilité à distance.

#### 5. Changement des traitements des données

- Les méthodes classiques de traitement ne sont plus suffisantes, de nouveaux **modèles de traitement** émergent :
  - **Traitement en batch** : Regroupe les données et les traite par lots (ex. MapReduce sur Hadoop).
  - **Traitement en flux (streaming)** : Analyse des données en temps réel dès qu'elles sont générées (ex. Apache Kafka, Apache Spark Streaming).
- Exemple : La détection des fraudes bancaires nécessite un traitement **instantané** pour bloquer une transaction suspecte avant qu'elle ne soit validée.

#### 6. Evolution des usages

- Les données sont exploitées pour de **nouvelles applications** comme :
  - **La personnalisation** des services (Netflix recommande des films basés sur vos préférences).
  - **La prévention des pannes** (les capteurs industriels détectent les anomalies avant une panne).
  - **Les alertes en temps réel** (prévisions météorologiques, notifications de trafic, surveillance des réseaux).

Remarque:

Un **cluster** est un ensemble de **serveurs interconnectés** qui fonctionnent ensemble comme un **seul système** afin d'améliorer la **performance, la disponibilité et la tolérance aux pannes**.

##### ◆ Cluster et Datacenter : Quelle relation ?

Un cluster est souvent déployé dans un **datacenter**, qui est un centre physique regroupant un grand nombre de **serveurs, systèmes de stockage et équipements réseau**.

- **Le datacenter** est l'infrastructure qui **héberge plusieurs clusters** pour gérer des applications critiques, le stockage des données et le cloud computing.

- **Le cluster** est une **unité logique** à l'intérieur d'un datacenter, composée de serveurs travaillant ensemble pour une tâche spécifique (ex. hébergement de sites web, analyse Big Data, intelligence artificielle, etc.).

#### ◆ **Avantages d'un Cluster dans un Datacenter**

- ✓ Scalabilité** : Possibilité d'ajouter de nouveaux serveurs pour augmenter la puissance de calcul.
- ✓ Haute disponibilité** : Réduction des interruptions grâce à la redondance.
- ✓ Répartition de charge** : Optimisation des ressources pour éviter les surcharges.

 **Exemple concret :**

Les **datacenters de Google, Amazon ou Microsoft** hébergent des milliers de clusters pour garantir un accès rapide aux services comme YouTube, AWS ou Azure. 

### 3/Les 5V du Big Data

Le **Big Data** repose principalement sur **cinq dimensions clés**, appelées les **5V** :

#### 1. Volume

- La quantité de données générées est **immense et en croissance exponentielle**.
- Aujourd'hui, on parle en **Zettaoctets (ZB)** et **Yottaoctets (YB)**.
- **Exemple :**
  - En 2020, l'humanité a produit **plus de 40 Zettaoctets** de données.
  - En 2025, nous devrions atteindre **175 Zettaoctets**.

Pourquoi c'est un défi ?

- Le stockage et la gestion nécessitent des **infrastructures massives (Cloud, Datacenters, Data Lakes)**.
- Les bases de données relationnelles classiques (**SQL**) ne suffisent plus, d'où l'adoption des **bases NoSQL (MongoDB, Cassandra)**.

#### 2. Vélocité

- Le Big Data repose sur la **rapidité de génération et de traitement des données**.
- Les flux de données doivent être **analysés en temps réel** ou en quasi-temps réel.
- **Exemple :**
  - Google traite plus de **100 000 requêtes par seconde**.
  - Les systèmes de **trading algorithmique** prennent des décisions en millisecondes.

Comment gérer cette vitesse ?

**✓ Parallélisation des traitements :**

- Utilisation de **frameworks distribués** comme **Hadoop (MapReduce)** et **Spark** pour exécuter des tâches en parallèle.
- ✓ **Streaming de données :**
- Outils comme **Apache Kafka** et **Apache Flink** pour traiter les données en continu.

### 3. Variété

- Les données du Big Data proviennent de **sources multiples** et existent sous **différentes formes**.
- **Types de données :**
  - **Structurées**  : Bases de données relationnelles (ex. MySQL).
  - **Semi-structurées**  : JSON, XML, logs.
  - **Non-structurées**  : Images, vidéos, réseaux sociaux, IoT.

**Pourquoi c'est un défi ?**

- Les **modèles relationnels classiques** ne sont pas adaptés à cette diversité.
- Il faut utiliser des **bases NoSQL**, des **data lakes**, et des outils spécialisés en analyse d'image et de texte.

### 4. Valeur

- Les données doivent générer de la **valeur exploitable**.
- Une **bonne gestion des données** permet de prendre de meilleures décisions et d'optimiser les performances.

**Exemples de valeur créée :**

- **Stratégique** : Amélioration de la prise de décision grâce à l'analyse de données.
- **Opérationnelle** : Automatisation des tâches et optimisation des ressources.
- **Business** : Détection de nouvelles opportunités et tendances du marché.

### 5. Véracité

- Toutes les données ne sont pas fiables : certaines peuvent être **erronées, incomplètes ou biaisées**.
- Une **mauvaise qualité des données** peut entraîner de **mauvaises décisions**.

**Solutions pour garantir la véracité :**

- ✓ Nettoyage des données et suppression des doublons.
- ✓ Validation via des **algorithmes de détection d'anomalies**.
- ✓ Vérification des sources pour éviter les **fausses informations**.

**Remarque : Adoption d'un Plan Big Data** 

 **Les trois premiers V (Volume, Vélocité, Variété) sont les plus critiques pour adopter une stratégie Big Data.**

Si une organisation génère **d'énormes volumes de données**, avec une **grande diversité de formats**, et qu'elle doit les **traiter rapidement**, alors **elle doit mettre en place un plan Big Data**.

Cela implique :

-  **Des infrastructures adaptées** (Hadoop, Cloud, Data Lakes).
-  **Des technologies performantes** pour le traitement parallèle et en temps réel.
-  **Une organisation des données efficace** pour maximiser leur valeur et leur véracité.

 **Conclusion** : Une entreprise qui respecte ces **5V** peut réellement exploiter la puissance du Big Data et en tirer un **avantage concurrentiel majeur**.

## 4/Les débuts du Big Data

L'émergence du **Big Data** a nécessité de nouveaux modèles de traitement et de stockage adaptés aux **données massives**. Deux technologies phares ont marqué ces débuts :

### 1. MapReduce

MapReduce est un **modèle de programmation parallèle** développé par **Google** pour traiter de **grandes quantités de données** sur des clusters de machines en **mode maître-esclave**.

- **Principe :**
  1. **Phase Map**  : Le **nœud maître (Master)** divise les données en plusieurs **fragments** (splits) et les assigne aux **nœuds esclaves (Workers)** pour un traitement parallèle.
  2. **Phase Reduce**  : Les résultats intermédiaires sont collectés et agrégés pour produire la sortie finale.

#### **Avantages :**

- Permet de traiter d'**immenses volumes de données** de manière distribuée.
- Réduit le temps de calcul grâce à la **parallélisation**.

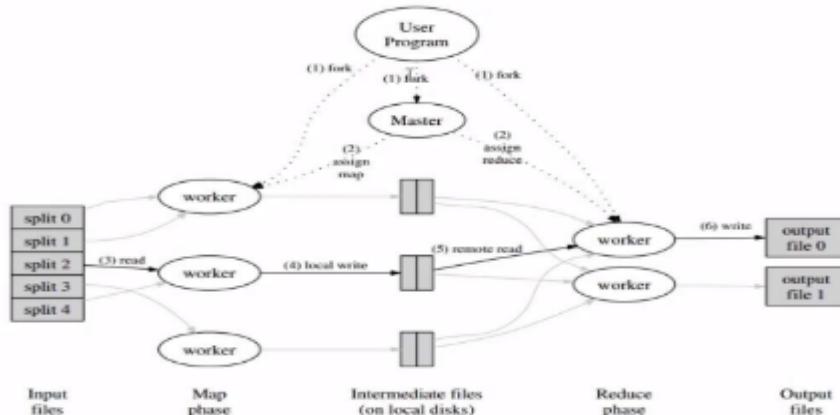
#### **Limite :**

- **Traitement par batch** (non adapté aux flux de données en temps réel).

## Les débuts du Big Data

### MapReduce

Map = Traitement de chaque élément  
Reduce = Regroupement



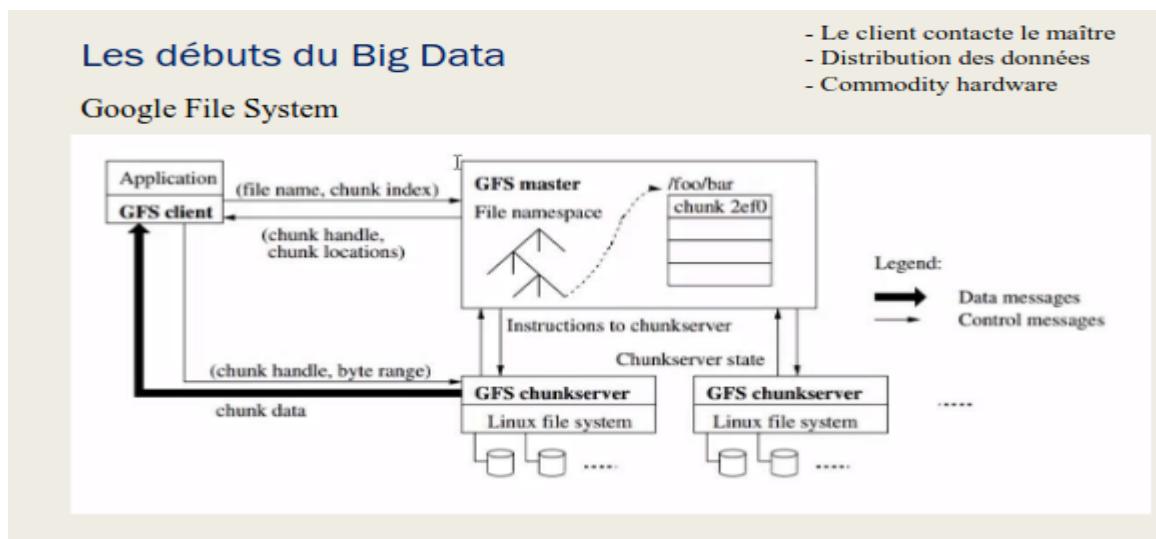
## 2. Google File System (GFS)

Le **Google File System** est un **système de stockage distribué** conçu pour gérer efficacement **de très grands fichiers** sur un **ensemble de machines standards (commodity hardware)** en architecture **maître-esclave**.

- **Architecture :**
  - Un **GFS Master (Maître)** contrôle l'organisation des fichiers, la répartition des fragments (**chunks**) et gère la répartition des requêtes.
  - Les **GFS Chunkservers (Esclaves)** stockent les **fragments de fichiers** et assurent la **réplication des données** pour éviter toute perte.
  - Le **GFS Client** contacte le **Maître** pour obtenir les informations de localisation des chunks.

### ✓ Avantages :

- Assure une **haute disponibilité et tolérance aux pannes** grâce à la **réplication des données**.
- Permet d'**évoluer facilement** en ajoutant des serveurs.



## Résumé

💡 **MapReduce et GFS** ont été les premiers piliers du Big Data, permettant de stocker et traiter efficacement des volumes massifs de données sur des clusters de serveurs en architecture maître-esclave.

- ◆ **MapReduce** : Modèle de traitement parallèle où le maître distribue les tâches aux nœuds esclaves.
- ◆ **GFS** : Système de stockage distribué où un maître gère les chunkservers esclaves pour assurer la répartition et la redondance des données.

👉 Ces innovations ont ouvert la voie aux technologies modernes comme **Hadoop**, **Spark**, et les **Data Lakes**. 🚀

## 5/L'écosystème Hadoop

Hadoop est un framework open-source qui permet de stocker et traiter de grandes quantités de données de manière distribuée sur un cluster de machines. Il repose sur une architecture modulaire avec plusieurs composants essentiels :

### 1. Stockage des données

- **HDFS (Hadoop Distributed File System)** :
  - Système de fichiers distribué conçu pour stocker des données massives.
  - Gère la réplication des fichiers pour assurer fiabilité et tolérance aux pannes.

### 2. Traitement des données

- **MapReduce v1** :
  - Modèle de programmation qui exécute des calculs en parallèle sur plusieurs machines.
  - Fonctionne en deux phases : **Map** (traitement des données) et **Reduce** (agrégation des résultats).

- **YARN (Yet Another Resource Negotiator)** :
  - Gestionnaire de ressources qui permet de **lancer et coordonner plusieurs traitements en parallèle**.
  - Remplace **MapReduce v1** dans Hadoop 2 pour une meilleure **efficacité et flexibilité**.

### 3. Interrogation des données

- **Hive** :
  - Outil qui permet d'exécuter des **requêtes SQL sur Hadoop** via **HiveQL** (un langage proche de SQL).
  - Utile pour les **analystes de données** habitués aux bases relationnelles.
- **Pig** :
  - Langage de script **haut niveau** (Pig Latin) pour manipuler les **données non structurées** dans Hadoop.
  - Idéal pour les **développeurs et data engineers** qui veulent traiter des volumes massifs de données sans SQL.

### 4. Base de données NoSQL

- **HBase** :
  - Base de données **NoSQL** conçue pour stocker des **grandes quantités de données en temps réel** sur HDFS.
  - Inspirée de **Google Bigtable**, elle permet un **accès rapide aux données** contrairement à Hive/Pig qui traitent en batch.

### 5. Gestion et planification

- **Zookeeper** :
  - Service centralisé qui assure la **coordination et la synchronisation** des composants Hadoop.
  - Utilisé pour la **gestion des clusters** et des configurations.
- **Oozie** :
  - Outil de **planification de workflows** pour automatiser l'exécution des **traitements Big Data**.
  - Permet de **chaîner plusieurs jobs** (ex : d'abord un job MapReduce, puis un job Hive).

### 6. Liaison avec les bases relationnelles

- **Sqoop** :
  - Outil permettant de **transférer des données** entre **Hadoop et les bases de données relationnelles (SGBDR)** comme **MySQL, PostgreSQL, Oracle**.
  - Facilite l'**import/export des données** entre le monde **Big Data** et les **bases traditionnelles**.

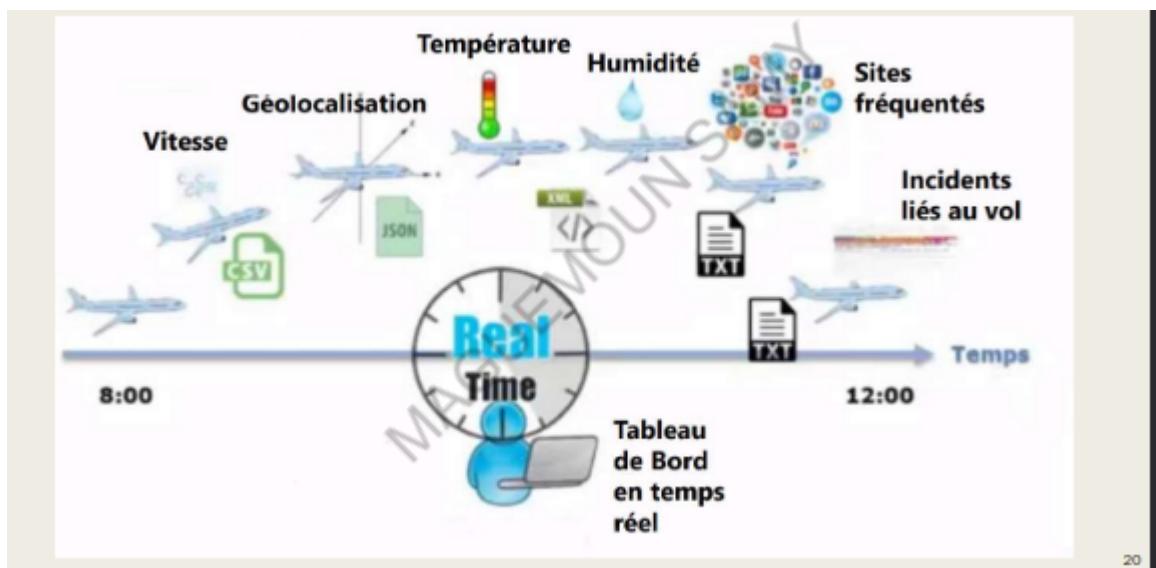
## Résumé

💡 L'écosystème Hadoop est un ensemble de technologies permettant de stocker, traiter et interroger efficacement des **volumes massifs de données** de manière distribuée.

- ✓ HDFS pour le stockage.
- ✓ MapReduce / YARN pour le traitement.
- ✓ Hive / Pig pour les requêtes.
- ✓ HBase pour le stockage NoSQL.
- ✓ Oozie, Zookeeper et Sqoop pour la gestion et l'intégration.

♦ Grâce à cet écosystème, Hadoop est devenu une référence incontournable pour les projets Big Data ! 🚀

## 6/Exemple d'application du Big Data en temps réel dans l'aviation ✈️



Cette image illustre un **exemple concret d'application du Big Data** dans le secteur aérien, notamment pour la **surveillance des vols en temps réel**.

### 1. Données collectées

L'aviation moderne génère un **volume massif de données** provenant de plusieurs sources en temps réel :

- **Vitesse et Géolocalisation (GPS)** ⚡
- **Température et Humidité** 🌡️
- **Sites fréquentés par les passagers** 🌎
- **Incidents liés au vol** ⚠️

Ces données sont stockées sous différents formats (CSV, JSON, XML, TXT) et doivent être traitées efficacement.

## 2. Pourquoi le Big Data est nécessaire ? 🤔

L'analyse de ces **données hétérogènes et volumineuses** nécessite une **infrastructure adaptée**, c'est-à-dire un système capable de :

- ✓ **Stocker et traiter des données en continu (temps réel).**
- ✓ **Corréler** les informations pour détecter des **problèmes** (ex : conditions météorologiques affectant un vol).
- ✓ **Générer des alertes et des rapports** en temps réel pour les compagnies aériennes.

Des technologies comme **Hadoop, Spark, Kafka** permettent de traiter ces flux de données massives en **temps réel**.

## 3. Objectif : Un Tableau de Bord en Temps Réel 📊

Les données collectées sont **affichées en temps réel** dans un **tableau de bord interactif**.

Cela permet aux compagnies aériennes et aux contrôleurs aériens de :

- 🚀 **Améliorer la sécurité des vols** en détectant rapidement les incidents.
- 🚀 **Optimiser l'itinéraire des avions** en fonction des conditions météorologiques.
- 🚀 **Analyser le comportement des passagers** pour une expérience plus personnalisée.

## Conclusion 🎯

L'aviation moderne repose sur **l'analyse en temps réel de données massives**, ce qui **justifie l'utilisation du Big Data**. Grâce à des outils comme **Hadoop, Spark et les bases de données NoSQL**, il est possible d'exploiter efficacement ces données pour améliorer la **sécurité, la gestion des vols et l'expérience passager**.

## Chapitre 2: Fondements de l'environnement hadoop

### 1/Architectures logicielles et leur lien avec les systèmes de fichiers distribués

L'**architecture logicielle** définit la manière dont les composants d'un système sont organisés et interagissent. Il existe plusieurs types d'architectures, dont certaines sont utilisées dans les **systèmes de fichiers distribués** comme HDFS.

#### 1. Architecture 1-Tier, 2-Tier et 3-Tier

Ces architectures sont utilisées principalement pour les **applications logicielles**, mais certaines peuvent être adaptées aux **systèmes de stockage distribués**.

- **1-Tier (Monolithique)**
  - Tous les composants (base de données, interface utilisateur, logique métier) sont regroupés dans un **même environnement**.
  - **✗ Pas adaptée aux fichiers distribués**, car tout est centralisé.
- **2-Tier (Client-Serveur)**
  - Séparation entre **le client** (interface utilisateur) et **le serveur** (base de données et logique métier).
  - Exemple : Une base **SQL classique** comme MySQL.
  - **✗ Limité pour les fichiers distribués**, car un seul serveur devient un **goulot d'étranglement**.
- **3-Tier (Multi-couches)**
  - Ajoute une couche intermédiaire (**middleware**), séparant la **base de données, la logique métier et l'interface utilisateur**.
  - **✓ Peut être adaptée aux systèmes distribués** si la couche middleware gère plusieurs serveurs de stockage (ex. Hadoop avec YARN).

