



**Support de cours**

## **BIG DATA & BASES DE DONNÉES NOSQL**



## **CHAPITRE II: ARCHITECTURE ET ÉCOSYSTÈME BIG DATA**

# Plan

- **INTRODUCTION**
- **CONCEPTS DE SYSTÈMES DISTRIBUÉS**
- **APACHE HADOOP**
- **APACHE SPARK**
- **NOTIONS DE BASE SUR L'ARCHITECTURE BIG DATA**
- **BASES DU STREAMING**

# Introduction

L'explosion des données générées par les activités humaines, les objets connectés, les transactions en ligne et les réseaux sociaux a conduit à un défi majeur : **comment stocker, gérer et analyser efficacement ces volumes massifs de données?** Les approches traditionnelles, basées sur un seul serveur ou des bases de données relationnelles classiques, montrent rapidement leurs limites.

→ En effet, lorsqu'on atteint des téraoctets, des pétaoctets ou plus encore, une seule machine ne peut ni contenir ni traiter l'ensemble de ces données dans des délais raisonnables.

→ Pour répondre à ces besoins, l'informatique a évolué vers des architectures distribuées, capables de répartir la charge de travail sur un grand nombre de machines. Cette logique est à la base du **Big Data**, et elle repose sur un écosystème riche de technologies complémentaires.



# Introduction

Le Big Data ne peut pas être traité efficacement avec une seule machine. En effet, les volumes de données dépassent rapidement les capacités de stockage et de calcul d'un serveur unique. C'est pourquoi l'architecture Big Data repose sur des **systèmes distribués**.

## □ Ce chapitre aborde

- Les **principes fondamentaux des systèmes distribués** (scalabilité, tolérance aux pannes, réplication).
- L'outil historique **Hadoop**, avec ses sous-composants : HDFS, YARN et MapReduce.
- L'évolution vers **Apache Spark**, moteur de traitement rapide en mémoire.
- Les **Data Lakes** et pipelines ETL, qui facilitent l'ingestion et l'organisation des données massives.
- Les **bases du streaming**, indispensables pour traiter des flux de données en temps réel.

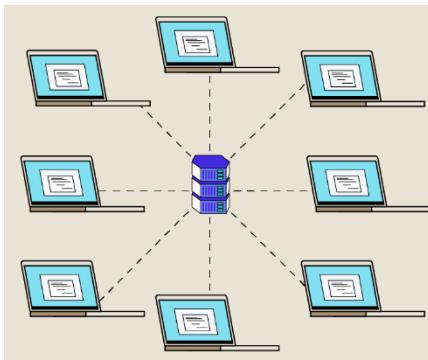
# Concepts de systèmes distribués

## → Définition

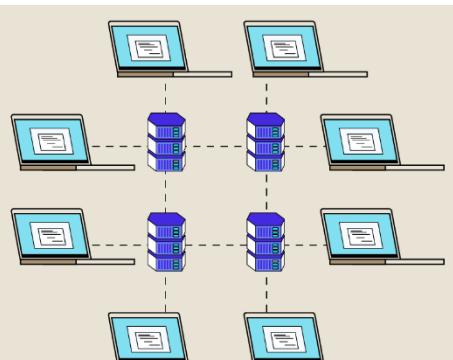
Un **système distribué** est un ensemble de machines (serveurs ou nœuds) qui travaillent ensemble comme s'il s'agissait d'un seul système cohérent. L'idée est de répartir les données et les calculs sur plusieurs ordinateurs afin d'augmenter la capacité globale.

### Exemple:

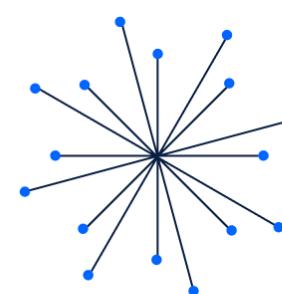
Imaginez une seule personne qui doit porter 100 sacs de ciment. Impossible!! Mais si 100 personnes portent chacun 1 sac, la tâche devient faisable rapidement. C'est exactement le rôle d'un système distribué : diviser pour mieux traiter.



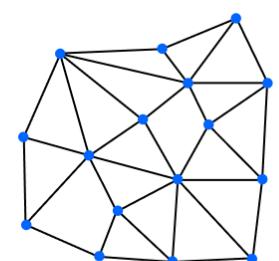
Système Centralisé



Système Distribué



Centralized



Distributed

# Concepts de systèmes distribués

## □ Pourquoi les systèmes distribués sont-ils nécessaires dans le Big Data ?

- **Volume massif** : les bases classiques ne peuvent plus stocker des données dépassant plusieurs téraoctets sur une seule machine.
- **Vitesse** : les traitements doivent être faits en quelques secondes ou minutes, pas en jours.
- **Fiabilité** : les machines tombent en panne ; un système distribué doit continuer à fonctionner malgré cela.
- **Flexibilité** : possibilité d'ajouter facilement des serveurs quand la quantité de données augmente (scalabilité horizontale).