#### 2. Architecture Peer-to-Peer (P2P)

Dans une architecture **Peer-to-Peer**, chaque nœud joue à la fois le rôle de **client et de serveur**.

- **Caractéristiques du P2P :**
  - Il n'y a **pas de serveur central** : chaque machine partage ses ressources.
  - Utilisé dans des **réseaux distribués** comme **BitTorrent**.
  - **✓ Peut être utilisé pour un système de fichiers distribué**, mais **pas adapté** pour Hadoop (qui fonctionne avec un **maître-esclave** via HDFS).

#### 3. Systèmes de fichiers distribués et architectures adaptées

 **Un système de fichiers distribué** (DFS - Distributed File System) permet le stockage et l'accès à des fichiers sur plusieurs machines.

- **✓ HDFS (Hadoop Distributed File System)** repose sur une **architecture maître-esclave (3-tier adaptée)**.

- Un modèle P2P pur est rarement utilisé pour les systèmes de fichiers distribués en entreprise, car il manque de contrôle centralisé.

## 4. Scalabilité dans les systèmes distribués

La **scalabilité** désigne la capacité d'un système à **s'adapter à une augmentation de la charge** (plus de données, plus de requêtes).

- **Scalabilité verticale**
  - Ajouter plus de puissance à une seule machine (ex. plus de RAM, CPU).
  - Limité, car une seule machine ne peut pas croître indéfiniment.
- **Scalabilité horizontale**
  - Ajouter **plusieurs machines** pour partager la charge (ex. Hadoop).
  - Adapté aux systèmes distribués comme HDFS.

Hadoop et HDFS sont conçus pour la **scalabilité horizontale**, ce qui permet d'ajouter plus de DataNodes pour gérer davantage de données sans perte de performance.

## Conclusion

- HDFS et Hadoop utilisent une architecture maître-esclave (3-Tier) et sont conçus pour la **scalabilité horizontale**.
- Le modèle P2P est décentralisé, mais peu adapté aux fichiers distribués en entreprise.
- La **scalabilité horizontale** est clé pour les systèmes Big Data comme Hadoop, Spark et NoSQL.

## 2/L'écosystème Hadoop

L'environnement Hadoop repose sur plusieurs composants :

- **HDFS (Hadoop Distributed File System)** : Système de fichiers distribué permettant de stocker des données massives.
- **MapReduce** : Modèle de programmation pour traiter de grands volumes de données en parallèle.
- **YARN (Yet Another Resource Negotiator)** : Gestionnaire de ressources qui organise l'exécution des tâches Hadoop.

## 3/Systèmes de fichiers

Un **système de fichiers** est une structure permettant de gérer l'organisation des données sur un disque.

**Exemples** : FAT16, FAT32, NTFS, Ext4, etc.

## 4/HDFS : Système de gestion de fichiers distribué

Le Hadoop Distributed File System (HDFS) est le système de gestion de fichiers distribué utilisé par Hadoop pour stocker de grandes quantités de données de manière fiable, évolutive et tolérante aux pannes.

## 1. Architecture de HDFS

HDFS suit une architecture maître-esclave et est composé de plusieurs éléments clés :

### 1.1. NameNode (Maître)

- **Rôle** : Gère la métadonnée du système de fichiers (informations sur la structure des fichiers et leur emplacement).
- Ne stocke pas les données elles-mêmes, mais sait où chaque bloc est stocké dans le cluster.
- Responsable de la gestion des accès aux fichiers et de la répartition des blocs.

### 1.2. DataNodes (Esclaves)

- Stockent réellement les blocs de données.
- Exécutent les opérations de lecture et écriture demandées par les clients.
- Envoient régulièrement des heartbeats (signaux de vie) au NameNode pour signaler leur état.

### 1.3. Secondary NameNode

- Sauvegarde les métadonnées du NameNode à intervalles réguliers pour éviter la perte de données en cas de panne.
- Ne remplace pas le NameNode, mais aide à récupérer l'état du cluster après un redémarrage.

#### Exemple de fonctionnement :

1. Lorsqu'un client stocke un fichier dans HDFS, il est découpé en blocs et stocké sur plusieurs DataNodes.
2. Le NameNode garde une trace de l'emplacement des blocs.
3. Lorsqu'un client veut accéder à un fichier, il interroge le NameNode, qui lui indique où récupérer les blocs.

## 2. Caractéristiques principales de HDFS

### 2.1. Distribution des données

- HDFS divise chaque fichier en blocs de données et les répartit sur plusieurs nœuds du cluster.
- Cela permet d'assurer une scalabilité horizontale, car on peut ajouter plus de DataNodes pour stocker plus de fichiers.

## 2.2. Taille des blocs

- Par défaut, un bloc HDFS fait 128 Mo.
- Peut être ajusté à 64 Mo, 256 Mo, 512 Mo, en fonction des besoins.
- Plus la taille des blocs est grande, moins il y aura de métadonnées à gérer par le NameNode.

## 2.3. RéPLICATION DES DONNÉES

- Chaque bloc est dupliqué sur plusieurs DataNodes.
- Par défaut, facteur de réPLICATION = 3 (chaque bloc est copié sur 3 nœuds différents).
- Si un DataNode tombe en panne, HDFS recrée automatiquement des copies sur d'autres nœuds.

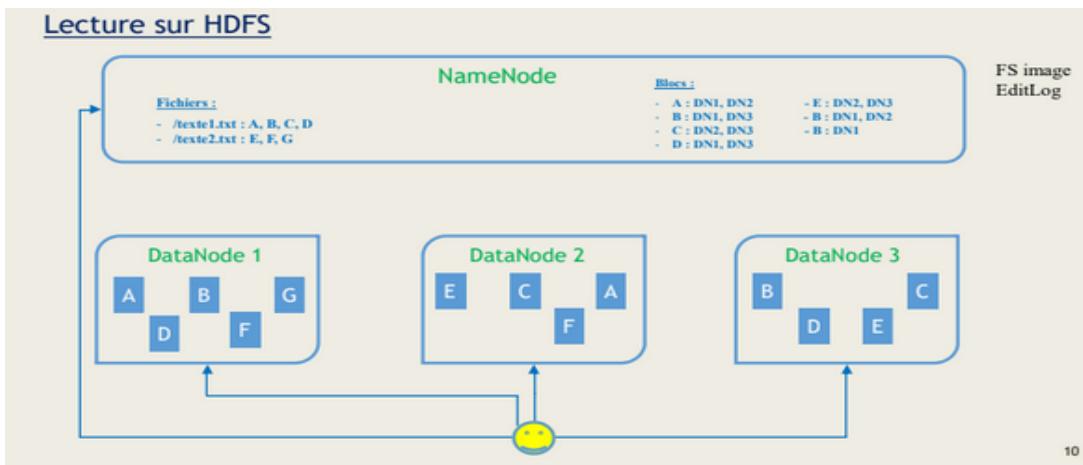
## 2.4. Haute disponibilité (High Availability)

- Grâce à la réPLICATION, si un nœud tombe en panne, les données restent accessibles sur un autre.
- Permet d'éviter la perte de données et d'assurer une continuité de service.

# 3. Lecture et écriture sur HDFS

## 3.1. Lecture sur HDFS

1. Le client demande au NameNode l'emplacement des blocs du fichier.
2. Le NameNode fournit la liste des DataNodes contenant les blocs.
3. Le client contacte directement les DataNodes pour récupérer les blocs et reconstruire le fichier.
4. Le NameNode maintient deux fichiers essentiels :
  - **FSImage** : Contient une image complète des métadonnées du système de fichiers.
  - **EditLog** : Journal des modifications récentes qui seront appliquées à FSImage.



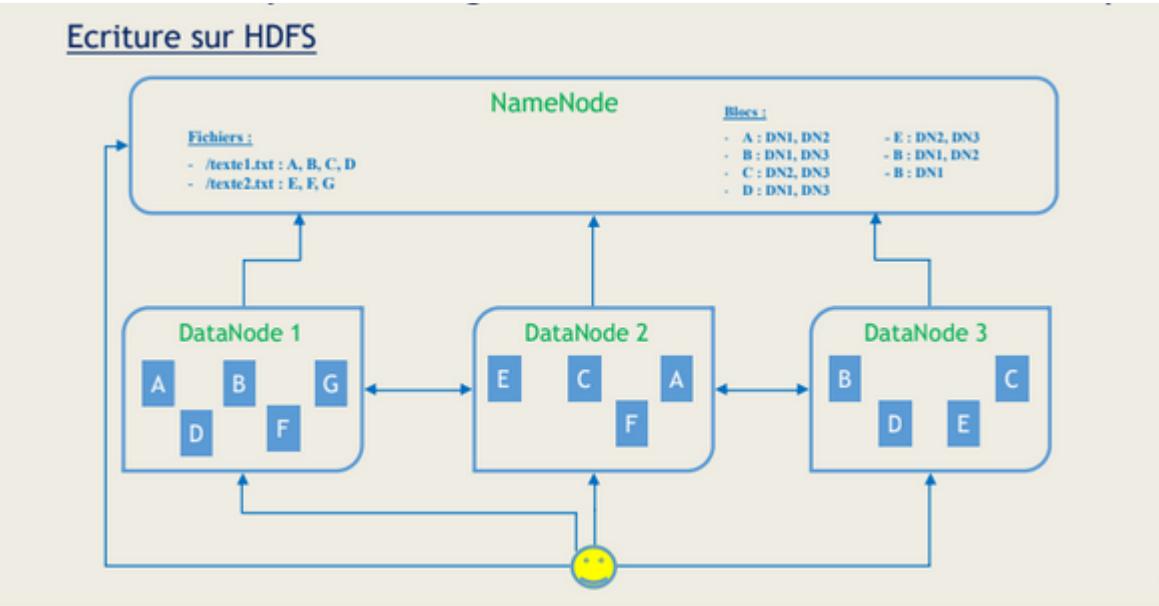
L'image représente un schéma de répartition des fichiers dans un système distribué.  
Voici les points clés :

1. **Un ordinateur client (1)** : Il envoie une requête à un serveur central ou un gestionnaire de fichiers.
2. **Un serveur central (2)** : Il agit comme un point d'entrée pour le stockage ou la récupération des fichiers.
3. **Distribution des fichiers (3)** : Les fichiers sont ensuite fragmentés et distribués sur plusieurs serveurs de stockage.

L'objectif ici est d'optimiser la disponibilité et la redondance des fichiers, afin d'améliorer la tolérance aux pannes et la rapidité d'accès aux données.

### 3.2. Écriture sur HDFS

1. Le client contacte le NameNode pour signaler l'ajout d'un fichier.
2. Le NameNode attribue les DataNodes où seront stockés les blocs.
3. Le client envoie les blocs aux DataNodes, qui les répliquent sur d'autres nœuds.
4. Une fois terminé, le NameNode met à jour ses métadonnées, enregistre les changements dans EditLog, puis fusionne avec FSImage lors de points de sauvegarde.



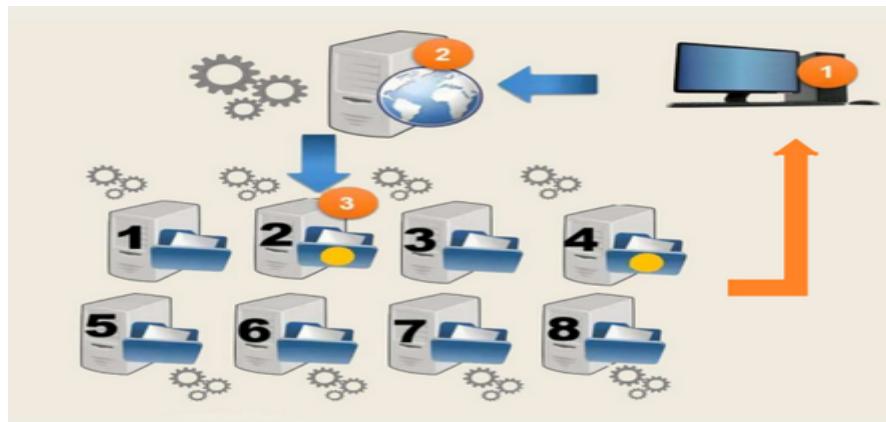
L'image représente le processus d'écriture des fichiers dans le HDFS (Hadoop Distributed File System). Explication :

- **NameNode** : Il est responsable de la gestion des métadonnées (répartition des blocs, emplacement des fichiers, etc.).
- **DataNodes (1, 2, 3)** : Ils stockent réellement les blocs de données. Chaque fichier est divisé en plusieurs blocs, qui sont répliqués sur différents nœuds pour assurer la fiabilité.
- **Exemple de fichiers et de blocs :**
  - `/test1.txt` est découpé en blocs A, B, C, D, stockés sur différents DataNodes.
  - `/test2.txt` est découpé en blocs E, F, G et réparti également.

### 👉 Pourquoi HDFS ?

- Il permet de stocker de très grandes quantités de données en répartissant la charge.
- Il assure la redondance (chaque bloc est répliqué sur plusieurs DataNodes).
- Il est scalable et robuste face aux pannes.

## HDFS:Le système de gestion distribué des fichiers Hadoop:



L'image illustre le fonctionnement d'un système de fichiers distribué comme HDFS, basé sur une architecture maître-esclave.

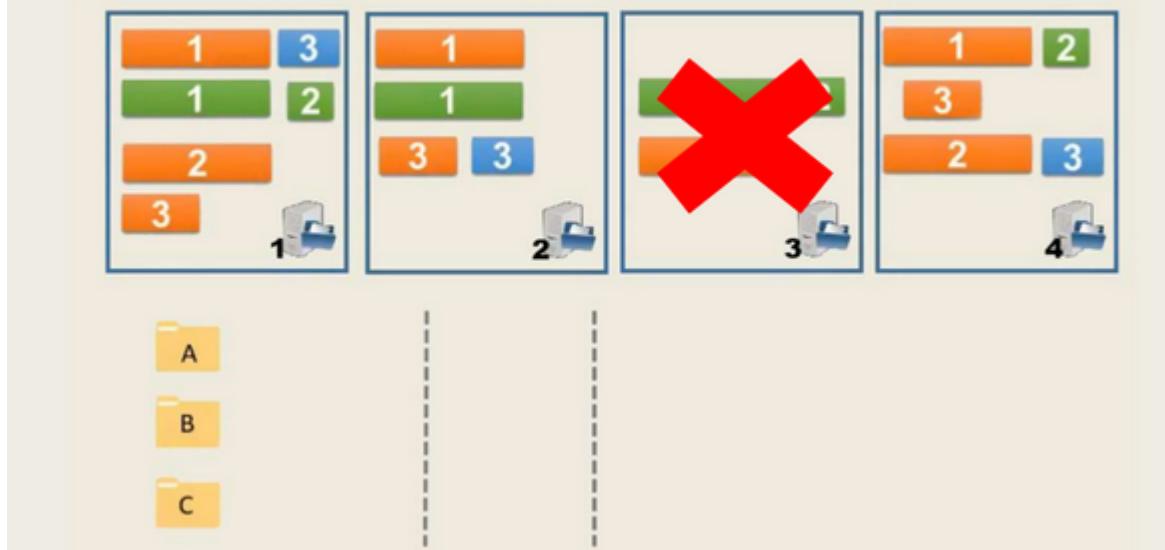
1. Requête utilisateur : Un client envoie une demande pour stocker ou récupérer un fichier.
2. Gestion par le NameNode : Le serveur principal (NameNode) reçoit la requête et gère l'emplacement des fichiers dans le cluster.
3. Stockage sur les DataNodes : Les fichiers sont découpés en blocs, répliqués sur plusieurs DataNodes pour assurer la tolérance aux pannes et la disponibilité.
4. Scalabilité horizontale : L'infrastructure peut évoluer en ajoutant de nouveaux serveurs, permettant de gérer plus de données sans surcharger une seule machine.

**Conclusion :** Ce modèle garantit fiabilité, tolérance aux pannes et évolutivité, ce qui en fait une solution clé pour le Big Data.

## 4. Le système de fichiers Hadoop FS

- Hadoop FS est l'interface qui permet d'interagir avec HDFS et d'autres systèmes de fichiers.
- **Types de systèmes de fichiers pris en charge :**
  - **HDFS (Hadoop Distributed File System)** : Stockage natif d'Hadoop conçu pour distribuer les fichiers sur plusieurs nœuds.
  - **Local FS** : Système de fichiers local utilisé pour les tests.
  - **Amazon S3 FS** : Intégration d'Hadoop avec Amazon S3 pour un stockage cloud évolutif.
  - **HFTP FS** : Utilisé pour transférer des fichiers entre clusters Hadoop distants via HTTP.
  - **S3A FS** : Version optimisée de S3 FS avec une meilleure gestion des permissions et performances accrues.

## HDFS : Le système de gestion distribué des fichiers Hadoop



L'image représente HDFS (Hadoop Distributed File System) et la gestion des fichiers dans un environnement distribué.

- Chaque couleur représente un bloc de données.
- Chaque numéro représente une réplique de ces blocs stockée sur différents nœuds (serveurs).
- La croix rouge sur le troisième serveur indique une panne ou une défaillance.
- Malgré cette panne, les fichiers restent accessibles car HDFS réplique les données sur plusieurs nœuds, garantissant tolérance aux pannes et fiabilité.

### → Concept clé : Scalabilité et Résilience

- **Scalabilité** : Capacité d'ajouter des serveurs pour augmenter la capacité de stockage et le traitement des données.
- **Résilience** : Même en cas de panne d'un serveur, les données restent accessibles grâce aux copies stockées ailleurs.

Ces concepts sont fondamentaux pour le Big Data et expliquent pourquoi Hadoop est largement utilisé pour gérer de gros volumes de données de manière efficace et sécurisée.



## 5.Les Daemons d'Hadoop HDFS

HDFS repose sur trois types de daemons (processus en arrière-plan) pour assurer son fonctionnement :

### ① NameNode (NN) – Nœud Maître

- Gère les métadonnées du système de fichiers (structure des fichiers, emplacements des blocs, permissions).
- Ne stocke pas les données elles-mêmes, mais garde une cartographie des blocs

répartis sur les DataNodes.

Stocke ses métadonnées sur disque sous forme de fsimage et enregistre les modifications dans Edits.

## 2 Secondary NameNode (SNN) – Sauvegarde du NameNode

Ne remplace pas le NameNode mais aide à sa récupération en cas de panne.  
 Fusionne régulièrement les fichiers fsimage et Edits pour éviter une surcharge mémoire.

## 3 DataNode (DN) – Nœuds Esclaves

Stockent réellement les blocs de données et exécutent les opérations de lecture/écriture.  
 Assurent la réPLICATION et la suppression des blocs sous l'ordre du NameNode.  
 Envoient un heartbeat (pulsion) toutes les 3 secondes au NN pour signaler leur bon fonctionnement.

♦ En résumé :

- NameNode → Gère la structure et l'emplacement des fichiers.
- Secondary NameNode → Sauvegarde les métadonnées pour éviter la perte de données.
- DataNode → Stocke et manipule les données sous forme de blocs, tout en restant sous l'autorité du NameNode.

Ces daemons assurent la scalabilité et la fiabilité du stockage distribué dans Hadoop !

## 6. Le paradigme MapReduce

### Explication du paradigme MapReduce avec l'exemple de WordCount

MapReduce est un modèle de programmation utilisé pour traiter de grandes quantités de données en parallèle sur un cluster de machines. Il se compose de deux phases principales : **Map** et **Reduce**. L'exemple ci-dessus illustre son fonctionnement à travers le **compte de mots** (WordCount).

#### 1 Étape 1 : Phase MAP

**Objectif** : Transformer les données d'entrée en paires (clé, valeur).

**Processus** :

- L'entrée est divisée en plusieurs fragments (Split).
- Chaque fragment est traité en parallèle par une fonction **Map** qui extrait chaque mot et lui associe la valeur 1.

♦ **Exemple :**

Entrée (fichier texte) :

A, B, C

C, A, C

C, B, B

Transformation par la fonction Map :

(A, 1), (B, 1), (C, 1)

(C, 1), (A, 1), (C, 1)

(C, 1), (B, 1), (B, 1)

## 2 Étape 2 : Phase SHUFFLE & SORT

**Objectif** : Regrouper toutes les paires ayant la même clé.

 **Processus** :

- Les paires issues de la phase **Map** sont réorganisées (Shuffle).
- Les valeurs sont regroupées par clé (Sort).

♦ **Exemple** :

(A, [1,1])

(B, [1,1,1])

(C, [1,1,1,1])

## 3 Étape 3 : Phase REDUCE

**Objectif** : Effectuer un traitement sur les données regroupées et produire le résultat final.

 **Processus** :

- La fonction **Reduce** applique une agrégation sur les valeurs associées à chaque clé.
- Dans notre cas, elle additionne les valeurs pour compter l'occurrence de chaque mot.

♦ **Exemple** :

(A, 2)

(B, 3)

(C, 4)

#### Résultat final :

A : 2 fois

B : 3 fois

C : 4 fois

#### Conclusion

 **MapReduce** est un modèle efficace pour traiter de **grandes quantités de données en parallèle**.

- La **fonction Map** divise le travail en sous-tâches indépendantes.
- La **phase Shuffle & Sort** regroupe les résultats.
- La **fonction Reduce** agrège les valeurs pour produire le résultat final.

 **Avantage** : Idéal pour le Big Data (ex. : recherche de tendances, indexation Web, analyse de logs). 

## 7.Jobs (daemons) d'hadoop MapReduce

Hadoop MapReduce utilise deux types de **daemons** pour gérer les tâches :

### ① JobTracker (Gestionnaire de tâches) :

- Gère l'exécution des tâches **MapReduce**.
- Reçoit le **programme** (fichier .jar) et les **données d'entrée** stockées sur **HDFS**.
- Détermine où stocker les **résultats**.
- Communique avec le **NameNode** pour localiser les données.
- Il y a **un seul JobTracker** dans le cluster.

### ② TaskTracker (Exécuteur des tâches) :

- Exécute les **tâches Map et Reduce** assignées par le JobTracker.

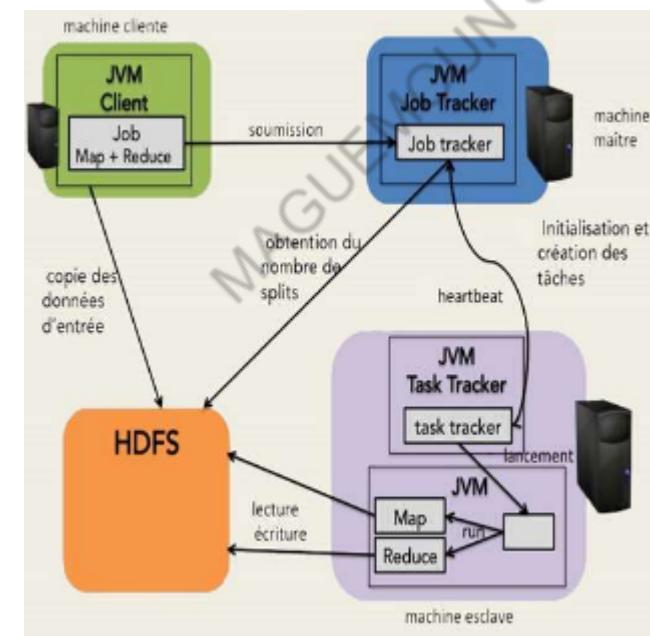
- Accède aux **blocs de données** sur **HDFS**.
- Fonctionne sur plusieurs machines du cluster et **communique avec le JobTracker**.

✓ **Résumé :**

Le **JobTracker** gère l'ensemble du processus et distribue les tâches aux **TaskTrackers**, qui exécutent les calculs sur les données stockées dans **HDFS**.

## 8.Exécution d'un job dans hadoop MapReduce

- 1 Le client copie ses données sur **HDFS**.
- 2 Il soumet un **job** (fichier **.jar** + noms des fichiers d'entrée et sortie) au **JobTracker**.
- 3 Le **JobTracker** interroge le **NameNode** pour connaître l'emplacement des blocs de données.
- 4 Il assigne ensuite les tâches aux **TaskTrackers** les plus proches des données.
- 5 Les **TaskTrackers** envoient régulièrement des "**heartbeats**" au **JobTracker** pour informer de leur statut.
- 6 Le **JobTracker** leur envoie les instructions **Map/Reduce** à exécuter sur les blocs de données.
- 7 Une fois toutes les tâches **terminées**, le job est considéré comme réussi.



### Explication de l'image :

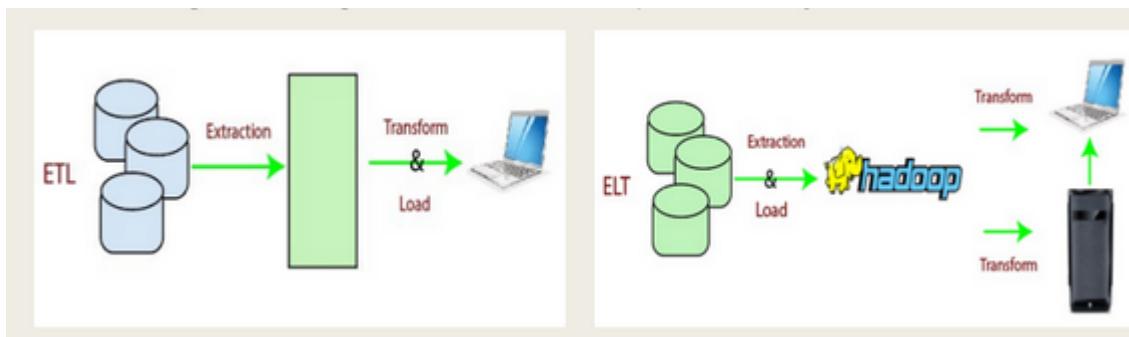
- **JVM Client** (en vert) : Machine cliente qui soumet un **job MapReduce**.
- **JVM Job Tracker** (en bleu) :
  - Situé sur la **machine maître**.

- Il reçoit le job, récupère l'emplacement des **données sur HDFS** et attribue les tâches aux **TaskTrackers**.
- **HDFS** (en orange) :
  - Stocke les **données d'entrée** et les **résultats** du traitement.
  - Permet la lecture et l'écriture des données par les **TaskTrackers**.
- **JVM Task Tracker** (en violet) :
  - Situé sur des **machines esclaves**.
  - Exécute les **tâches Map et Reduce** envoyées par le **JobTracker**.
  - Communique avec le **JobTracker** via des **heartbeats** pour signaler son état.

 **Conclusion :** Hadoop MapReduce exécute un traitement distribué en parallèle en optimisant la proximité des données et en répartissant la charge de calcul sur plusieurs nœuds.

## 9. Extraction, Transformation et Chargement (ETL & ELT)

- **ETL (Extract, Transform, Load)** : Ce processus suit trois étapes :
  - **Extract** : Extraction des données depuis diverses sources (bases de données, fichiers logs, API, etc.).
  - **Transform** : Nettoyage, structuration et enrichissement des données pour les rendre exploitables.
  - **Load** : Chargement des données transformées dans un entrepôt de données pour analyse.
- **ELT (Extract, Load, Transform)** : Variante où les données sont d'abord chargées avant d'être transformées dans un environnement Big Data comme Hadoop.
  - Permet d'exploiter la puissance de calcul de Hadoop pour transformer les données après stockage.
- HDFS facilite ces processus en servant de zone de stockage intermédiaire pour traiter de grandes quantités de données brutes avec MapReduce, Hive ou Spark.



L'image explique la différence entre ETL (Extract, Transform, Load) et ELT (Extract, Load, Transform), qui sont deux processus de traitement des données dans un contexte de Big Data.

- **ETL :**
  1. **Extraction (Extract)** : Récupération des données depuis une source (base de données, fichiers, API, etc.).
  2. **Transformation (Transform)** : Nettoyage, formatage et structuration des données avant leur stockage.
  3. **Changement (Load)** : Insertion des données transformées dans un système de stockage final.
- **ELT :**
  1. **Extraction (Extract)** : Récupération des données brutes.
  2. **Changement (Load)** : Stockage direct des données brutes dans un système comme Hadoop.
  3. **Transformation (Transform)** : Traitement des données directement dans le système cible (ex : via Hadoop ou un Data Warehouse).

→ **Déférence clé :**

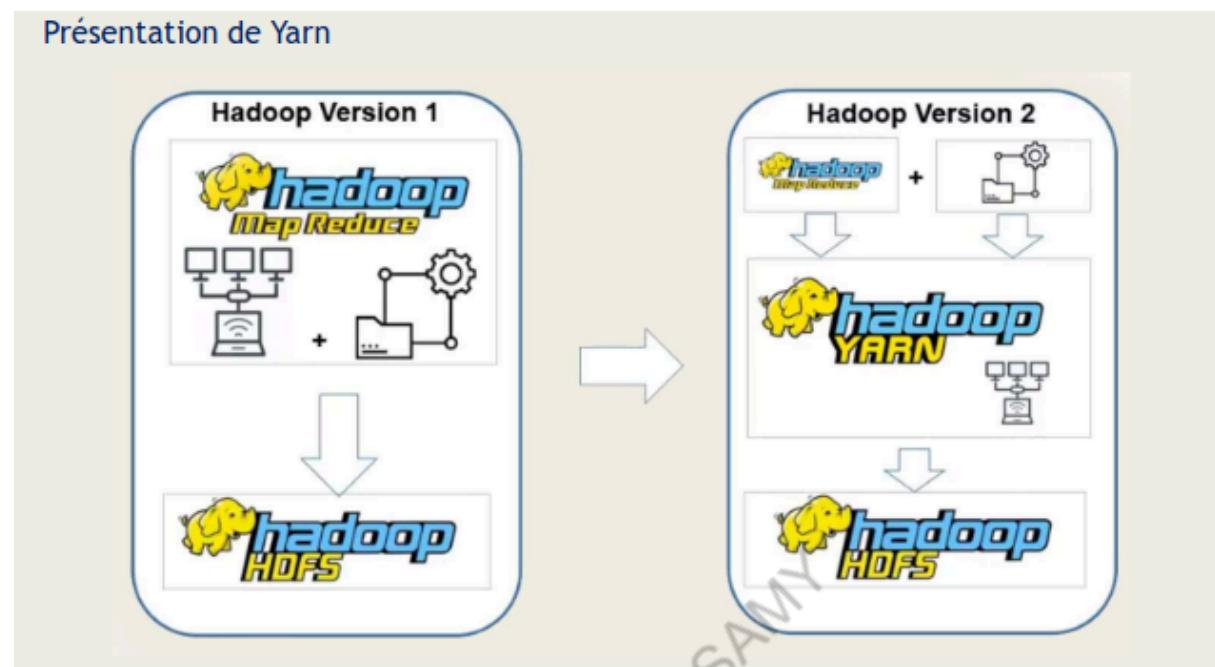
- ETL transforme les données avant le stockage.
- ELT stocke d'abord les données et les transforme ensuite, ce qui est plus adapté aux grandes quantités de données et aux systèmes distribués comme Hadoop.

## 10. Problématiques de MapReduce :

- ① **Surcharge du JobTracker** : Trop de tâches à gérer (ordonnancement, monitoring, allocation des ressources).
- ② **Nombre limité de nœuds** : Les **DataNodes** et **TaskTrackers** sont restreints par la capacité du cluster.
- ③ **Répartition inefficace des tâches** : Les slots **Map/Reduce** sont définis à l'avance, ce qui peut entraîner une mauvaise utilisation des ressources.
- ④ **Single Point of Failure** : Si le **JobTracker** tombe en panne, il doit être redémarré, bloquant ainsi tout le traitement.

# 11.YARN:Gestionnaire de cluster Hadoop :

## Présentation de Yarn



## Explication de l'image

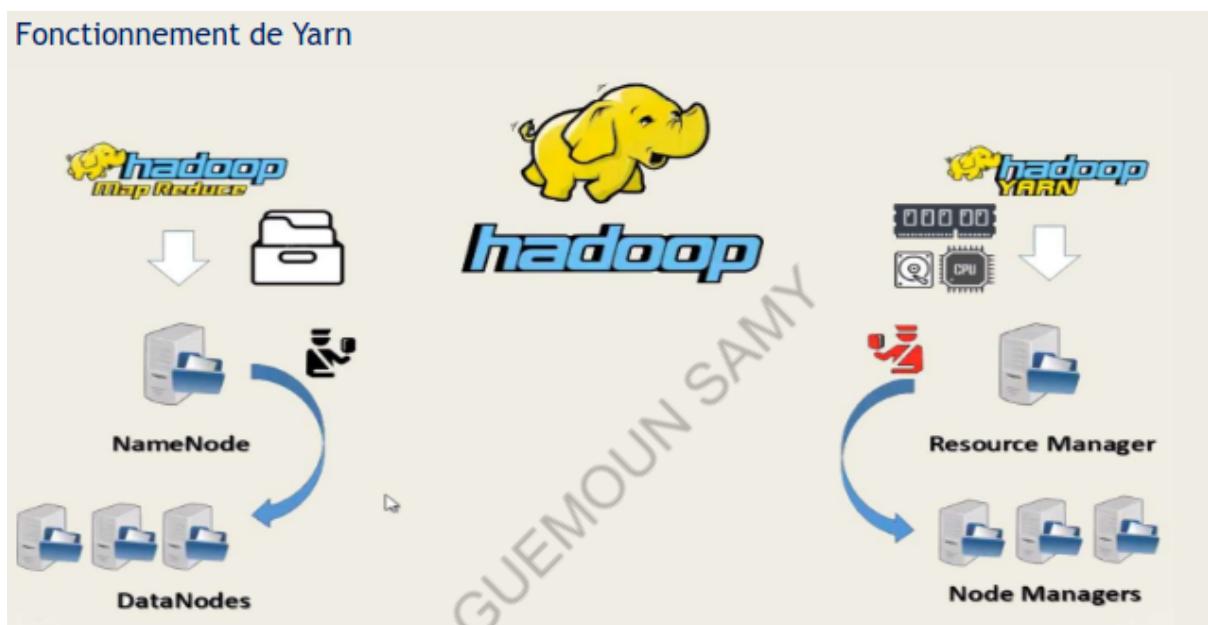
L'image illustre l'évolution de Hadoop entre **Hadoop Version 1** et **Hadoop Version 2** avec l'introduction de **YARN**.

- **Hadoop Version 1 :**
  - Utilise **MapReduce** comme unique moteur d'exécution.
  - MapReduce gère à la fois l'exécution des tâches et la gestion des ressources, ce qui surcharge le **JobTracker** et pose des problèmes de scalabilité.
  - **HDFS (Hadoop Distributed File System)** est utilisé pour stocker les données.
- **Hadoop Version 2 :**
  - Introduit **YARN (Yet Another Resource Negotiator)**, qui sépare la **gestion des ressources** et l'**exécution des tâches**.

- Permet de supporter d'autres modèles de calcul en plus de **MapReduce** (ex : Spark, Tez, etc.).
- Améliore la **scalabilité**, la **gestion des ressources** et évite le **Single Point of Failure** du JobTracker.

► **Résumé** : L'introduction de **YARN dans Hadoop v2** a permis une meilleure répartition des tâches et une utilisation plus efficace des ressources du cluster.

## Fonctionnement de Yarn



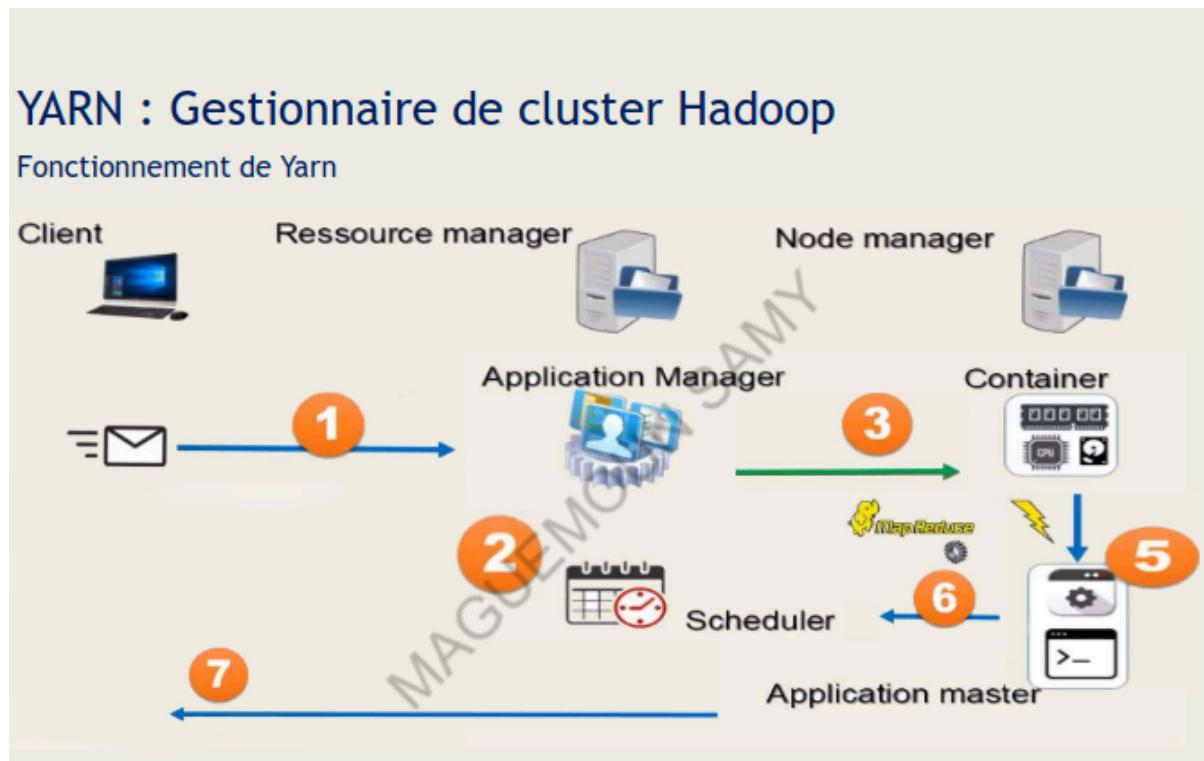
## Explication de l'image

L'image illustre le **fonctionnement de YARN** dans Hadoop, en comparant la gestion du stockage et des ressources.

- **Partie gauche (HDFS - Stockage des données) :**
  - **HDFS (Hadoop Distributed File System)** gère le stockage des données.
  - Les fichiers sont divisés en **blocs** et répartis sur plusieurs **DataNodes** sous la supervision du **NameNode**, qui garde la table des métadonnées.
- **Partie droite (YARN - Gestion des ressources) :**
  - **YARN (Yet Another Resource Negotiator)** gère les ressources du cluster.

- Le **Resource Manager** attribue les ressources aux tâches et surveille leur exécution.
- Les **Node Managers**, présents sur chaque machine, exécutent les tâches et rapportent leur état au **Resource Manager**.

► Résumé : Hadoop sépare le stockage des données (HDFS) et la gestion des ressources (YARN), permettant une exécution plus efficace des traitements distribués.



### Explication de l'image

L'image illustre le **fonctionnement de YARN en tant que gestionnaire de cluster Hadoop**, en montrant les étapes de traitement des tâches soumises par un client.