# Concepts de systèmes distribués

## → Concepts clés

### □ Scalabilité

La scalabilité est la capacité d'un système à grandir et à gérer plus de données ou plus de requêtes, sans perte de performance.

- **Scalabilité verticale** : augmenter la puissance d'une seule machine (plus de RAM, plus de CPU). → Limité et coûteux.
- **Scalabilité horizontale** : ajouter plusieurs machines simples (clusters de serveurs).  
→ C'est ce qui est utilisé en Big Data.

### Exemple

- Une base de données classique (Oracle, MySQL) qu'on héberge sur une seule machine est scalable verticalement.
- Hadoop ou Spark fonctionnent sur des dizaines/centaines de serveurs reliés entre eux : c'est la scalabilité horizontale.

# Concepts de systèmes distribués

## → Concepts clés

### □ Tolérance aux pannes

Un système distribué doit continuer à fonctionner même si une partie de ses nœuds échoue.

- Les données et les calculs sont répliqués (dupliqués) sur plusieurs machines.
- Si un serveur tombe en panne, un autre prend le relais.

### Exemple

Dans un cluster Hadoop, si un serveur qui stocke un bloc de données tombe, les autres serveurs qui possèdent une copie de ce bloc garantissent que le système continue de fonctionner normalement.

# Concepts de systèmes distribués

## → Concepts clés

### □ RéPLICATION

La réPLICATION consiste à conserver plusieurs copies d'une donnée sur des machines différentes.

- Elle augmente la **séCURITÉ** (on ne perd pas les données si une machine tombe).
- Elle permet aussi d'améliorer les **PERFORMANCES** (accès à la copie la plus proche ou la plus disponible).

### Exemple

:

Dans HDFS, chaque bloc de donnée est stocké **3 fois par défaut** : une copie principale et deux copies secondaires.

# Concepts de systèmes distribués

## → Caractéristiques des systèmes distribués

- **Transparence** : l'utilisateur interagit avec le système comme si c'était une seule machine.
- **Partage des ressources** : CPU, mémoire, disques sont mis en commun.
- **Concurrence** : plusieurs utilisateurs/programmes peuvent accéder aux données en même temps.
- **Hétérogénéité** : les machines du cluster peuvent avoir des configurations matérielles différentes.

# Concepts de systèmes distribués

## Exemple

Supposons que l'université doive analyser les résultats des examens de **50 000 étudiants** répartis sur 10 années.

- Si on utilise un seul ordinateur, le traitement prendra des heures/jours.
- Avec un cluster de 10 machines, on divise les données en 10 parties : chaque machine calcule les moyennes, puis les résultats sont regroupés.
- Temps réduit de façon drastique : au lieu de 10h, on peut descendre à quelques minutes.

# Concepts de systèmes distribués

## Exercice 01

Un fichier de 1 Go est découpé en blocs de 128 Mo. Le facteur de réPLICATION est 3.

- Combien de blocs au total sont créés ?
- Combien de copies seront stockées dans le cluster ?

## Exercice 02

On a un site web de vidéos qui reçoit chaque jour **50 To** de nouvelles données.

Explique pourquoi un système classique (un seul serveur) est insuffisant et comment un système distribué peut résoudre ce problème.

# Apache Hadoop



# Apache Hadoop

**Hadoop** est un framework open source développé par la fondation **Apache**, initialement créé par **Yahoo!** et inspiré des travaux de **Google** (Google File System et MapReduce). Il est conçu pour stocker et traiter de grandes quantités de données de manière distribuée et efficace. Son principal objectif est de permettre le traitement parallèle de volumes massifs de données sur un cluster de machines standards (*commodity hardware*).



À la différence des systèmes traditionnels où les données sont centralisées sur un seul serveur, Hadoop repose sur une **architecture distribuée** : les données sont **fragmentées et répliquées** sur plusieurs nœuds, et le traitement est exécuté localement sur ces nœuds, ce qui améliore les performances et la tolérance aux pannes.

# Apache Hadoop

## ❑ Pourquoi Hadoop est important ?

- Il permet de traiter des données qu'aucune base classique ne peut gérer.
- Il repose sur deux idées fondamentales :
  - **Stocker les données en blocs répartis et répliqués (HDFS).**
  - **Exécuter des calculs en parallèle directement là où les données sont stockées (MapReduce).**
- Il a un **gestionnaire de ressources** pour coordonner les calculs (YARN).

## ❑ L'objectif principal de Hadoop est de:

- Stocker des données massives (structurées ou non) de façon fiable.
- Traiter ces données en parallèle sur plusieurs nœuds.
- Offrir une scalabilité horizontale (ajout facile de nouvelles machines).
- Réduire le coût d'infrastructure en utilisant des serveurs ordinaires.

# Apache Hadoop

## □ Composants principaux de Hadoop

- **HDFS** (Hadoop Distributed File System) : système de stockage distribué.
- **YARN** (Yet Another Resource Negotiator) : gestionnaire de ressources et ordonnancement.
- **MapReduce** : modèle de calcul distribué en mode batch.
- Écosystème étendu : Hive, Pig, Spark, HBase, etc.

Hadoop est aujourd’hui utilisé dans de nombreux domaines : analyse de logs, machine learning distribué, traitement de données IoT, systèmes de recommandation, etc.

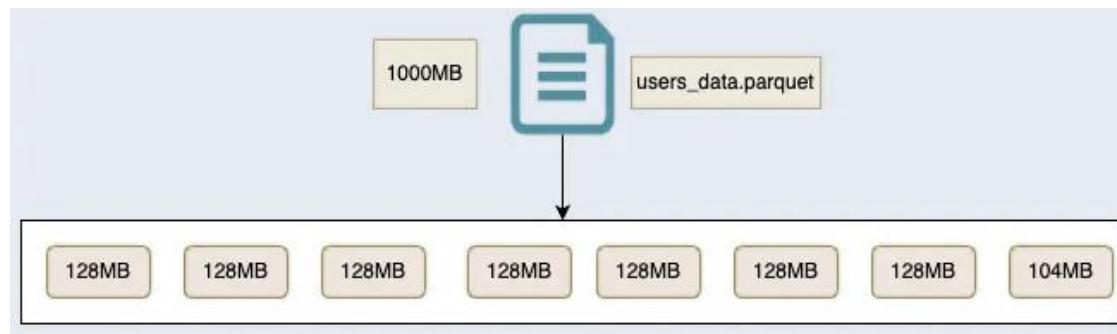


# Apache Hadoop

## → **HDFS (Hadoop Distributed File System)**

**HDFS** est le système de fichiers distribué de Hadoop, conçu pour stocker de très gros volumes de données (de plusieurs Go à plusieurs To) en les découplant en blocs et en les répliquant sur plusieurs nœuds (machines). Il garantit la tolérance aux pannes et la disponibilité.

- Il divise un gros fichier (exemple 1 Go) en **blocs** (par défaut 128 Mo), puis stocke ces blocs sur différentes machines du cluster.
- HDFS a une architecture **maître/esclave**. Sur le nœud **maître**, s'exécutent les démons **NameNode**, et sur les nœuds **esclaves**, s'exécutent les démons **DataNode**.



# Apache Hadoop

## → HDFS (Hadoop Distributed File System)

### □ Architecture de HDFS

HDFS adopte une architecture maître-esclave (master/slave) :

- **NameNode**

- C'est le maître du cluster HDFS.
- Il garde et stocke la métadonnée (où chaque bloc est stocké, qui a quelle copie, arborescence des fichiers, emplacement des blocs, permissions... etc.).
- Sans lui, le cluster est inutilisable (mais il peut être sauvegardé avec un **Secondary NameNode**).
- Il ne stocke pas les données elles-mêmes.

- **DataNode**

- C'est le nœud esclave.
- Il stocke les blocs de données réels.
- Les DataNodes envoient périodiquement un heartbeat (signal qu'ils sont vivants) au NameNode pour indiquer leur disponibilité.