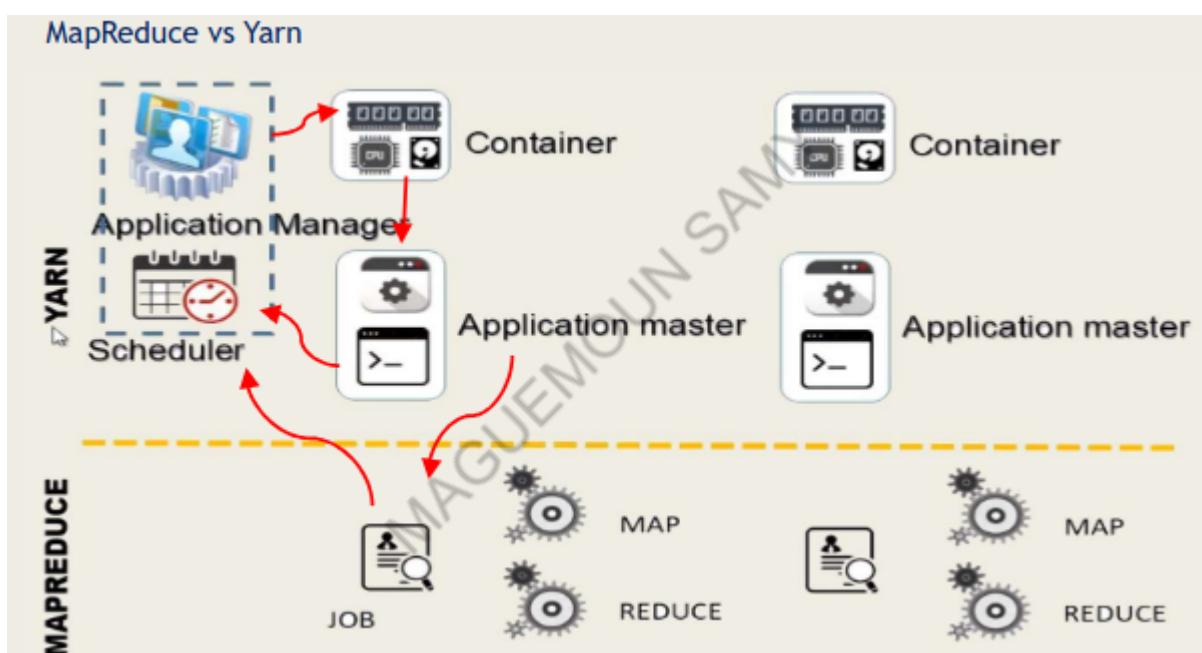
### Étapes du processus

1. **Le client soumet une tâche au Resource Manager**, qui est responsable de la gestion des ressources dans le cluster.
2. **Le Scheduler** (planificateur) du **Resource Manager** attribue les ressources nécessaires à la tâche.
3. **Un conteneur est alloué** sur un **Node Manager** pour exécuter l'Application Master.

4. L'**Application Master** est lancé sur le conteneur, il gère l'exécution de la tâche et communique avec le Resource Manager.
5. L'**Application Master demande des ressources supplémentaires** (containers) pour exécuter les différentes étapes du job (comme MapReduce).
6. Le **Scheduler** assigne ces ressources aux nœuds appropriés.
7. **Une fois le job terminé, le client récupère les résultats.**

► **Résumé** : YARN optimise la gestion des ressources en attribuant dynamiquement des conteneurs aux applications distribuées, améliorant ainsi l'efficacité de Hadoop.

## MapReduce VS Yarn



## Explication de l'image

L'image illustre la **différence entre MapReduce et YARN** en termes de gestion des ressources et d'exécution des tâches.

### Partie YARN (en haut)

- **Application Manager** et **Scheduler** (Planificateur) gèrent la distribution des ressources.

- Des **conteneurs** sont attribués pour exécuter les **Application Masters**, qui supervisent l'exécution des jobs.
- Chaque **Application Master** gère un job indépendamment, demandant des ressources aux nœuds appropriés.

### Partie MapReduce (en bas)

- Les jobs sont exécutés selon le modèle **MAP → REDUCE**.
- L'Application Master alloue les tâches de **Map** et **Reduce** aux ressources disponibles.
- Plusieurs jobs peuvent s'exécuter simultanément grâce à la gestion des ressources par YARN.

→ Résumé : YARN améliore l'exécution de MapReduce en séparant la gestion des ressources et l'exécution des jobs, permettant une meilleure scalabilité et efficacité dans un cluster Hadoop.

## 12. MapReduce en Java :

### Classe Map (Exemple : WordCount)

```
public class WordCountMapper extends MapReduceBase implements
Mapper<LongWritable, Text, Text, IntWritable> {

private final static IntWritable one = new IntWritable(1);

private Text word = new Text();

public void map (LongWritable key, Text value, OutputCollector<Text, IntWritable> collector,
Reporter reporter) throws IOException {

String line = value.toString();

 StringTokenizer st = new StringTokenizer(line, " ");

while (st.hasMoreTokens()) {

word.set(st.nextToken());

collector.collect(word, one); }

} }
```

### Explication du Code WordCountMapper

Ce code est un **Mapper** dans **Hadoop MapReduce** qui compte le nombre d'occurrences de chaque mot dans un texte. Il est écrit en Java et utilise l'ancienne API de Hadoop.

### Décomposition du Code

```
public class WordCountMapper extends MapReduceBase  
    implements Mapper<LongWritable, Text, Text, IntWritable> {
```

- **WordCountMapper** : Classe qui hérite de **MapReduceBase** et implémente l'interface **Mapper**.
- **Mapper<LongWritable, Text, Text, IntWritable>** :
- **LongWritable** : Clé d'entrée (offset de la ligne dans le fichier).
  - **Text** : Valeur d'entrée (la ligne de texte).
  - **Text** : Clé de sortie (mot unique).
  - **IntWritable** : Valeur de sortie (compteur de 1 pour chaque mot).

```
private final static IntWritable one = new IntWritable(1);
```

```
private Text word = new Text();
```

- **one** : Valeur constante (1) qui sera associée à chaque mot rencontré.
- **word** : Objet de type **Text** utilisé pour stocker les mots.

```
public void map (LongWritable key, Text value, OutputCollector<Text, IntWritable> collector,  
Reporter reporter) throws IOException {
```

- **map** : Méthode exécutée pour chaque ligne du fichier d'entrée.

### Paramètres :

- **key** : Position de la ligne dans le fichier (non utilisée ici).
- **value** : Contenu de la ligne.
- **collector** : Collecteur qui enregistre les paires (**mot**, **1**).
- **reporter** : Utilisé pour signaler la progression (non utilisé ici).

```
String line = value.toString();
```

```
StringTokenizer st = new StringTokenizer(line, " ");
```

- Convertit la ligne en chaîne de caractères.
- Utilise `StringTokenizer` pour découper la ligne en mots ( séparateur : espace " " ).

```
while (st.hasMoreTokens()) {
    word.set(st.nextToken());
    collector.collect(word, one);
}
```

- Parcourt tous les mots de la ligne.
- Chaque mot est stocké dans `word`.
- Ajoute la paire (`mot, 1`) au `collector`.

## Résumé

Ce Mapper prend un fichier texte en entrée et produit des paires (`mot, 1`).

Chaque ligne est lue, découpée en mots, et chaque mot est émis avec la valeur **1**.

Le Reducer additionnera ensuite ces valeurs pour compter les occurrences des mots.

## Classe Reducer (Exemple : WordCount)

```
public class WordCountReducer extends MapReduceBase implements Reducer<Text,
IntWritable, Text, IntWritable> {

    public void reduce (Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable>
outputCollector,
    Reporter reporter) throws IOException {

        int sum = 0;

        while (values.hasNext()) {
            sum = sum + values.next().get();
        }

        outputCollector.collect(key, new IntWritable(sum));
    }
}
```

## Explication du code WordCountReducer

Ce code représente la **classe Reducer** d'un programme **MapReduce** en Java pour **compter le nombre d'occurrences de chaque mot** (exemple classique du WordCount).

### Décomposition du Code

#### 1. Déclaration de la classe

```
public class WordCountReducer extends MapReduceBase implements Reducer<Text,  
IntWritable, Text, IntWritable> {
```

- La classe **WordCountReducer** hérite de **MapReduceBase** et implémente l'interface **Reducer<Text, IntWritable, Text, IntWritable>**.
- Elle définit un Reducer qui :
  - **Reçoit** une clé **Text** (le mot) et une liste de valeurs **IntWritable** (le nombre d'occurrences).
  - **Renvoie** une clé **Text** (le mot) et une valeur **IntWritable** (le total des occurrences).

#### 2. Méthode **reduce**

```
public void reduce (Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable>  
outputCollector, Reporter reporter) throws IOException {
```

- La méthode **reduce** prend en entrée :
  - **key** : la clé actuelle (un mot).
  - **values** : un **itérateur** contenant toutes les valeurs associées à cette clé (les 1 de la phase Map).
  - **outputCollector** : un objet pour stocker le résultat final.
  - **reporter** : utilisé pour signaler des informations d'exécution (peu utilisé ici).

#### 3. Calcul du total des occurrences

```
int sum = 0;  
  
while (values.hasNext()) {  
  
    sum = sum + values.next().get();  
  
}
```

- On initialise `sum` à 0.
- On parcourt toutes les valeurs associées à la clé en utilisant un `while`.
- On additionne ces valeurs pour obtenir le nombre total d'occurrences du mot.

#### 4. Émission du résultat

```
outputCollector.collect(key, new IntWritable(sum));
```

- Une fois le total calculé (`sum`), on enregistre le résultat avec la clé (`key`) et sa valeur (`sum`).

Exemple :

- Entrée dans le Reducer : (`cléA, [1, 1, 1]`)
- Sortie du Reducer : (`cléA, 4`), indiquant que `cléA` est apparu 4 fois.

### Exemple d'Exécution

#### Entrée du Reducer

Après la phase Map et le regroupement (shuffle), le Reducer reçoit :

(cléA, [1, 1, 1])

(cléB, [1, 1])

(cléC, [1, 1, 1, 1])

#### Sortie du Reducer

Après le traitement, le Reducer émet :

(cléA, 3)

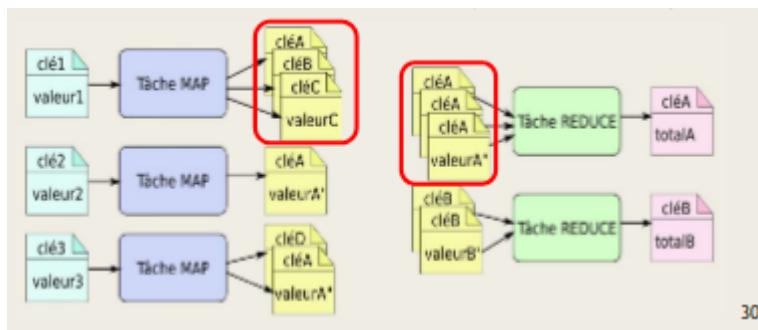
(cléB, 2)

(cléC, 4)

Cela indique que `cléA` apparaît 3 fois, `cléB` 2 fois, et `cléC` 4 fois dans le texte analysé.

### Résumé

- Le Mapper produit des paires (`mot, 1`).
- Hadoop regroupe les valeurs par clé (`mot`).
- Le Reducer additionne toutes les valeurs associées à un mot.
- Le résultat final donne le nombre total d'occurrences de chaque mot.



30

## Explication de l'image

L'image illustre le fonctionnement de **MapReduce** avec l'étape **MAP** et l'étape **REDUCE**.

### 1. Étape MAP (Partie gauche de l'image)

- Chaque ligne du fichier est traitée par une **tâche MAP**.
- La tâche MAP extrait des paires (**clé**, **valeur**).
- Ici, la clé est un mot (ex : **cléA**, **cléB**, etc.) et la valeur est un nombre (ex : **valeur1**, **valeur2**).
- Après traitement, les mots sont regroupés par clé.

### 2. Shuffle & Sort (Partie centrale en rouge)

- Les clés identiques sont regroupées ensemble avant l'étape de réduction.
- Exemple : toutes les occurrences de **cléA** sont rassemblées.

### 3. Étape REDUCE (Partie droite de l'image)

- Une tâche REDUCE reçoit chaque clé et additionne ses valeurs.
- Exemple : **cléA** est associée aux valeurs **valeurA**, **valeurA\***, qui sont combinées pour produire **totalA**.
- Le résultat final est une liste de (**clé**, **total**) indiquant le nombre d'occurrences de chaque clé.

## Lien avec le Code WordCountMapper

- Le **Mapper** du code extrait chaque mot d'une ligne et l'associe à **1** (comme **cléA** -> **1**).

- L'image montre comment ces paires sont triées et regroupées avant d'être additionnées dans la phase **Reduce**.

## Interface Writable

L'interface **Writable** permet la **sérialisation et la désérialisation** des données sous forme d'octets, facilitant leur échange entre machines aux architectures différentes. Elle est essentielle pour le fonctionnement distribué de Hadoop. Des types comme **Text**, **IntWritable** et **LongWritable** en sont des implémentations.

MAGUEMOUN SAMY

# CHAPITRE 3 : LES BASES DE DONNÉES NOSQL

## 1. Modèles de gestion des données Les SGBD Relationnels

**Hiérarchique** : Données organisées en arbre (parent → enfant), structure rigide, peu adaptée aux relations complexes.

**Réseau** : Extension du modèle hiérarchique, avec relations multiples (graphe), plus flexible.

**Relationnel** : Données sous forme de tables, manipulation via SQL, très utilisé mais limité pour les données massives et non structurées.

**Orienté objet** : Données modélisées comme des objets (avec données + méthodes), adapté aux systèmes complexes, peu utilisé en production.

**XML** : Données hiérarchiques avec métadonnées, bon pour l'échange entre systèmes hétérogènes, mais verbeux et lent à traiter.

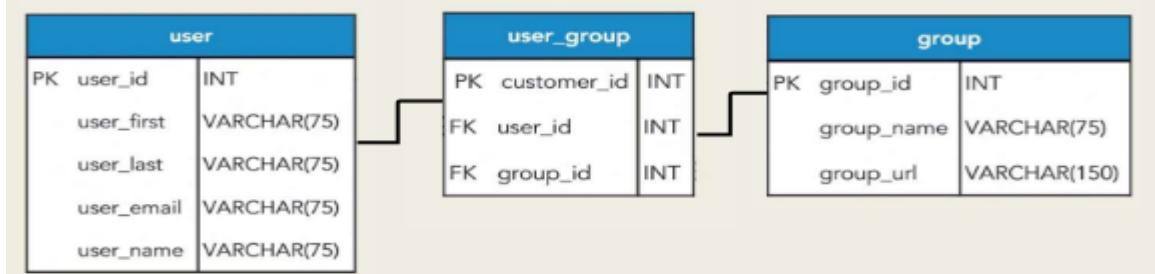
**Graphe** : Données sous forme de noeuds et relations, excellent pour représenter les connexions (réseaux sociaux, moteurs de recommandation).

**Document** : Données stockées dans des documents JSON/BSON, très flexible, scalable, idéal pour les bases NoSQL comme MongoDB.

### 📌 Définition du modèle relationnel :

Le **modèle relationnel** est un modèle de gestion de base de données où les **données sont organisées sous forme de tables** (appelées *relations*), avec des **colonnes** représentant les attributs et des **lignes** représentant les enregistrements. Les relations entre les données sont établies via des **clés primaires (PK)** et **clés étrangères (FK)**. Ce modèle est la base des systèmes de gestion de bases de données relationnelles (SGBDR) comme MySQL, PostgreSQL ou Oracle.

#### Exemple : Le modèle relationnel





## Explication de l'image : Exemple de modèle relationnel

L'image montre un **exemple classique d'un schéma relationnel** avec **trois tables** : `user`, `group` et `user_group`.

### 1. Table `user`

Contient les informations des utilisateurs :

- `user_id` (clé primaire)
- `user_first, user_last, user_email, user_name` (attributs VARCHAR)

### 2. Table `group`

Contient les informations des groupes :

- `group_id` (clé primaire)
- `group_name` et `group_url`

### 3. Table `user_group`

C'est une **table d'association (relation N-N)** entre `user` et `group` :

- `customer_id` (clé primaire)
- `user_id` (clé étrangère vers `user.user_id`)
- `group_id` (clé étrangère vers `group.group_id`)



### Résumé :

Ce schéma permet de **lier plusieurs utilisateurs à plusieurs groupes**, grâce à une table d'association. C'est un **cas typique de relation "plusieurs à plusieurs"** dans un modèle relationnel.

## 2. Les SGBD Relationnels



### Transactions ACID (dans les Systèmes de Gestion de Bases de Données Relationnels)

Les transactions doivent respecter les **propriétés ACID** pour garantir la fiabilité des opérations sur la base de données :

### 1. Atomicité :

Une transaction est **indivisible**. Elle s'exécute en totalité ou pas du tout.

➤ *Pas de demi-mise à jour.*

### 2. Cohérence :

La base passe d'un **état cohérent à un autre** après la transaction.

➤ *Les règles d'intégrité sont toujours respectées.*

### 3. Isolation :

Les transactions sont **indépendantes** les unes des autres.

➤ *Une transaction en cours n'impacte pas les autres.*

### 4. Durabilité :

Une fois validée (commit), une transaction est **permanente**.

➤ *Les données restent intactes même en cas de panne.*

## 📌 Caractéristiques des SGBD-R :

### ✓ Jointures puissantes :

Permettent de créer des requêtes complexes en liant plusieurs tables.

### ✓ Intégrité référentielle :

Assure que les relations entre les entités (tables) sont valides et cohérentes.

### ⚠ Limites des SGBD-R (en contexte distribué) :

#### ✗ Coût élevé des mécanismes (jointures, intégrité) sur des systèmes distribués.

#### ✗ Placement difficile des données :

Les données liées doivent rester sur le **même nœud**, ce qui complique la distribution.

#### ✗ Difficulté à respecter ACID :

Plus les données sont distribuées, plus il est compliqué de garantir les propriétés **ACID**.

## 3. Pourquoi opter pour NoSQL ? Les Bases de données NoSQL

- **Nouveaux besoins** : Les modèles traditionnels ne suffisent plus pour gérer la diversité et la volumétrie actuelle des données.
- **Big Data & Web à grande échelle** : NoSQL est conçu pour supporter d'énormes volumes de données et une montée en charge horizontale (web-scale).
- **Agilité & élasticité** : Adapté aux environnements changeants, NoSQL offre plus de flexibilité pour faire évoluer les modèles de données.
- **Open Source** : La majorité des bases NoSQL sont libres, favorisant l'innovation, la personnalisation et la réduction des coûts.

## 4. Les Bases de données NoSQL

### Caractéristiques des bases NoSQL

- **Modèle non relationnel** : Utilisent une structure de données différente des bases relationnelles (clé-valeur, document, graphe, etc.).
- **Complément aux SGBD-R** : Ne visent pas à les remplacer, mais offrent des solutions mieux adaptées à certains besoins (big data, web...).
- **Performance élevée** : Idéales pour les applications web gérant de grandes quantités de données et nécessitant une forte réactivité.
- **Moins de contraintes ACID** : Sacrifient partiellement les garanties ACID pour offrir **plus de scalabilité et d'évolutivité horizontale**.
- **Schéma flexible ou inexistant** : Les données peuvent être enregistrées sans structure fixe, permettant plus d'agilité.
- **Données distribuées** : Reposent sur un **partitionnement horizontal** sur plusieurs nœuds, souvent avec des algorithmes comme **MapReduce**.

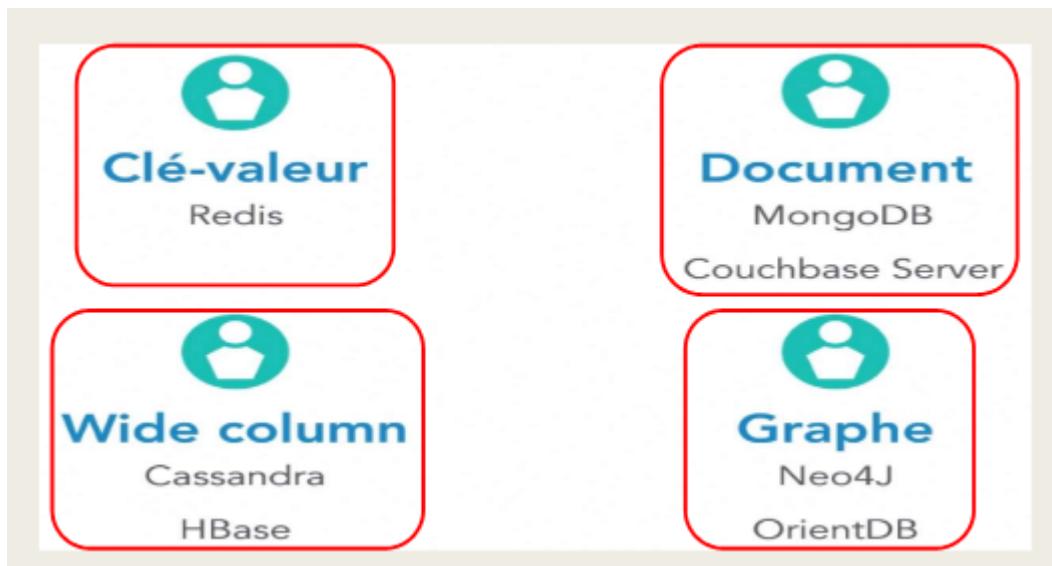
## 5. Propriétés des systèmes distribués: CAP

### ▲ CAP : Consistency, Availability, Partition tolerance

-  **Consistency (Cohérence)** :  
Toutes les copies des données dans le système sont mises à jour simultanément après une modification.  
➤ *Tous les nœuds voient les mêmes données au même moment.*
-  **Availability (Disponibilité)** :  
Chaque requête adressée à un nœud du système reçoit une réponse, même si certaines parties du système sont défaillantes.  
➤ *Le système reste réactif.*
-  **Partition tolerance (Tolérance au partitionnement)** :  
Le système continue de fonctionner même si des communications entre nœuds échouent (coupures, ajouts, suppressions).  
➤ *Pas d'interruption en cas de partition du réseau.*

 **Remarque** : Dans un système distribué, on ne peut garantir **que deux propriétés sur trois** en même temps. C'est le principe du **théorème CAP**

## 6. Modèles de gestion des données NoSQL 1) Modèle de BD en Clé-Valeur



Cette image illustre **les quatre grandes familles de bases de données NoSQL**, chacune avec des exemples populaires. Voici une explication simple pour chaque type :

◆ **Clé-Valeur (Key-Value)**

- **Exemple** : Redis
- **Principe** : Stocke les données sous forme de paires *clé* → *valeur*.
- **Utilisation** : Très rapide pour récupérer une valeur à partir d'une clé.
- **Cas typique** : Caches, sessions utilisateur.

◆ **Document**

- **Exemples** : MongoDB, Couchbase Server
- **Principe** : Stocke des documents (généralement JSON ou BSON). Chaque document peut avoir une structure différente.
- **Utilisation** : Parfait pour les données semi-structurées, applications web.
- **Avantage** : Flexibilité du schéma.

◆ **Wide Column (Colonnes larges)**

- **Exemples** : Cassandra, HBase
- **Principe** : Stocke les données par colonnes au lieu de lignes, dans des structures appelées *families*.

- **Utilisation** : Idéal pour les gros volumes de données et les requêtes analytiques.
- **Cas typique** : Big Data, séries temporelles.

#### ◆ **Graphe**

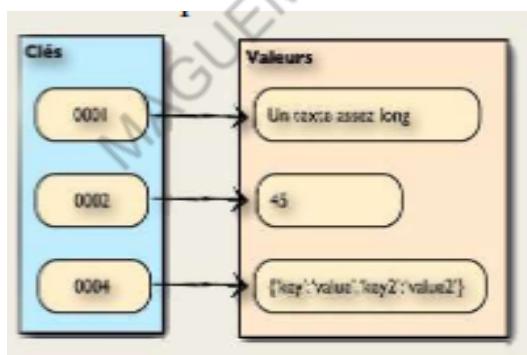
- **Exemples** : Neo4j, OrientDB
- **Principe** : Représente les données sous forme de nœuds (entités) et d'arêtes (relations).
- **Utilisation** : Pour des données très liées (réseaux sociaux, recommandations).
- **Avantage** : Excellente performance pour les requêtes relationnelles complexes.

#### 📌 Résumé :

Cette image classe les bases NoSQL selon leur **modèle de données** et montre les **SGBD les plus connus** pour chaque catégorie. Chaque type répond à **des besoins spécifiques** selon la nature des données et l'usage.

### 1) Modèle de BD en Clé-Valeur

Le modèle **clé-valeur** ressemble à une table de hachage, où chaque valeur est récupérée via une clé, sans structure fixe. La valeur peut être de tout type (chaîne, objet, etc.), ce qui rend les requêtes plus limitées et moins précises.



Cette image illustre le **modèle de base de données clé-valeur**, un type de base NoSQL.

#### **Explication :**

- **Colonne de gauche ("Clés")** : contient des identifiants uniques (ex : 0001, 0002, 0004) qui servent à accéder aux données.
- **Colonne de droite ("Valeurs")** : chaque clé pointe vers une valeur, qui peut être :
  - **Un texte** (ex : Un texte assez long)

- Un **nombre** (ex : 45)
- Une **structure complexe** comme un objet ou une collection sérialisée (ex : `{"key1": "value1", "key2": "value2"}`)

### À retenir :

Ce modèle est **simple, rapide et efficace** pour les opérations de lecture/écriture, mais ne permet pas de requêtes complexes. C'est idéal pour des cas où on connaît la clé à interroger, comme les sessions utilisateurs, les caches ou les profils simples.

### Avantages :

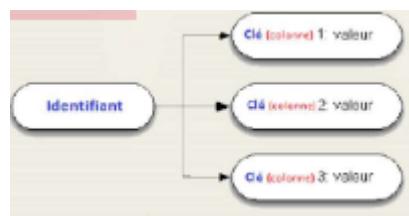
- Lecture/écriture très rapide
- Modèle simple à utiliser
- Facilement scalable (passage à l'échelle)

### Inconvénients :

- Requêtes limitées, il faut connaître la clé
- Modèle trop basique pour les données complexes
- L'application doit gérer elle-même une partie de la logique normalement traitée par SQL

## 2) Bases de données orientées colonnes

Les données sont stockées par colonnes (et non par lignes), ce qui permet une meilleure compression et performance pour certaines requêtes. Chaque colonne est un couple clé/valeur, et peut contenir d'autres colonnes (super colonnes). Le modèle est flexible (colonnes dynamiques, pas de NULL stocké) mais moins optimisé pour les insertions ligne par ligne.



Cette image illustre le **modèle de base de données orienté colonne**, typique des bases NoSQL comme **Cassandra** ou **HBase**.

## Explication de l'image :

- L'identifiant (à gauche) représente une **clé de ligne** unique (comme une clé primaire).
- À droite, cette clé est associée à **plusieurs paires clé-valeur**, appelées **colonnes**.
  - Par exemple :
    - Clé (colonne) 1 : valeur
    - Clé (colonne) 2 : valeur
    - Clé (colonne) 3 : valeur

## À retenir :

- Contrairement aux bases relationnelles qui stockent ligne par ligne, ici **chaque ligne peut avoir un nombre et des types de colonnes différents**.
- Ce modèle est :
  - **Flexible** : colonnes dynamiques.
  - **Optimisé pour les lectures/écritures en masse par colonnes**.
  - Idéal pour les systèmes de **gestion de logs, analyses temps réel, ou big data**.

C'est une bonne solution quand les données sont volumineuses, structurées en colonnes, et quand les requêtes ne concernent qu'un sous-ensemble de colonnes.

## Avantages :

- **Bon passage à l'échelle** : Le système peut facilement gérer un volume croissant de données.
- **Indexation naturelle (colonnes)** : Les données sont indexées de manière optimisée pour les colonnes, facilitant les recherches.
- **Ajout facile de nouvelles colonnes** : Il n'est pas nécessaire de redimensionner les lignes lorsque de nouvelles colonnes sont ajoutées.

- **Rapidité d'accès aux données** : Il est plus rapide d'accéder aux informations d'une colonne entière plutôt que de parcourir toutes les lignes pour extraire une seule donnée.

## Inconvénients :

**Efficace uniquement dans un contexte «Big Data»** : Ce modèle est surtout adapté aux grandes quantités de données et peut être moins performant à petite échelle.

**Lecture difficile pour les données complexes** : Les données plus complexes peuvent être difficiles à manipuler et à lire efficacement.

**Maintenance complexe** : Ajouter, supprimer ou regrouper des colonnes peut être un processus compliqué à gérer.

**Système de requêtes minimalistes** : Les requêtes sont souvent limitées à des opérations simples comme "select", "update" ou "delete", ce qui restreint la flexibilité.

## 3) Bases de données orientées documents

Les bases de données orientées documents, un type de base NoSQL, stockent des informations sous forme de documents semi-structurés. Elles offrent une grande flexibilité dans la gestion de données variées et complexes, sans nécessiter un modèle de données prédéfini, contrairement aux bases relationnelles traditionnelles. stockage d'une collection de documents :

- **Basé sur des paires clés-valeurs** : Chaque document est identifié par une clé unique, et sa valeur est un document semi-structuré (en XML, JSON, etc.).
- **Structure des documents** : Un document est constitué de champs avec des valeurs associées.
  - Les valeurs peuvent être de types simples (entiers, chaînes de caractères, dates, etc.).
  - Les valeurs peuvent aussi être des sous-documents, eux-mêmes constitués de paires clé-valeur.
- **BD "schemaless"** : Il n'est pas nécessaire de définir un modèle de données avant d'enregistrer les documents.
- **Hétérogénéité** : Les documents dans la base de données peuvent être très variés en termes de structure et de contenu.

### **Avantages :**

- Modèle de données simple mais puissant, basé sur des documents hiérarchiques.
- Aucune maintenance nécessaire pour l'ajout ou la suppression des documents.
- Possibilité de réaliser des requêtes assez complexes, simplifiant le développement de l'application.

### **Inconvénients :**

- Lenteur pour les requêtes volumineuses.
- Moins adapté pour gérer des données interconnectées.
- Le modèle de requête est limité aux clés, restreignant la flexibilité.

## **4) Bases de données orientées Graphes**

Les bases de données orientées graphes sont conçues pour stocker des données complexes avec des relations. Elles reposent sur la **théorie des graphes**, où les **nœuds** représentent les éléments de stockage (similaires aux bases documentaires) et les **relations** décrivent les arcs entre ces nœuds. Ces bases utilisent des **nœuds**, des **relations** et des **propriétés** associées pour organiser les données. Elles sont particulièrement adaptées pour la manipulation d'objets complexes organisés en réseaux, comme dans la **cartographie** ou les **réseaux sociaux**.

### **Cas pratique des bases de données orientées**

#### **Colonnes : Présentation de HBase**

**HBase est une base de données de l'environnement Hadoop.**

**Installation sur un cluster Hadoop** : HBase fonctionne sur un cluster Hadoop.

**Utilisation de HDFS** : HBase utilise le système de fichiers distribué Hadoop (HDFS) pour distribuer et répliquer les données sur le cluster.

**Performances exceptionnelles** : HBase offre des performances élevées pour les lectures et écritures massives de données.

**Conçu pour de très grandes tables** : HBase est optimisée pour stocker de grandes tables (des milliards d'enregistrements avec des millions de colonnes).

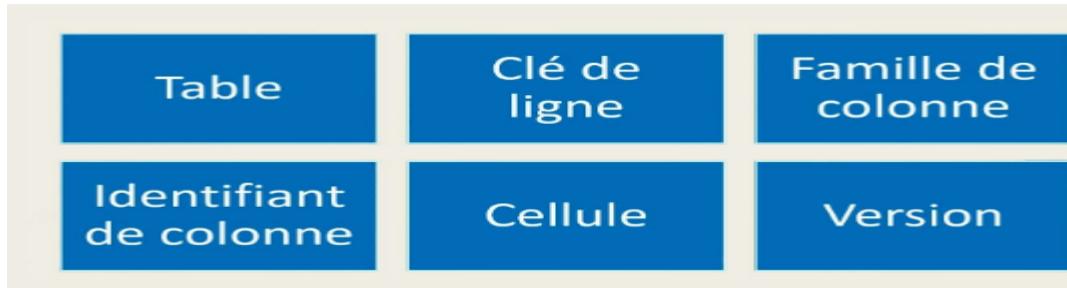
**Adaptée pour de grandes collections de documents** : Idéale pour le stockage de grandes quantités de données ou pour du profiling.

## HBase versus SGBD-Relationnel

**SGBDR** : Fortement structuré, avec un modèle de données statique.

**HBase** : Pauvrement structuré, avec un modèle de données souple et scalable.

### Modèle de données HBase



**Table** : Les données sont organisées en tables, et les noms des tables sont des chaînes de caractères.

**Clé de ligne** : Chaque table est composée de lignes, identifiées par une **clé unique** (Row Key). Cette clé n'a pas de type spécifique et est traitée comme un tableau d'octets.

**Colonne** :

- Chaque Row Key peut contenir un nombre illimité d'attributs dans les colonnes.
- Aucune structure de schéma strict n'est imposée pour les colonnes, et de nouvelles colonnes peuvent être ajoutées instantanément.
- Le **column qualifier** permet d'accéder aux données et est spécifié lors de l'insertion des données, pas lors de la création de la table.
- Le **column qualifier**, comme les Row Keys, est traité comme un tableau d'octets et n'a pas de type spécifique.

**Famille de colonnes (Column Family)** :

- Les colonnes sont regroupées en **Column Families**. Toutes les lignes d'une table partagent les mêmes Column Families, qui peuvent être vides.
- Les Column Families doivent être définies lors de la création de la table et ne peuvent pas être modifiées après.
- Les noms des **Column Families** sont des chaînes de caractères.

### Cellule :

- Une cellule est identifiée de manière unique par la combinaison de la **RowKey**, **Column Family**, et **Column Qualifier**.
- Les données stockées dans une cellule sont appelées les **valeurs** de cette cellule.
- Les valeurs n'ont pas de type spécifique, elles sont toujours traitées comme un tableau d'octets.

### Version :

- Les valeurs dans une cellule sont versionnées, chaque version étant identifiée par un **Timestamp**.
- Une fois qu'une valeur est écrite dans HBase, elle ne peut pas être modifiée. Une nouvelle version peut être ajoutée avec un **Timestamp** plus récent.
- Le **Timestamp** est de type **long**.

Région	Clé de ligne	Identité		Adresse		
		Prénom	Nom	Rue	Code Postal	Ville
Région 1	Clé 1	Prénom 1	Nom 1		15000	Tizi-Ouzou
	Clé 2	Prénom 2	Nom 2	Rue Didouche	16000	Alger
Région 2	Clé 3	Prénom 3	Nom 3			

Cette image illustre un modèle de base de données orientée colonne, similaire à celles utilisées dans les systèmes NoSQL comme Cassandra ou HBase.

### Explication de l'image :

1. **Structure générale :**
  - Le tableau est divisé en quatre grandes colonnes : **Région**, **Clé de ligne**, **Identité**, et **Adresse**.
  - Les colonnes **Identité** et **Adresse** sont elles-mêmes subdivisées en sous-colonnes (Prénom, Nom, Rue, Code Postal, Ville).
2. **Clé de ligne :**

- La colonne **Clé de ligne** (par exemple, "Clé 1", "Clé 2", etc.) agit comme un identifiant unique pour chaque enregistrement, similaire à une clé primaire dans une base de données relationnelle.

### 3. Colonnes dynamiques :

- Chaque ligne peut avoir des valeurs différentes pour les colonnes, y compris des valeurs manquantes (cellules vides). Par exemple :
  - Pour la "Clé 1", la colonne **Rue** est vide, tandis que le **Code Postal** et la **Ville** sont renseignés.
  - Pour la "Clé 3", toutes les sous-colonnes de **Adresse** sont vides.

### 4. Exemple de données :

- **Clé 1 :**
  - Identité : Prénom 1, Nom 1
  - Adresse : Code Postal = 15000, Ville = Tizi-Ouzou
- **Clé 2 :**
  - Identité : Prénom 2, Nom 2
  - Adresse : Rue = Rue Didouche, Code Postal = 16000, Ville = Alger
- **Clé 3 :**
  - Identité : Prénom 3, Nom 3
  - Adresse : Toutes les sous-colonnes sont vides.

### À retenir :

- **Flexibilité :**  
Contrairement aux bases de données relationnelles, où toutes les lignes doivent avoir les mêmes colonnes, ce modèle permet des colonnes dynamiques et des valeurs manquantes.
- **Optimisation :**  
Ce modèle est optimisé pour les opérations massives sur des colonnes spécifiques, ce qui le rend idéal pour les analyses de données volumineuses ou les requêtes ciblées.
- **Cas d'usage :**  
Il est particulièrement adapté pour :
  - Les systèmes de gestion de logs.
  - Les analyses en temps réel.
  - Les applications big data où seules certaines colonnes sont fréquemment interrogées.

**En résumé**, ce modèle est une solution puissante pour les données hétérogènes ou évolutives, offrant à la fois performance et adaptabilité.

Clé	Colonne	Version	Valeur
Clé 1	Identité : Prénom	1	Prénom 1
Clé 1	Identité : Nom	1	Nom 1
Clé 1	Adresse : Rue	1	Rue Didouche
Clé 1	Adresse : Code Postal	1	16000
Clé 1	Adresse : Ville	1	Alger
Clé 2	Identité : Prénom	1	Prénom 2

Cette image illustre un modèle de stockage clé-valeur avec versionnement, typique des bases de données orientées colonnes ou des systèmes comme Apache Cassandra ou Google Bigtable.

### Explication de l'image :

#### 1. Structure générale :

- Le tableau est organisé en 4 colonnes : **Clé**, **Colonne**, **Version**, et **Valeur**.
- Chaque ligne représente une **cellule** individuelle dans la base de données, associée à une clé, une colonne, une version et sa valeur correspondante.

#### 2. Clé de ligne :

- La colonne **Clé** (ex. "Clé 1", "Clé 2") identifie de manière unique un enregistrement ou une entité.
- Une même clé peut avoir plusieurs lignes, chacune correspondant à une colonne différente (modèle "wide column").

#### 3. Colonnes et sous-colonnes :

- La colonne **Colonne** utilise une hiérarchie (ex. **Identité:Prénom**, **Adresse:Ville**) pour organiser les données en familles de colonnes.
- Cela permet une structuration flexible, similaire aux modèles NoSQL comme HBase.

#### 4. Versionnement :

- La colonne **Version** (toujours "1" dans cet exemple) indique la version de la donnée.
- Certains systèmes utilisent des timestamps ou des numéros de version pour gérer l'historique des modifications.

#### 5. Exemple de données :

##### ○ Clé 1 :

- **Identité:Prénom** → "Prénom 1"
- **Identité:Nom** → "Nom 1"
- **Adresse:Rue** → "Rue Didouche"
- **Adresse:Code Postal** → "16000"
- **Adresse:Ville** → "Alger"

##### ○ Clé 2 :

- Seul **Identité:Prénom** est renseigné ("Prénom 2").

## À retenir :

- **Modèle scalable :**  
Idéal pour les données distribuées, où chaque ligne peut avoir des colonnes différentes sans schéma fixe.
- **Flexibilité :**  
Permet d'ajouter dynamiquement des colonnes (ex. ajouter `Adresse : Pays` sans modifier les autres entrées).
- **Optimisation :**  
Les requêtes sur des colonnes spécifiques (ex. tous les `Code Postal`) sont très efficaces.
- **Cas d'usage :**
  - Stockage de profils utilisateurs avec des attributs variables.
  - Données temporelles ou versionnées (ex. suivis de modifications).
  - Systèmes où les lectures sont plus fréquentes que les écritures.

### Différence avec la capture précédente :

Ici, chaque attribut est stocké comme une ligne distincte (format "dénormalisé"), tandis que la première image montrait une vue tabulaire classique. Ce modèle est plus adapté aux bases distribuées.