# Apache Hadoop

## → HDFS (Hadoop Distributed File System)

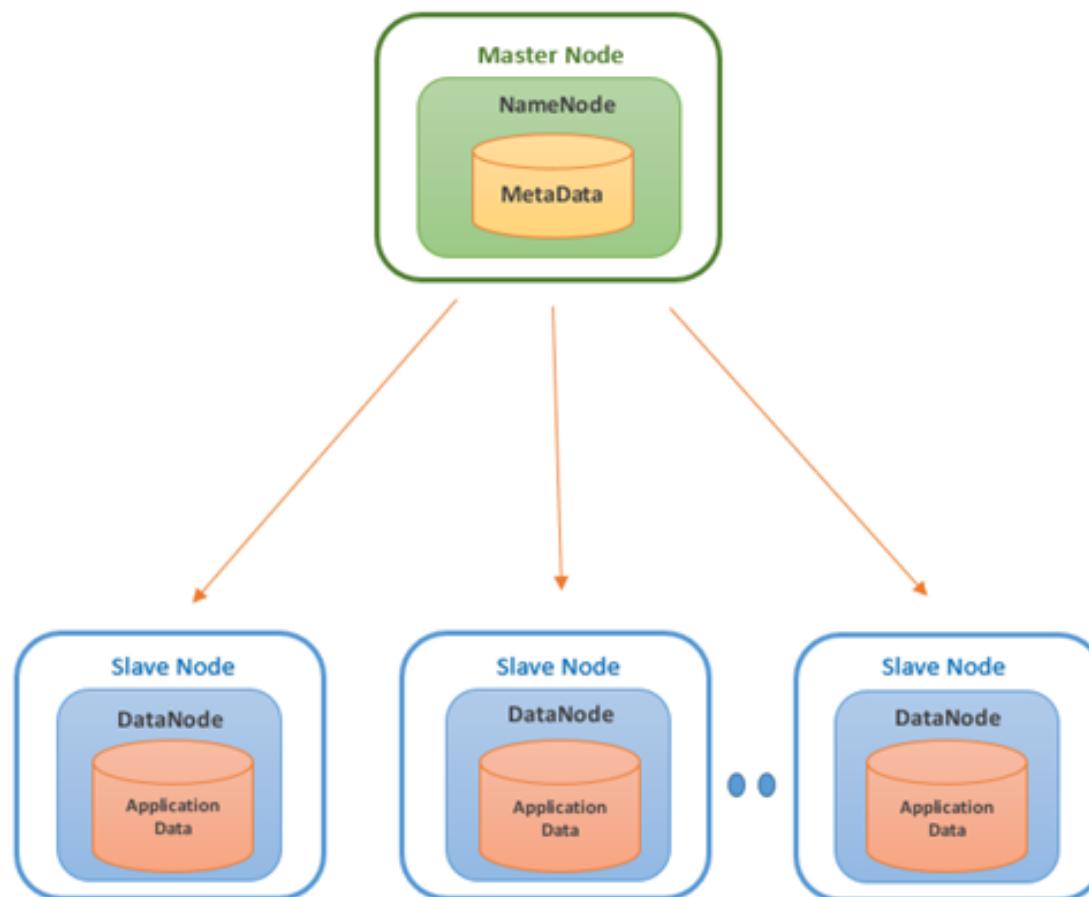
### □ Architecture de HDFS

- **Le Secondary NameNode** ou le Checkpoint Node :
  - Ne remplace pas le *NameNode*.
  - Le NameNode conserve une image des métadonnées (fsimage) et un journal des modifications (edits). Le Secondary NameNode / Checkpoint node aide à fusionner fsimage et edits pour éviter que le journal ne devienne trop volumineux.
  - Il fait périodiquement une sauvegarde de la métadonnée pour soulager le NameNode.
  - Il aide à reconstruire les informations en cas de panne.
- **Backup Node**
  - Le Backup Node peut maintenir une copie en mémoire du *namespace* pour être prêt à prendre le relais.

**Remarque :** Le *Secondary NameNode* ne doit pas être confondu avec un **Backup Node**. Le premier effectue une fusion périodique des métadonnées (fsimage et edits logs), tandis que le second maintient une copie à jour du NameNode et peut le remplacer en cas de panne.