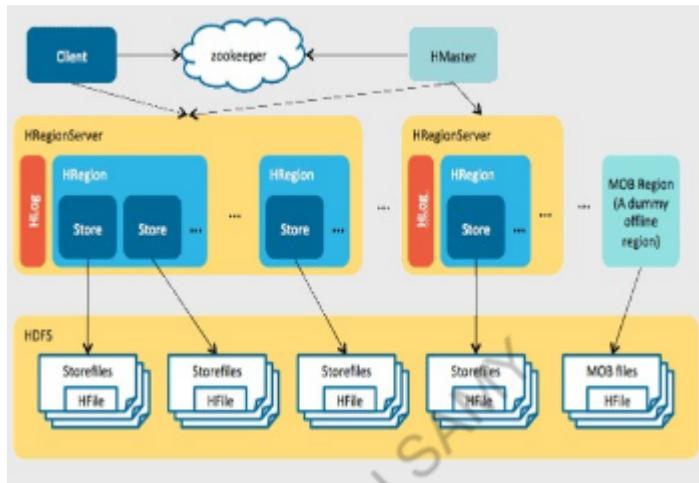
## Architecture de HBase

### Services :

- **HBase Master** : Coordonne l'ensemble du cluster HBase, gère les régions et les serveurs.
- **HBase RegionServer** : Gère les régions de données, où les données sont effectivement stockées et lues/écrites.
- **HRegion** : Unité de stockage de données dans HBase, chaque région contient une portion des données.
- **Zookeeper** : Assure la coordination et la gestion des configurations distribuées entre les différents serveurs HBase.
- **HBase REST** : Interface permettant d'accéder aux données HBase via HTTP.
- **HBase Thrift** : Interface permettant l'accès aux données HBase via le protocole Thrift.

## Opérations d'administration :

- **Equilibrage** : Processus d'équilibrage des données et des régions entre les serveurs pour garantir une répartition uniforme.
- **Haute disponibilité** : Mécanismes garantissant que le système continue de fonctionner même en cas de défaillance de certains serveurs, assurant ainsi une disponibilité constante.



## Opérations de HBase

### DDL (Data Definition Language) :

- **Create** : Créer une table ou une famille de colonnes.
- **Describe** : Afficher la structure d'une table.
- **Alter** : Modifier la structure d'une table existante.
- **Drop** : Supprimer une table.
- **Enable** : Activer une table désactivée.
- **Disable** : Désactiver une table pour effectuer des opérations administratives.

### DML (Data Manipulation Language) :

- **Put** : Insérer ou mettre à jour des données dans une table.

- **Get** : Récupérer des données d'une table.
- **Delete** : Supprimer des données d'une table.
- **Scan** : Effectuer une lecture complète des données dans une table, souvent utilisée pour récupérer plusieurs enregistrements.

MAGUEMOUN SAMY

# CHAPITRE 4 : Introduction au cloud computing

## Partie 1 : Introduction au Cloud Computing

### 1. Définition du Cloud Computing :

Le **Cloud Computing** est une méthode de gestion des ressources informatiques (comme les serveurs) qui permet de **s'adapter rapidement aux variations de charge** sans intervention directe de l'administrateur ni perturbation pour l'utilisateur.

Au lieu d'être hébergées localement, les **applications cloud** s'exécutent sur un ensemble de **serveurs interconnectés** situés dans de grands **datacenters**.

Cette flexibilité est rendue possible grâce à la **virtualisation**, une technologie qui permet de faire fonctionner **plusieurs systèmes virtuels** sur un **même serveur physique**.

---

### 2. Quels intérêts de faire du cloud ? :

**Rentabilité** : Réduction des coûts d'infrastructure et paiement à l'usage.

**Agilité** : Mise en place rapide de services et adaptation facile aux besoins.

**Haute disponibilité** : Accès continu aux services, même en cas de panne.

**Sécurité** : Protection des données assurée par les fournisseurs cloud.

**Égalité d'accès** : Tous les utilisateurs peuvent bénéficier des mêmes ressources, peu importe leur taille ou localisation.

---

### 3. Caractéristiques du Cloud Computing :

#### 1. Libre-service à la demande

- L'utilisateur peut activer des ressources (ex. stockage, serveur) automatiquement, sans intervention humaine.

#### 2. Accès réseau étendu

- Les services sont accessibles partout via Internet ou un réseau local, à l'aide de protocoles standards.

#### 3. Mutualisation des ressources

- Les ressources sont partagées entre plusieurs utilisateurs et attribuées dynamiquement selon les besoins.

#### 4. Élasticité et scalabilité

- Les ressources peuvent être rapidement augmentées ou réduites selon la

demande, donnant l'impression de ressources illimitées.

## 5. Facturation à l'usage

- L'utilisation des services est surveillée et mesurée, permettant une transparence totale dans la consommation et la facturation.
- 

## 4. Avantages & considérations liés à l'utilisation du Cloud :

### 1. Haute disponibilité

- Les services cloud assurent un fonctionnement continu, même en cas de panne.
- Cela repose sur :
  - Une **conception d'application** adaptée
  - Une **infrastructure technique** capable de soutenir cette disponibilité

### 2. Scalabilité (Évolutivité)

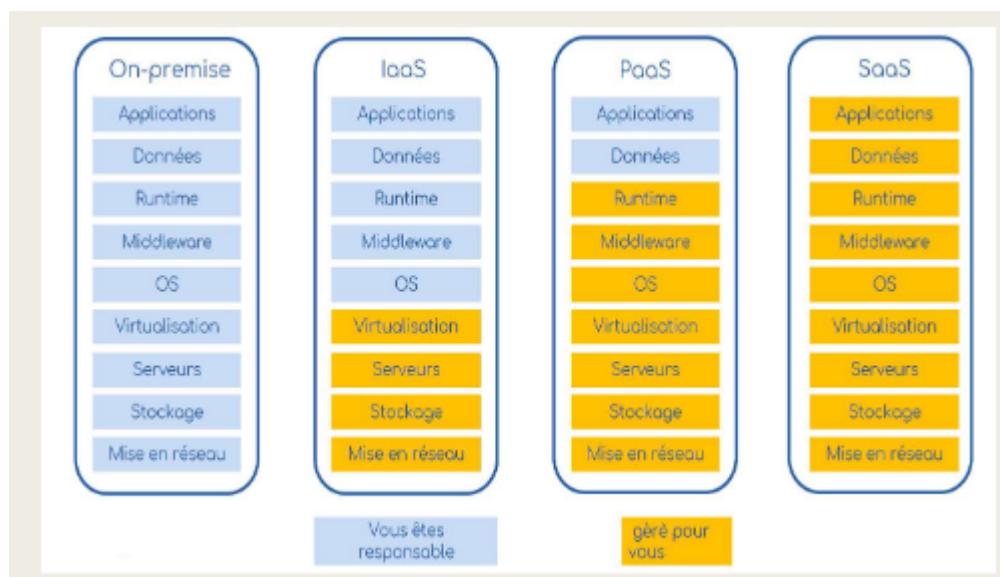
- Capacité à **adapter les ressources** en fonction des besoins :

- **Scale-up** : augmenter/diminuer la puissance d'un serveur (vertical)
- **Scale-out** : ajouter/retirer des serveurs (horizontal)

### 3. Élasticité (Agilité)

- Le système **s'ajuste automatiquement** à la charge de travail, assurant performance et optimisation des ressources en temps réel.
- 

## 5. Les modèles de cloud :



L'image illustre la **structure technique des solutions on-premise** (sur site) et met en contraste les **responsabilités** entre l'entreprise (vous) et le fournisseur ("gère pour vous").

## 1. Partie supérieure : Stack technique On-premise

La liste répétitive montre les couches d'une infrastructure locale typique :

- **Applications** : Logiciels métiers (ex : ERP, bases de données).
- **Données** (*Domées* = probablement une coquille pour *Données*).
- **Runtime** : Environnements d'exécution (ex : JVM, .NET).
- **Middleware** : Logiciels intermédiaires (ex : API, messagerie).
- **OS** : Système d'exploitation (ex : Windows Server, Linux).
- **Virtualisation** : Hyperviseurs (ex : VMware, Hyper-V).
- **Hardware** : Serveurs, stockage (SAN/NAS), mise en réseau (routeurs, switchs).

*Note* : La répétition des sections "Tools" est probablement une erreur de conception, mais elle souligne l'imbrication de ces couches.

## 2. Partie inférieure : Répartition des responsabilités

- "**Vous êtes responsable**" : En on-premise, l'entreprise gère **toutes les couches** (de la virtualisation au hardware, en passant par les correctifs logiciels).
- "**gère pour vous**" : Contraste implicite avec le cloud, où le fournisseur prend en charge certaines couches (ex : dans le IaaS, l'OS et le hardware sont gérés par AWS/Azure).

### Message clé

L'image met en avant :

- **La complexité** de l'infrastructure on-premise (tout est sous votre contrôle, mais aussi sous votre responsabilité).
- **La différence avec le cloud**, où une partie de la stack est externalisée.

*Exemple* :

- **On-premise** : Vous gérez les serveurs, les sauvegardes, les mises à jour de l'OS.
- **Cloud IaaS** : Le fournisseur gère le hardware, vous gérez l'OS et les applications

## 1. IaaS (Infrastructure as a Service)

### ♦ Définition :

L'utilisateur loue une **infrastructure informatique** (serveurs, stockage, réseau) virtualisée sur Internet.

Il installe et gère lui-même ses systèmes d'exploitation, applications, et données.

### Avantages :

- Contrôle total sur l'environnement logiciel.
- Ressources flexibles (scalabilité simple).
- Paiement à l'usage (réduction des coûts matériels initiaux).
- Idéal pour les développeurs ou les entreprises ayant des besoins spécifiques.

#### Inconvénients :

- Nécessite des compétences techniques (administration système, sécurité...).
- Maintenance et mises à jour logicielles à la charge de l'utilisateur.

#### Exemples :

- Amazon EC2 (machines virtuelles)
- Amazon S3 (stockage en ligne)

## 2. PaaS (Platform as a Service)

#### ♦ Définition :

Fournit un environnement de développement complet avec les **outils, bibliothèques, et plateformes nécessaires**.

L'utilisateur code et déploie des applications sans gérer l'infrastructure sous-jacente.

#### Avantages :

- Gain de temps : l'environnement est préconfiguré.
- Facturation à l'usage.
- Accès facile via un navigateur web.
- Possibilité d'avoir plusieurs environnements de dev/test/production.

#### Inconvénients :

- Tous les langages ne sont pas toujours pris en charge.
- Risque de dépendance au fournisseur (vendor lock-in).
- Les mises à jour imposées peuvent affecter la compatibilité des applications.

### Exemples :

- Google App Engine (Java, Python)
- Microsoft Azure (.NET, PHP, Ruby...)

## 3. SaaS (Software as a Service)

### ◆ Définition :

Le fournisseur propose des **applications hébergées** accessibles à distance via un navigateur.

L'utilisateur utilise simplement le service, sans gérer quoi que ce soit techniquement.

### Avantages :

- Aucun besoin d'installation ou de maintenance.
- Mises à jour automatiques.
- Accessible partout, depuis n'importe quel appareil.
- Adapté aux utilisateurs non techniques.

### Inconvénients :

- Peu ou pas de personnalisation possible.
- Dépendance forte à la connectivité Internet.
- Sécurité et confidentialité des données confiées au fournisseur.

### Exemples :

- Google Docs, Gmail, Google Calendar
- Office 365
- Facebook, LinkedIn

### Résumé final à retenir :

Modèle	Vous gérez quoi ?	Fournisseur gère quoi ?	Usage typique
IaaS	OS, applications, données	Matériel, virtualisation	Hébergement personnalisé, développement libre
PaaS	Applications, données	OS, runtime, serveurs, stockage	Développement rapide d'applications
SaaS	Rien (juste utilisation)	Tout	Utilisation d'outils prêts à l'emploi (bureautique, messagerie...)

## 6. Les architectures de cloud :

Les types d'architectures dépendent de plusieurs critères :

→ Propriétaire, taille de l'organisation, type d'accès, exigences réglementaires et de sécurité.

Il existe trois grandes architectures :

- ◆ Cloud Public
- ◆ Cloud Privé
- ◆ Cloud Hybride

### 1. ☁ Cloud Public

- ◆ Définition :

Infrastructure mise à disposition par un fournisseur (ex : AWS, Azure, Google Cloud) pour plusieurs clients via Internet.

Les ressources sont partagées entre différents utilisateurs (multi-tenant).

#### ✓ Avantages :

- Accès facile et rapide via Internet.
- Large catalogue de services (stockage, calcul, IA, etc.).
- Coût maîtrisé grâce au modèle "pay-as-you-go" (paiement à l'usage).

- Évolutivité quasi illimitée.

 **Inconvénients :**

- **Moins de contrôle** sur la sécurité et la localisation des données.
- **Partage d'infrastructure** avec d'autres organisations (risque potentiel).
- Risque de **dépendance** au fournisseur (vendor lock-in).
- Complexité à gérer lorsqu'on conserve une partie de l'infrastructure en interne (on-premise).

## 2. Cloud Privé

♦ **Définition :**

Infrastructure cloud **dédiée à une seule organisation**, hébergée dans ses propres datacenters ou par un prestataire tiers, mais non partagée avec d'autres entreprises.

 **Avantages :**

- **Sécurité et confidentialité renforcées** : les ressources ne sont pas partagées.
- **Conformité réglementaire** : idéal pour des secteurs sensibles (banque, santé, administration...).
- Personnalisation de l'environnement selon les besoins métier.

 **Inconvénients :**

- **Coûts élevés** (infrastructure, maintenance, personnel qualifié).
- Moins de flexibilité et d'élasticité que le cloud public.
- Demande une gestion et une expertise technique en interne.

 **Utilisation typique :**

- Organisations avec **exigences fortes** en matière de sécurité ou de localisation des données (RGPD, lois nationales...).

## 3. Cloud Hybride

♦ **Définition :**

Combinaison de **cloud public, cloud privé et infrastructures on-premise**.

Permet d'utiliser plusieurs environnements selon les besoins (sécurité, performance, coût).

### Avantages :

- **Flexibilité** : déploiement des charges de travail là où c'est le plus adapté.
- Optimisation des **coûts** : services sensibles en privé, services moins critiques en public.
- **Continuité d'activité** : possibilité de basculer d'un environnement à l'autre.
- Favorise une transition progressive vers le cloud.

### Inconvénients :

- **Complexité de gestion** (synchronisation, sécurité, réseau...).
- Risques accrus de **compatibilité** entre les environnements.
- Coût global potentiellement plus élevé à long terme (maintenance multiple).

### Cas d'usage :

- Entreprises qui souhaitent garder certaines applications critiques sur site tout en profitant de la puissance du cloud public pour d'autres tâches (ex. : traitement de données massives).

## Résumé comparatif

Architecture	Sécurité	Flexibilité	Coût	Gestion	Exemple de cas
Cloud Public	Moyenne	Élevée	Faible à modéré	Simple à externaliser	Startups, applications web
Cloud Privé	Élevée	Moyenne	Élevé	Complexe, interne	Banque, santé, secteur public
Cloud Hybride	Élevée	Très élevée	Variable	Complexe	Grandes entreprises, migration progressive

## Partie 2 :Technologie du Cloud Computing

### I/ La virtualisation :

#### 1. Définition :

La virtualisation est une **technologie logicielle** qui permet de créer des **versions virtuelles** de ressources physiques telles que :un **système d'exploitation**,un **serveur**,un **stockage**,ou un **réseau**.

Elle permet d'exécuter plusieurs **machines virtuelles (VMs)** sur une **même machine physique**, tout en les **isolant** les unes des autres.

Cela facilite :les tests,la sécurité,la gestion des ressources,et la flexibilité informatique.

#### Système Hôte :

C'est la **machine physique** qui héberge les machines virtuelles.

#### Système Invité :

C'est le **système d'exploitation exécuté dans une machine virtuelle (VM)**, tournant au sein du système hôte.



#### 2. Avantages de la Virtualisation :

##### 1. Optimisation des ressources

→ Meilleure utilisation du **processeur** et du **stockage**, souvent sous-exploités sur les serveurs physiques traditionnels.

##### 2. Réduction de la charge d'administration

→ Simplifie la gestion en **regroupant plusieurs machines virtuelles** sur un **même serveur physique**, ce qui facilite les tâches d'administration.

### 3. Domaines d'application de la Virtualisation :

#### 1. Virtualisation d'application

→ Exécute une application de manière indépendante du système d'exploitation (ex : lancer une appli Windows sur Linux).

✓ *Portabilité et compatibilité accrues.*

#### 2. Virtualisation de serveur

→ Plusieurs serveurs virtuels sur une seule machine physique grâce à un **hyperviseur**.

✓ *Optimisation des ressources, réduction des coûts, environnements isolés.*

#### 3. Virtualisation de réseau

→ Simulation de composants réseau (routeurs, firewalls, switches...) dans un environnement virtuel.

✓ *Souplesse, indépendance matérielle, tests simplifiés.*

#### 4. Virtualisation du stockage

→ Fusionne plusieurs espaces de stockage en un seul système centralisé.

✓ *Gestion centralisée et utilisation efficace du stockage.*

#### 5. Virtualisation des postes (Postes de travail)

→ Création de machines virtuelles pour les utilisateurs finaux.

✓ *Mobilité, flexibilité, déploiement rapide pour sous-traitants ou télétravailleurs.*

---

### 4. Techniques de virtualisation de serveur :

#### 1/Notion d'Hyperviseur

La virtualisation de serveur (couramment utilisée dans le **Cloud Computing**) repose sur un logiciel clé : l'**hyperviseur**.

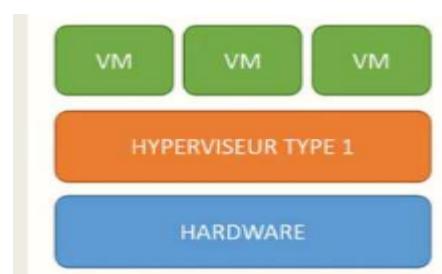
#### 🔧 Qu'est-ce qu'un Hyperviseur ?

C'est un logiciel installé sur un serveur physique, qui permet d'exécuter et de gérer plusieurs **systèmes d'exploitation invités (VM)** sur une même machine.

#### 📌 Deux types d'Hyperviseurs

##### 1. Hyperviseur de type 1 (bare metal)

- Installé **directement sur le matériel (hardware)**, sans OS intermédiaire.
- Plus performant, stable et sécurisé.



- **Utilisation** : Grandes entreprises, datacenters, cloud computing.
- **Exemples** : VMware ESXi, KVM (Linux), Hyper-V (Windows Server).

**Avantages :**

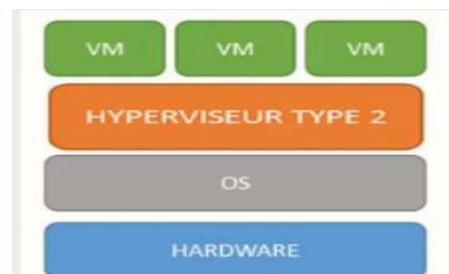
- Meilleures performances
- Plus robuste face aux pannes
- Moins de couches logicielles → moins de vulnérabilités

**Inconvénients :**

- Installation plus complexe
- Nécessite du matériel compatible et souvent dédié

## 2. Hyperviseur de type 2 (host metal)

- S'installe **au-dessus d'un système d'exploitation existant**.
- Moins performant, mais plus simple à utiliser.
- **Utilisation** : Tests, développement, petites structures, usage personnel.
- **Exemples** : VirtualBox, VMware Workstation, Virtual PC.



**Avantages :**

- Facile à installer et configurer
- Idéal pour les tests multiplateformes

**Inconvénients :**

- Performances limitées
- Moins adapté aux environnements de production

## 2/Types de virtualisation de serveur

### 1. Virtualisation complète

- Émule un **matériel complet** pour chaque machine virtuelle (VM).