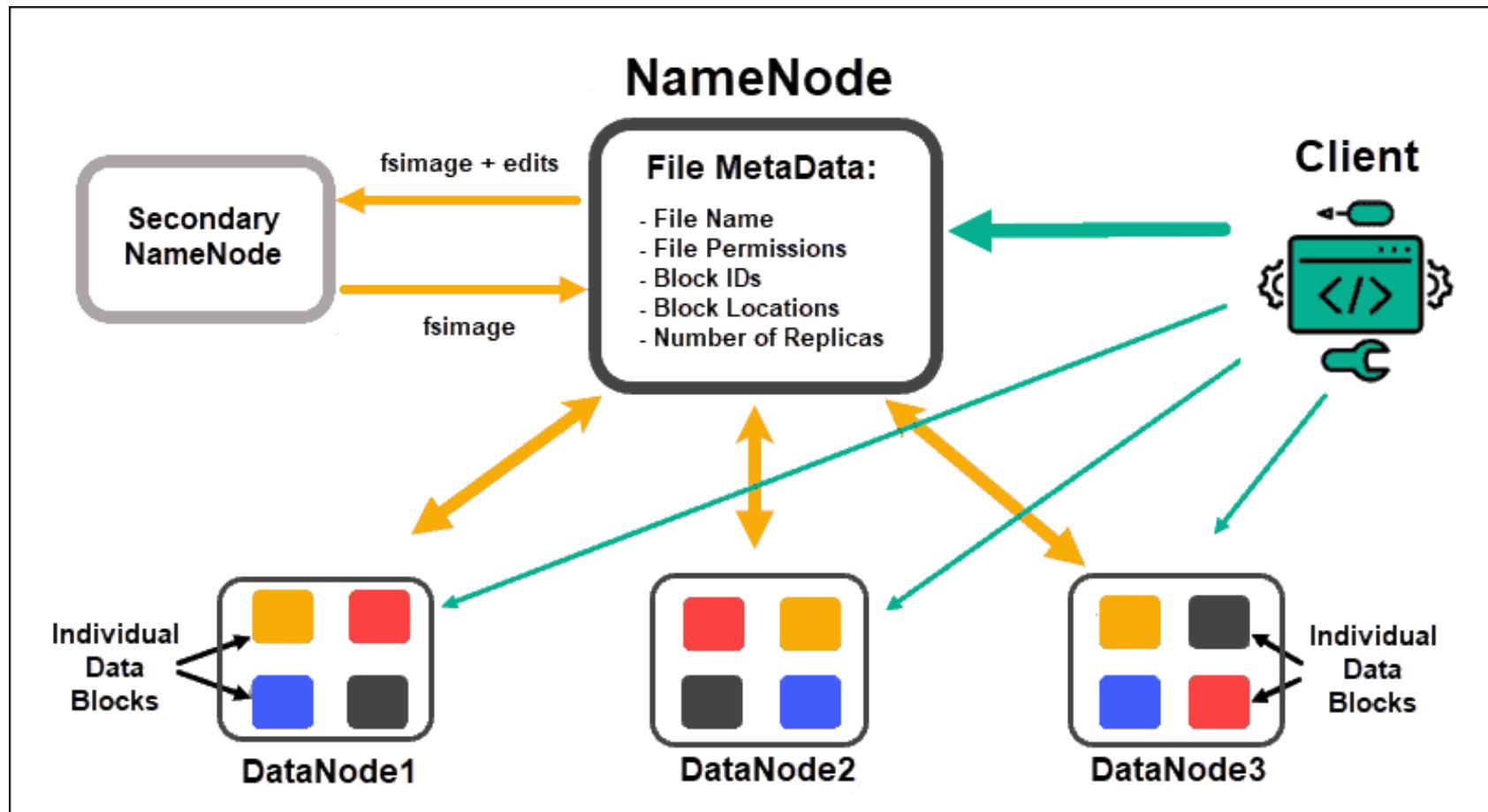
# Apache Hadoop

→ HDFS (Hadoop Distributed File System)



# Apache Hadoop

## → HDFS (Hadoop Distributed File System)



# Apache Hadoop

## → HDFS (Hadoop Distributed File System)

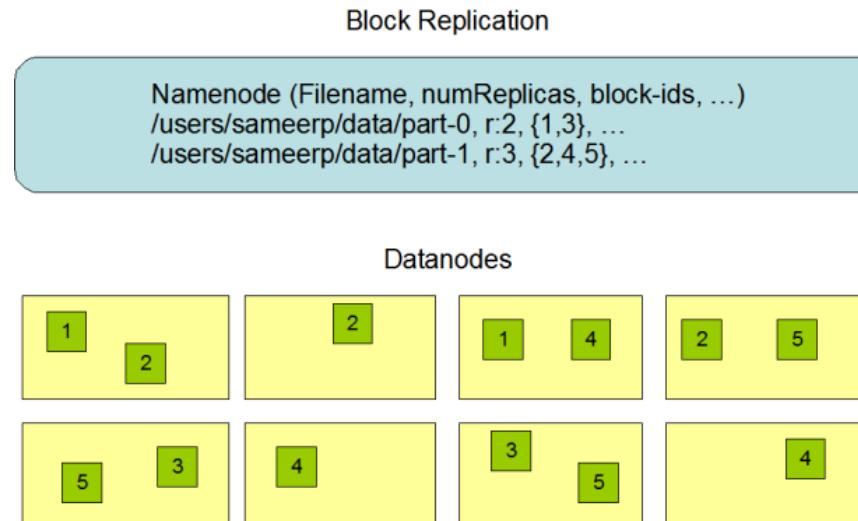
### □ Blocs de données

- Les fichiers sont découpés en blocs (par défaut 128 Mo dans Hadoop 3.x).
- Chaque bloc est stocké sur plusieurs DataNodes.

### □ RéPLICATION

- Chaque bloc est stocké **3 fois** par défaut (facteur de réPLICATION = 3).

Exemple : un fichier de 1 Go → découpé en 8 blocs de 128 Mo → chaque bloc a 3 copies → total de 24 blocs stockés dans le cluster.



# Apache Hadoop

## → HDFS (Hadoop Distributed File System)

### ❑ Rack Awareness (conscience du rack)

HDFS prend en compte la topologie physique du cluster :

- Un rack correspond à un ensemble de serveurs connectés au même commutateur réseau.
- Hadoop optimise la réPLICATION en plaçant les blocs dans différents racks pour améliorer :
  - La résilience aux pannes réseau.
  - La bande passante.

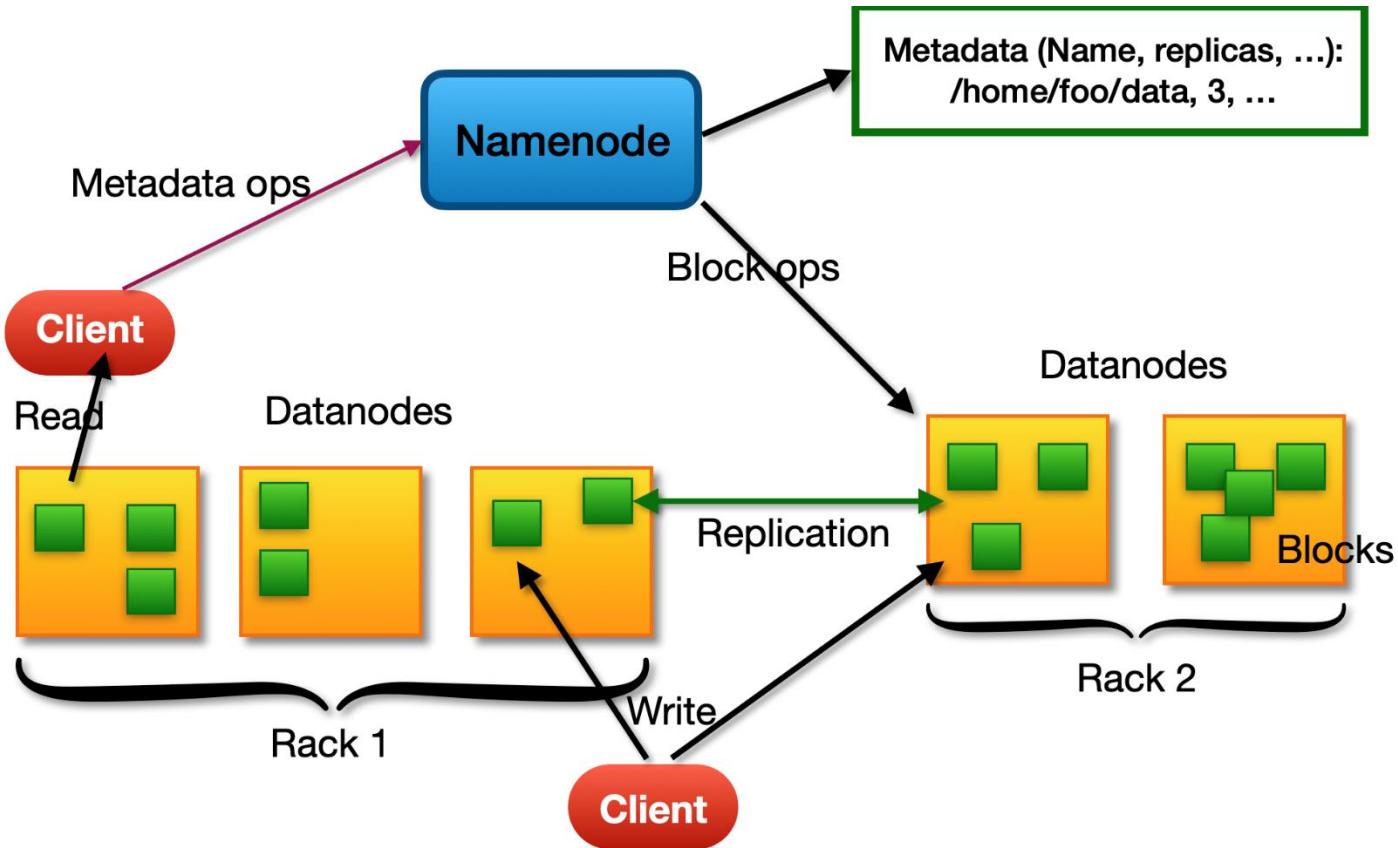
### Exemple de stratégie :

- 1ère copie dans le même rack que le client.
- 2ème copie dans un autre rack.
- 3ème copie dans le même rack que la 2ème.

→ Cela améliore la **résilience** et **réduit les risques de perte de données** si un rack entier tombe en panne.

# Apache Hadoop

## → HDFS (Hadoop Distributed File System)



# Apache Hadoop

## → **HDFS (Hadoop Distributed File System)**

### □ Fonctionnement d'une écriture dans HDFS

1. Le client contacte le NameNode pour demander où écrire.
2. Le NameNode répond avec une liste de DataNodes choisis pour stocker les blocs.
3. Le client envoie le bloc au premier DataNode, qui le transfère en pipeline au suivant, etc.
4. Une fois l'écriture terminée, les DataNodes confirment la réussite.
5. Le NameNode met à jour sa métadonnée.