- **Le système invité n'est pas conscient** d'être virtualisé.
- **Avantages :**
  - Compatible avec de nombreux OS.
  - Aucun besoin de modifier le système invité.
- **Limite : seuls les jeux d'instructions dédiés à la virtualisation** sont utilisables.

## 2. Para-virtualisation

- **Le système invité est conscient** d'être virtualisé.
- Utilise un **noyau hôte léger** + API pour communiquer avec le matériel.
- **Avantages :**
  - Performances améliorées.
  - Optimisation du noyau invité.
- **Inconvénient : nécessite de modifier l'OS invité**, donc moins universel.

## 3. Virtualisation assistée (par le matériel)

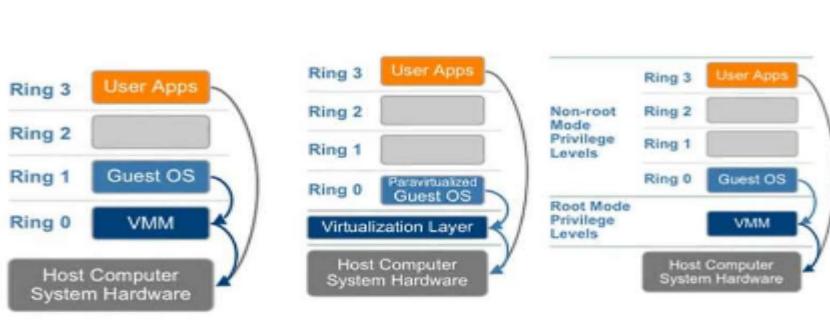
- Utilise les technologies matérielles comme **Intel VT** ou **AMD-V**.
- Permet au processeur d'**exécuter directement certaines instructions** des OS invités.
- **Avantages :**
  - Réduit la charge de l'hyperviseur.
  - Évite de modifier le système invité.
  - Meilleures performances et compatibilité.

## 3/Organisation des privilèges dans les processeurs x86

Les processeurs x86 utilisent **4 niveaux de privilèges** (appelés "rings") pour gérer la sécurité :

- **Niveau 0 :**
  - Accès complet au matériel (instructions privilégiées)
  - Réservé au **noyau du système d'exploitation**.

- **Niveau 1 et 2 :**
  - ⚙️ Intermédiaires, rarement utilisés par les OS modernes.
- **Niveau 3 :**
  - 📦 Niveau **utilisateur** (applications)
  - Accès restreint aux ressources physiques.
  - Pour accéder à des ressources sensibles, il faut passer par le **noyau (niveau 0)**



L'image illustre le modèle de **protection par anneaux (Rings)** utilisé dans les systèmes virtualisés, montrant comment les différents niveaux de priviléges (Ring 0 à Ring 3) s'organisent entre l'hôte physique (*Host Hardware*) et les machines virtuelles (*Guest OS*).

## 1. Principe des Anneaux (Rings)

- **Ring 0** (Noyau) : Niveau le plus privilégié (exécute le noyau OS, hyperviseur).
- **Ring 1** : Souvent utilisé par l'hyperviseur (VMM) pour isoler les machines virtuelles.
- **Ring 2** : Rarement utilisé (parfois pour des drivers ou composants spéciaux).
- **Ring 3** : Niveau utilisateur (applications non privilégiées).

## 2. Ce que montre l'image

- **Partie supérieure :**
  - **Ring 3** : Applications utilisateur (*User Apps*) dans une VM.
  - **Ring 2** : Système d'exploitation invité (*Guest OS*).
  - **Ring 1** : Hyperviseur (*VMM* = Virtual Machine Monitor).
  - **Host Hardware** : La couche physique (CPU, mémoire, etc.).
- **Partie inférieure (répétitive) :**
  - Mise en évidence des **rôles multiples de Ring 0** :
    - Dans une VM, le *Guest OS* s'exécute en **Ring 0 virtuel** (mais est en réalité contrôlé par le VMM en Ring 1 ou Ring -1 selon les architectures).
    - L'hyperviseur (VMM) peut utiliser **Ring 0 physique** pour accéder au hardware.

### 3. Message clé

- **Virtualisation "classique"** (Type 2) : Le *Guest OS* croit être en Ring 0, mais est en réalité virtualisé par le VMM (ex : VirtualBox).
- **Virtualisation matérielle assistée** (Intel VT-x / AMD-V) : Le *Guest OS* utilise un **Ring 0 virtuel**, tandis que le VMM opère en **Ring -1** (mode root, invisible pour les OS invités).

#### Exemple concret

- Sans virtualisation : Un OS natif (ex : Linux) utilise Ring 0 pour le noyau.
- Avec virtualisation :
  - Le VMM (ex : VMware ESXi) prend le contrôle du **vrai Ring 0**.
  - Le *Guest OS* (ex : Windows dans une VM) utilise un **Ring 0 virtuel**, isolé par le VMM.

#### Résumé des techniques de virtualisation système

Technique	OS invité modifié	Détails principaux
Para-virtualisation	✓ Oui	OS modifié pour interagir efficacement avec l'hyperviseur.
Virtualisation complète	✗ Non	Le matériel est émulé, pas besoin de modifier l'OS invité.
Virtualisation assistée	✗ Non	Utilise des extensions matérielles (Intel VT, AMD-V).

---

## 5. Techniques de migration de machines virtuelles :

### ➡ Définition

La migration consiste à déplacer une entité (VM, disque, service, processus...) d'un serveur hôte à un autre.

Elle est essentielle pour :

- ✓ **L'équilibrage de charge**
- ✓ **La tolérance aux pannes**
- ✓ **La réduction de la consommation d'énergie**

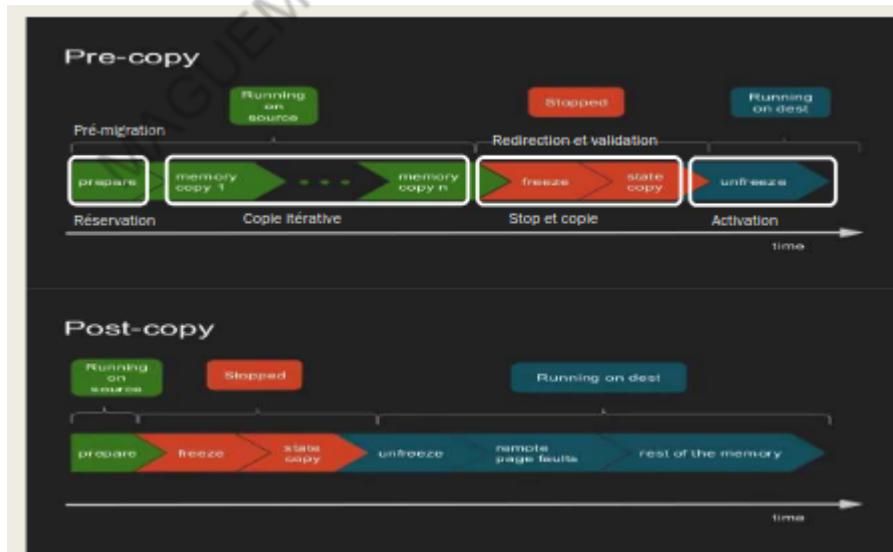
## Types de migration

### 1. Migration à froid :

-  Nécessite l'arrêt de la machine virtuelle.
-  Transfert complet de l'état (disque, mémoire) vers le nouvel hôte.
-  Le temps d'arrêt = temps de migration.
-  Simple mais interrompue.

### 2. Migration à chaud (Live Migration) :

-  Permet le transfert **sans arrêt de la VM**.
-  Utilisée dans des environnements critiques pour assurer la continuité de service.
- Deux approches :
  - **Pré-copie** : copie mémoire avant la pause finale.
  - **Post-copie** : exécution sur le nouveau hôte avant transfert complet.



L'image compare deux méthodes de **migration live** (déplacement de machines virtuelles entre hôtes sans interruption) : **Pre-copy** et **Post-copy**, en soulignant leurs étapes clés et risques potentiels.

### 1. Pre-copy (Copie préalable)

- **Processus :**

- **Copie initiale** : Transfert de l'état complet de la VM (mémoire, CPU) vers le nouvel hôte.
- **Itérations** : Synchronisation des modifications (pages mémoire modifiées pendant la copie).
- **Stop & Activation** : Bascule final vers le nouvel hôte (temps d'arrêt minimal).
- **Avantages** :
  - Perturbation réduite pour l'utilisateur.
  - Adapté aux VMs avec mémoire peu modifiée.
- **Risques** :
  - **"Fatal or true incorrect"** : Si la migration échoue, la VM peut corrompre ses données.

## 2. Post-copy (Copie a posteriori)

- **Processus** :
  - **Activation immédiate** : La VM démarre sur le nouvel hôte avec un état minimal.
  - **Récupération à la demande** : Les pages mémoire manquantes sont transférées depuis l'ancien hôte ("Recruiting an object").
  - **Prédiction** : Optimisation via anticipation des besoins en mémoire ("Predicts Sizes").
- **Avantages** :
  - Temps de migration plus court (pas de copie initiale).
  - Adapté aux VMs avec mémoire très dynamique.
- **Risques** :
  - **"Unintended Intrinsic"** : Latence élevée si accès à des pages non encore migrées.
  - **"Fatal"** : Dépendance critique à la connexion réseau entre hôtes.

## Résumé technique

Méthode	Prérequis	Temps d'arrêt	Risque principal
Pre-copy	Copie totale initiale	Minimal (ms)	Corruption si échec
Post-copy	Transfert à la demande	Variable (dépend du réseau)	Latence élevée

## 6. La virtualisation de stockage :

La virtualisation du stockage consiste à regrouper virtuellement différentes ressources de stockage (disques durs, mémoire flash, lecteurs de bandes) en un **pool unifié appelé Data Store**. Ce concept est crucial notamment dans les environnements **Cloud Computing**.

### 1/Objectifs principaux :

- **Disponibilité**
- **Redondance**

### 2/Technique clé : RAID

Le système **RAID** (Redundant Array of Independent Disks) permet de :

- Répartir les données sur plusieurs disques.
- **Améliorer la performance, la sécurité ou la tolérance aux pannes.**

### 3/Stockage en Bloc vs Stockage en Fichier

#### • **Stockage en Bloc :**

Les données sont **écrites et accédées sous forme de blocs** (volumes de bas niveau). Utilisé dans les SAN.

#### • **Stockage en Fichier :**

Les données sont **organisées via un système de fichiers** (ex. NFS, NTFS) et **accédées par fichiers**. Utilisé dans les NAS.

### 4/RAID – Redundant Array of Independent Disks

RAID regroupe plusieurs disques pour former un **volume unique** visant à améliorer **les performances ou la tolérance aux pannes**.

#### **Types de RAID :**

- **RAID 0 (Striping)**
  - Min. 2 disques
  - Excellente performance (lecture/écriture)
  - **✗ Aucune tolérance aux pannes**

- **RAID 1 (Mirroring)**
  - 2 disques
  - Données dupliquées à l'identique
  - Tolérance aux pannes
- **RAID 5 (Parité répartie)**
  - Min. 3 disques
  - Données + parité réparties
  - Tolérance aux pannes
- **RAID 10 (RAID 1+0)**
  - Min. 4 disques
  - Combinaison de **RAID 0 (striping)** et **RAID 1 (mirroring)**
  - Performance + redondance

## 5/Virtualisation du stockage & Types de serveurs

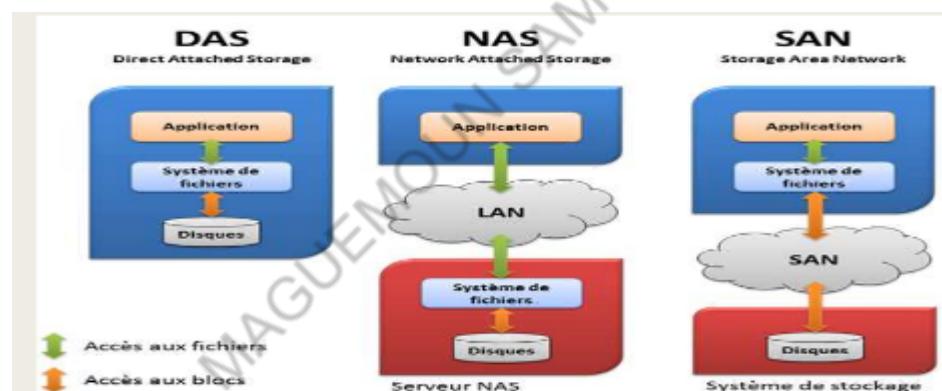
### ◆ **DAS (Direct Attached Storage)**

- Stockage **directement connecté** à un serveur (via SATA, NVMe, USB, etc.).
- **Accès en mode bloc.**
- **Pas de partage** entre plusieurs serveurs.
- **Pas de tolérance aux pannes.**
- **Espace non dynamique** (pas de redéploiement flexible).

### ◆ **NAS (Network Attached Storage)**

- **Serveur de stockage réseau**, accessible par plusieurs machines.
- Fonctionne au **niveau fichier** (protocoles : NFS, CIFS, FTP).
- Possède un **mini OS** et un **système de fichiers intégré**.

- **Administration facile** via une interface web.
  - **Tolérance aux pannes** via RAID.
  - Moins adapté aux **gros volumes de données** ou aux applications critiques.
- ◆ **SAN (Storage Area Network)**
- Réseau dédié au stockage, accès en **mode bloc**.
  - Utilise des **switchs, routeurs et Host Bus Adapters (HBA)**.
  - Intègre des **baies de disques RAID** pour la redondance.
  - Les échanges utilisent le protocole **SCSI**.
  - Conçu pour les **environnements critiques à haute performance**.



## 6/Protocoles de communication SAN

### ■ SCSI (Small Computer System Interface)

- Interface pour connecter ordinateurs et périphériques en **mode bloc**.
- Communication **locale** via un **bus parallèle ou série**.
- **Non adaptée** aux longues distances ou réseaux étendus.

### ■ iSCSI (Internet SCSI)

- Extension de **SCSI** pour réseaux IP.

- Transporte les blocs SCSI via le **protocole TCP/IP** (encapsulation dans des paquets TCP).
- Permet de construire un **SAN sans matériel spécifique** (switchs Fibre, câbles spéciaux...).
- **Facile à déployer** mais **vitesse limitée** (20–40 Mo/s), donc peu adaptée aux applications très exigeantes.

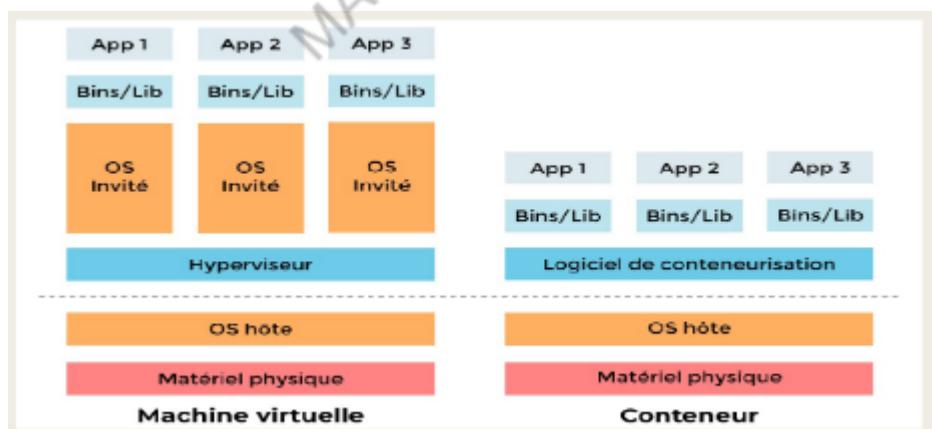
## ■ FCP (Fibre Channel Protocol)

- Utilise le réseau **Fibre Channel** pour transporter des commandes SCSI.
- Transfert de **données en blocs à haute vitesse**, sans perte.
- Peut atteindre jusqu'à **400 Mo/s**.
- Idéal pour les **environnements critiques et très performants**.

## II/ Introduction à la conteneurisation (Docker) :

### 1. Conteneur :

Un **conteneur** est une unité logicielle autonome qui s'exécute sur un système d'exploitation hôte. Il embarque tout le nécessaire pour faire fonctionner une application : code, fichiers binaires, bibliothèques et dépendances.



L'image oppose deux technologies d'isolation logicielle :

1. **Machines Virtuelles (VM)**
  - Chaque VM a son **OS invité complet** (couche lourde).
  - Exécutées sur un **hyperviseur** (ex : VMware).
  - **Isolation forte** mais consommation élevée de ressources.
2. **Conteneurs** (ex : Docker)

- Partagent l'**OS hôte** (pas de double OS).
- Isolent les apps via un **moteur de conteneurisation**.
- **Légers & rapides**, idéaux pour le cloud.

**Schéma clé :**

- **VM** : App → Bins/Lib → OS invité → Hyperviseur → OS hôte → Matériel.
- **Conteneur** : App → Bins/Lib → Moteur de conteneurs → OS hôte → Matériel.

**Avantages :**

- **VM** : Sécurité, compatibilité multi-OS.
- **Conteneurs** : Efficacité, déploiement agile.

*Exemple :*

- Une VM héberge un **Windows complet** sur Linux.
- Un conteneur exécute **une app Node.js** sans OS supplémentaire.

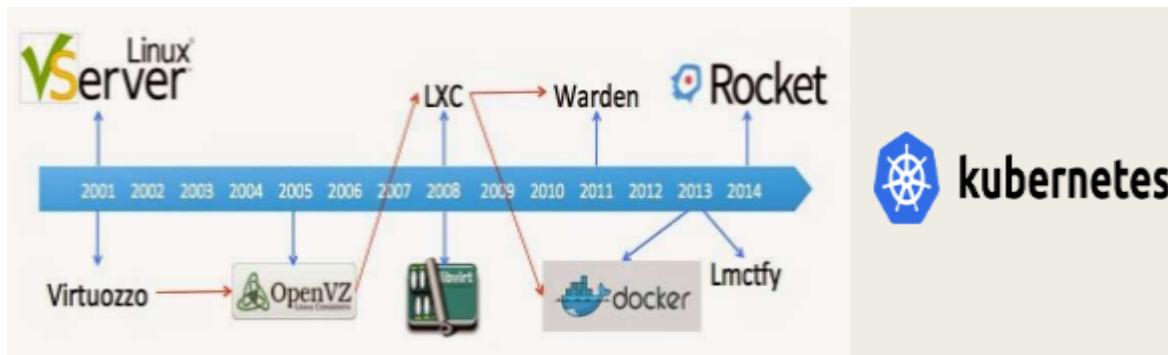
→ **VM = isolation totale | Conteneurs = légèreté & rapidité.**

---

## **2. Avantages des conteneurs :**

- **Légèreté** : Ils ne contiennent que les éléments essentiels, sans surcharge inutile.
  - **Performance** : Plus rapides à exécuter que les machines virtuelles.
  - **Portabilité** : Faciles à déplacer entre différents environnements (idéal pour le multi-cloud).
  - **Uniformité** : Offrent un environnement unifié pour le développement, les tests et la production.
  - **Modularité** : Facilitent la gestion d'applications composées de plusieurs services.
  - **Optimisation des ressources** : Permettent une densité plus élevée grâce à une meilleure utilisation des ressources système.
-

### 3. Solution de conteneurisation :



Linux a popularisé la conteneurisation, passant d'**OpenVZ** (noyau partagé modifié, 2001-2016) à **Docker** (standard portable, depuis 2013), grâce aux *cgroups/namespaces*. Docker domine avec son écosystème (Kubernetes, CI/CD), tandis que **LXC/Podman** offrent des alternatives légères. Aujourd'hui, les conteneurs sont indispensables pour le cloud, les microservices et l'IA, avec une évolution vers plus de sécurité (gVisor) et de simplicité (sans démon).

*Exemple :* OpenVZ pour les VPS économiques, Docker pour les apps cloud.

---

### 4. La Conteneurisation sous Docker

**Docker** est une plateforme qui permet de créer, déployer et exécuter des applications dans des **conteneurs** légers et isolés.

#### 🔧 Architecture de Docker :

- **Docker Engine** : Le cœur de Docker, il exécute les conteneurs, assure leur isolation, leur sécurité et la gestion des ressources.
- **Docker Client** : Interface utilisateur (en ligne de commande ou graphique) qui envoie des commandes au Docker Engine.
- **Image** : Modèle en lecture seule d'un conteneur. Elle peut être créée localement ou téléchargée depuis un registre.
- **Conteneur** : Instance d'exécution isolée basée sur une image, contenant tout ce nécessaire pour faire tourner une application.
- **Registry (registre)** : Système de stockage (local ou distant) où sont hébergées les images, accessible publiquement ou en privé.

# CHAPITRE 5 : Les traitements de données avec SPARK

## 1. Les principaux types de traitements en Big Data :

### 1. Traitement Batch (par lots)

- **Définition** : Le traitement par lots consiste à **collecter et stocker de grandes quantités de données**, puis à les traiter **périodiquement** dans une "fenêtre de batch".
- **Caractéristique** : Nécessite que les données soient **complètement disponibles** avant le traitement.
- **Utilisation** : Idéal pour les analyses historiques, les rapports périodiques, ou les processus ne nécessitant pas de résultat immédiat.



L'image illustre un **processus de gestion des données** entre la mémoire et le stockage (HDD), mettant en avant deux concepts clés :

1. **Sérialisation/Désérialisation**
  - **Sérialisation** : Conversion des données (objets, structures) en un format linéaire (ex: binaire, JSON) pour le stockage (**HDD**) ou le transfert (**Entrée/Sortie**).
  - **Désérialisation** : Reconstruction des données originales depuis le format sérialisé.
2. **RéPLICATION**
  - Copie des données sérialisées vers d'autres supports/nœuds pour la **redondance** ou l'accès distribué.

**Exemple concret :**

- Un objet Python est **sérialisé** en JSON → écrit sur le **disque dur (HDD)** → **répliqué** vers un backup → **désérialisé** pour être réutilisé.

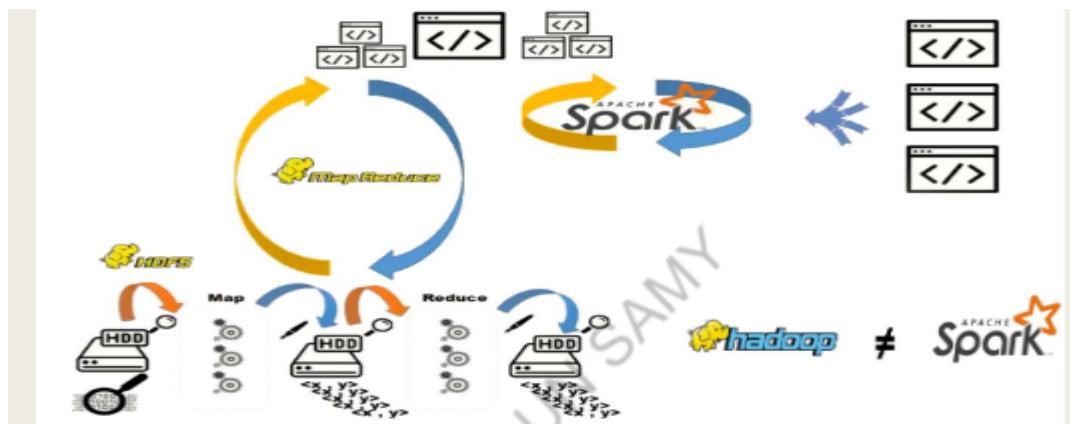
**Utilité :**

- Persistance des données (bases de données, sauvegardes).
- Communication réseau (envoi de données entre services).

*En bref : Transformer → Stocker/Transférer → Reconstruire → Dupliquer*

## 2. Traitement Stream (en flux)

- **Définition** : Le traitement en flux analyse les données **en temps réel, au fur et à mesure** de leur arrivée.
- **Caractéristique** : Ne nécessite pas d'attendre la fin de la collecte des données ; permet une **réactivité immédiate**.
- **Utilisation** : Surveillance en temps réel, détection de fraudes, IoT, recommandations instantanées.



## 3. Comparaison Batch vs Stream

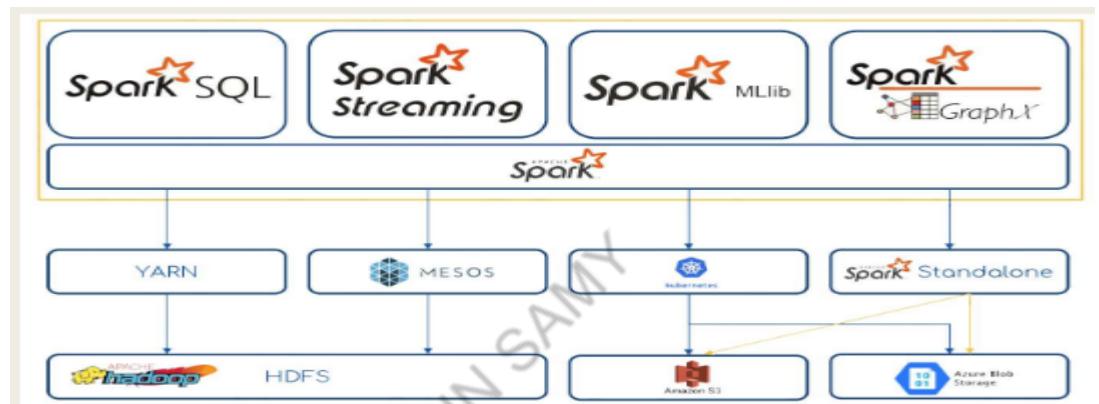
Critère	Traitement Batch	Traitement Stream
Volume de données	Très grand (stockées avant traitement)	Flux continu (traité en direct)
Latence	Élevée (traitement différé)	Faible (résultats instantanés)
Complexité	Moins complexe	Plus complexe techniquement
Cas d'usage typique	Statistiques, rapports, historique	Alertes, données temps réel

## 2. Le Framework Apache Spark :

## 1. Présentation générale

- **Créateur** : Matei Zaharia (2009, à l'université).
- **Objectif** : Accélérer les traitements Big Data, en alternative à MapReduce.
- **Atout** : Plus performant que Hadoop MapReduce, notamment grâce au traitement en mémoire.
- **Statut** : Projet Apache depuis 2013, avec plus de **1000 contributeurs**.

## 2. Composants principaux de Spark



### a. Spark Core

- Cœur du framework Spark.
- Gère :
  - **Planification des tâches**
  - **Gestion de la mémoire**
  - **Tolérance aux pannes**
  - **Accès aux systèmes de stockage**
  - **RDD (Resilient Distributed Dataset)** : structure de données fondamentale.

### b. Spark SQL

- Permet d'exécuter des requêtes **SQL** sur les données dans Spark.
- Supporte plusieurs langages : **Java, Scala, Python, R**.

- Compatible avec divers formats et sources : **JSON, Parquet, JDBC, Hive**.

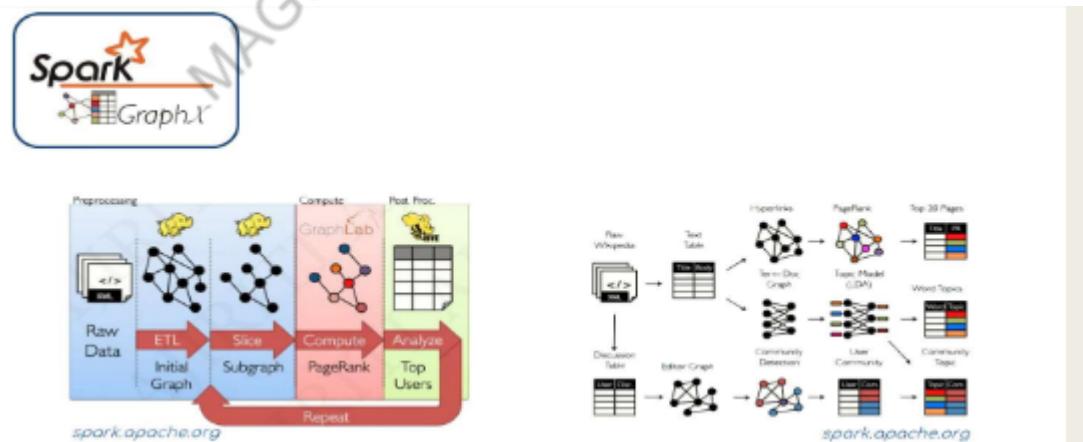
### c. Spark Streaming

- Composant pour le **traitement des données en flux**.
- Utilise des **DStreams (Discretized Streams)** = séquences de RDDs.
- Adapté au **temps réel** ou au **quasi temps réel**.

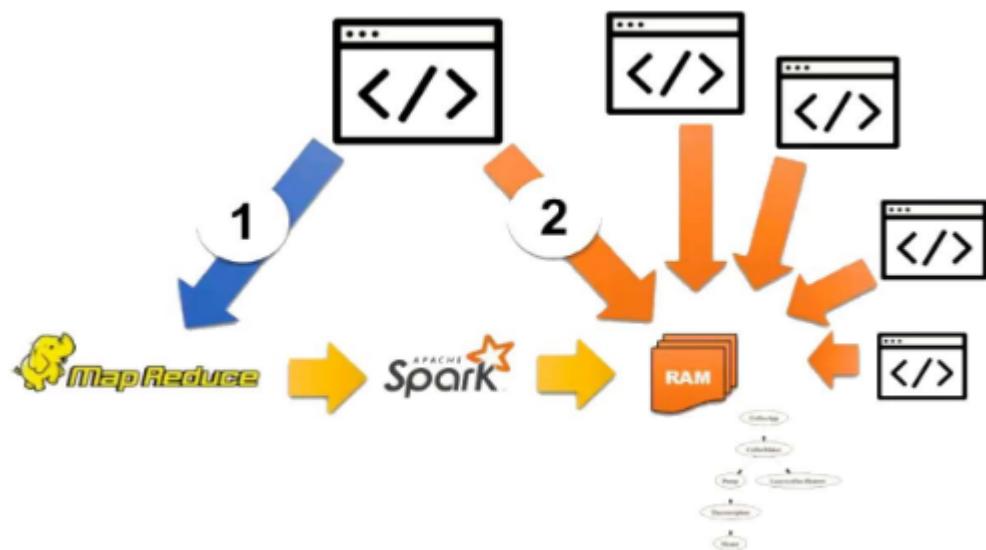


### d. GraphX

- API dédiée au traitement et à l'analyse de **données sous forme de graphes**.
- Permet de modéliser des relations complexes (ex : réseaux sociaux, recommandations...).



### 3. Caractéristique d'Apache Spark :



L'image compare **Hadoop MapReduce** (1) et **Apache Spark** (2) en illustrant la différence dans la gestion des traitements.

#### Explication de l'image :

##### ① Hadoop MapReduce (flèche bleue) :

- Fonctionne par **étapes successives** : chaque tâche lit et écrit sur le disque, ce qui ralentit le traitement.
- Adapté aux **traitements batch**, mais moins efficace pour le temps réel.

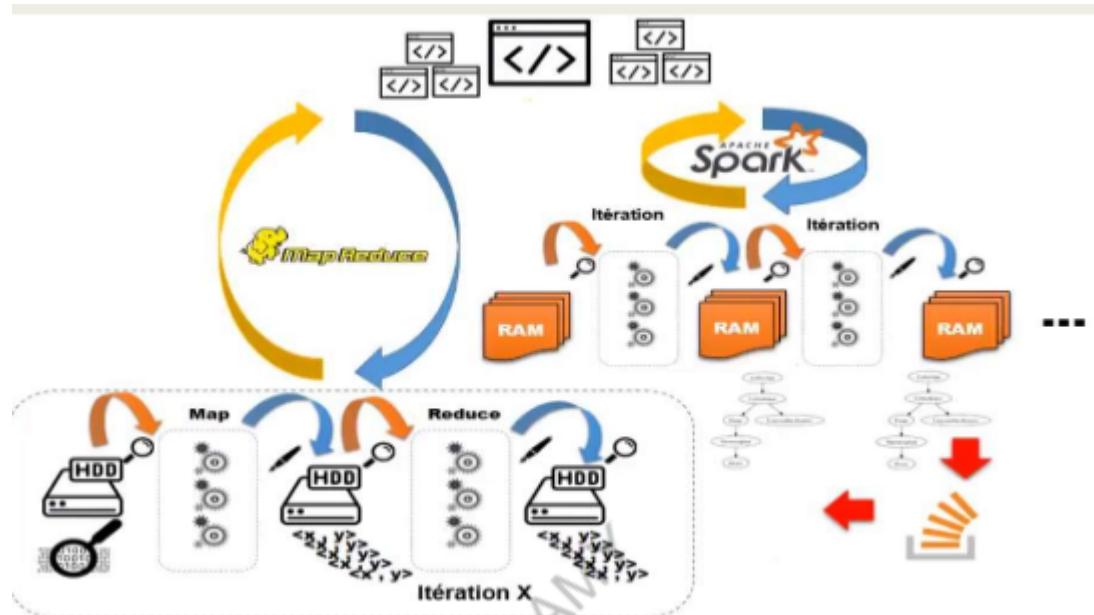
##### ② Apache Spark (flèche orange) :

- Utilise la **mémoire RAM** pour exécuter plusieurs tâches en **parallèle**.
- Plus rapide que MapReduce grâce au **traitement en mémoire (in-memory computing)**.
- Meilleur pour les **données en flux (streaming)** et les **analyses complexes** (Machine Learning, SQL, Graphes).

#### Caractéristiques clés d'Apache Spark :

- ✓ **Traitement en mémoire** → Plus rapide que Hadoop.
- ✓ **Exécution parallèle** → Gère plusieurs tâches simultanément.
- ✓ **Supporte plusieurs langages** → Scala, Python, Java, R.
- ✓ **Composants intégrés** → Spark SQL, Spark Streaming, MLlib, GraphX.

**En résumé :** Apache Spark est plus performant que Hadoop MapReduce, car il exploite la **mémoire vive** au lieu d'écrire constamment sur disque. Il est idéal pour les **traitements rapides, interactifs et en temps réel**.



L'image compare les mécanismes d'exécution **itératif** entre **MapReduce** et **Apache Spark**, en illustrant clairement leurs différences de performance et d'efficacité.

### ● Côté gauche – Hadoop MapReduce :

- Chaque **itération** (Map puis Reduce) lit et écrit les données sur le **disque dur (HDD)**.
- Cela engendre :
  - Une **lenteur importante** due aux accès disque.
  - Une **inefficacité** pour les algorithmes itératifs comme ceux utilisés en **machine learning** ou en **traitement de graphes**.
- Les flèches circulaires bleues et jaunes montrent le **retour constant au disque dur** à chaque étape.

### ● Côté droit – Apache Spark :

- Utilise la **RAM (mémoire vive)** pour stocker les données intermédiaires entre les itérations.
- Avantages :

- Traitements **plus rapides** (jusqu'à 100x plus rapide que MapReduce).
- Adapté aux traitements **itératifs**, interactifs, ou en **temps réel**.
- Spark optimise l'exécution par :
  - Le **caching en mémoire**.
  - Une **exécution en pipeline** fluide et continue.

## Résumé des caractéristiques d'Apache Spark :

- Traitement en mémoire (**RAM**)
  - Itérations rapides sans aller-retour disque
  - Efficace pour les boucles, les graphes, et le ML
  - Moins de latence que MapReduce
  - Architecture moderne pour le Big Data interactif
- 

## 4. Architecture générale d'Apache Spark :

### 1. Driver (Pilote)

- C'est le point de départ de l'application Spark.
- Il exécute le programme principal et crée un contexte Spark (**SparkContext**), qui gère la communication avec le cluster.

### 2. SparkContext

- Initialise l'application Spark.
- Envoie les tâches aux Executors sur les nœuds du cluster.

### 3. Master (Maître)

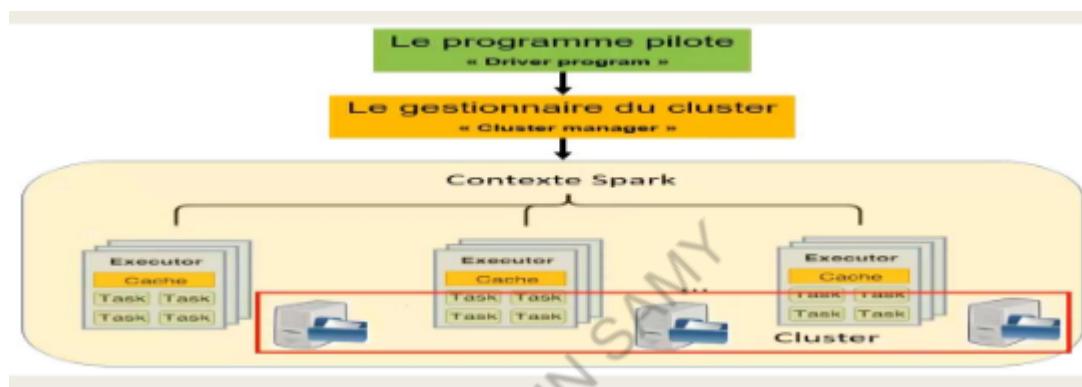
- Coordonne l'exécution globale.
- Il divise les données et les traitements en tâches parallèles.

### 4. Cluster Nodes (Nœuds du cluster)

- Ce sont les machines physiques/virtuelles qui participent au traitement.

## 5. Executors (Exécuteurs)

- Chaque nœud exécute un ou plusieurs Executors.
- Un Executor est responsable de :
  - L'exécution des tâches (tasks) attribuées par le driver.
  - Le stockage des données intermédiaires (ex : en mémoire).
  - Le retour des résultats au driver.



### 💡 Fonctionnement global :

- Le driver pilote l'application.
- Le master répartit les tâches sur les nœuds.
- Les executors sur les nœuds exécutent les tâches en parallèle.

---

## 5. Avantages d'Apache Spark

### 1. Rapidité

- Traitement des données **en mémoire (RAM)**, ce qui le rend **jusqu'à 100 fois plus rapide** que Hadoop MapReduce.

### 2. Analyse avancée

- Supporte des traitements complexes comme :
  - **Le machine learning**
  - **Le traitement de graphes**

## ■ Le traitement de flux (streaming)

### 3. Support multi-langage

- Spark permet de coder en **Java, Scala, Python et R**, ce qui le rend accessible à un grand nombre de développeurs.
- 

## 6. RDDs (Resilient Distributed Dataset) sous Spark :

### ◆ 1. Définition des RDDs

- **Immutable** : Une fois créé, un RDD ne peut pas être modifié.
- **Lazy** : Les transformations sont évaluées uniquement lors d'une action.
- **Cacheable** : Un RDD peut être mis en mémoire (cache) pour accélérer les traitements.

### ◆ 2. Distribution et Partitionnement

- Un RDD est **partitionné** à sa création.
- Le **nombre de partitions** dépend :
  - du **nombre de workers**
  - de la **configuration du cluster**
  - du **type de système de fichiers (ex : HDFS)**

### ◆ 3. Transformations vs Actions

- **Transformations** : opérations **paresseuses (lazy)** qui créent de nouveaux RDDs (ex : `map`, `filter`)
- **Actions** : déclenchent réellement le **calcul** et renvoient un résultat (ex : `count()`, `collect()`)

### ◆ 4. SparkContext

- Le **SparkContext (sc)** est utilisé pour **créer des RDDs** de différentes manières :
  - **À partir d'un fichier :**  
`rdd1 = sc.textFile('/home/cloudera/fichier.txt')`

- **Paralléliser une collection :**

```
liste = ['nombre1', 'nombre2']
```

```
rdd_liste = sc.parallelize(liste)
```

- **Transformer un RDD existant :**

```
rdd_transf = rdd_liste.map(lambda x: (x, 1))
```

## 5. Actions courantes

- **collect()** : renvoie tous les éléments sous forme de liste.
- **count()** : compte le nombre d'éléments.
- **first()** : renvoie le premier élément.
- **take(n)** : renvoie les  $n$  premiers éléments.