### □ Fonctionnement d'une lecture dans HDFS

1. Le client demande au NameNode l'emplacement des blocs.
2. Le NameNode renvoie la liste des DataNodes contenant ces blocs.
3. Le client lit directement les blocs depuis le DataNode le plus proche (meilleure bande passante).
4. Les blocs sont réassemblés pour reconstituer le fichier original.

# Apache Hadoop

## → HDFS (Hadoop Distributed File System)

### □ Commandes de base HDFS

HDFS possède un ensemble de commandes permettant de manipuler les fichiers et dossiers, similaires à celles d'un système UNIX classique. Elles s'exécutent depuis le terminal avec la commande **hdfs dfs** (ou **hadoop fs**).

#### 1. Gestion des répertoires

Action	Commande	Exemple
Créer un dossier	<b>hdfs dfs -mkdir /chemin/dossier</b>	<b>hdfs dfs -mkdir /user/hadoop/data</b>
Lister le contenu d'un dossier	<b>hdfs dfs -ls /chemin</b>	<b>hdfs dfs -ls /user/hadoop</b>
Supprimer un dossier	<b>hdfs dfs -rm -r /chemin</b>	<b>hdfs dfs -rm -r /user/hadoop/tmp</b>

# Apache Hadoop

## → HDFS (Hadoop Distributed File System)

### □ Commandes de base HDFS

#### 1. Gestion des fichiers

Action	Commande	Exemple
Copier un fichier local vers HDFS	<code>hdfs dfs -put fichier_local /chemin_hdfs</code>	<code>hdfs dfs -put data.txt /user/hadoop/input</code>
Copier un fichier de HDFS vers local	<code>hdfs dfs -get /chemin_hdfs fichier_local</code>	<code>hdfs dfs -get /user/hadoop/output/result.txt ./</code>
Supprimer un fichier dans HDFS	<code>hdfs dfs -rm /chemin_hdfs</code>	<code>hdfs dfs -rm /user/hadoop/input /data.txt</code>
Afficher le contenu d'un fichier	<code>hdfs dfs -cat /chemin_hdfs</code>	<code>hdfs dfs -cat /user/hadoop/input /data.txt</code>

# Apache Hadoop

## → HDFS (Hadoop Distributed File System)

### □ Commandes de base HDFS

#### 1. Informations et gestion avancée

Action	Commande
Consulter l'espace libre	<code>hdfs dfs -df -h</code>
Consulter la taille d'un fichier ou dossier	<code>hdfs dfs -du -h /chemin</code>
Voir les blocs d'un fichier	<code>hdfs fsck /chemin -files -blocks</code>
Changer les permissions	<code>hdfs dfs -chmod 755 /chemin</code>
Changer le propriétaire	<code>hdfs dfs -chown utilisateur:group /chemin</code>

# Apache Hadoop

## → HDFS (Hadoop Distributed File System)

### □ Commandes de base HDFS

#### 1. Commandes d'administration

Action	Commande	Description
Vérifier l'état du cluster	<code>hdfs dfsadmin -report</code>	Donne la liste des DataNodes, espace disponible et état de la réPLICATION
Lister les nœuds actifs	<code>hdfs dfsadmin -printTopology</code>	Affiche la topologie (racks et DataNodes)
Rééquilibrer les blocs	<code>hdfs balancer</code>	RéPARTIT éQUITABLEMENT les blocs entre nœuds

# Apache Hadoop

→ HDFS (Hadoop Distributed File System)

## □ Commandes de base HDFS

### Exemples

#### 1. Créer un dossier d'entrée :

```
hdfs dfs -mkdir /user/etudiant/input
```

#### 2. Copier un fichier local dans HDFS

```
hdfs dfs -put etudiants.csv /user/etudiant/input/
```

#### 3. Vérifier la présence du fichier

```
hdfs dfs -ls /user/etudiant/input
```

#### 4. Lire le contenu

```
hdfs dfs -cat /user/etudiant/input/etudiants.csv
```

# Apache Hadoop

## → HDFS (Hadoop Distributed File System)

### □ Avantages et limites de HDFS

#### → Les avantages

- Tolérance aux pannes grâce à la réPLICATION.
- Scalable horizontalement.
- Faible coût matériel.
- Lecture/écriture efficace de gros fichiers.

#### → Les limites

- Non optimisé pour les petits fichiers (surcoût de métadonnées).
- Ne supporte pas la modification d'un fichier en place.
- Latence d'écriture plus élevée que sur un disque local.

# Apache Hadoop

→ **YARN (Yet Another Resource Negotiator)**

**YARN** est la **couche de gestion des ressources** dans Hadoop. Elle permet de **partager les ressources du cluster** entre plusieurs applications Big Data.

Il décide :

- quel programme s'exécute,
- où il s'exécute (sur quel nœud),
- combien de ressources il reçoit (CPU, RAM).

## □ Objectifs

- Planifier l'utilisation des ressources.
- Allouer dynamiquement de la mémoire et du CPU.
- Lancer et surveiller les tâches distribuées.

# Apache Hadoop

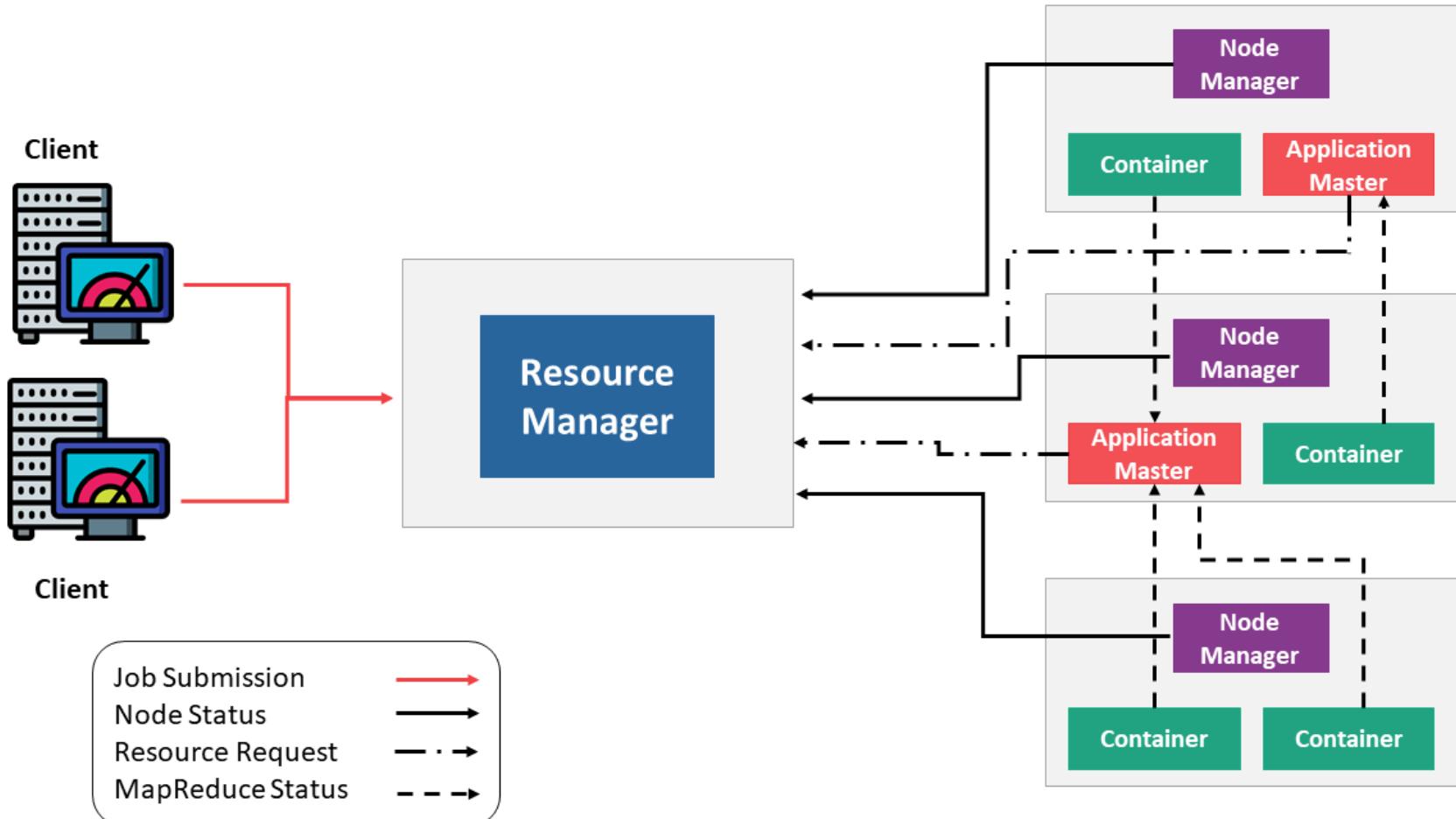
→ **YARN (Yet Another Resource Negotiator)**

## □ Architecture de YARN

- **ResourceManager**
  - Chef d'orchestre du cluster, il gère les ressources globales.
  - Décide où et quand lancer les tâches.
- **NodeManager**
  - Présent sur chaque nœud, il surveille les ressources locales.
  - Exécute les conteneurs et gère l'exécution des tâches.
- **ApplicationMaster**
  - Spécifique à chaque application/programme, il gère son exécution.
  - Négocie les ressources avec le ResourceManager (dialogue avec ResourceManager).
- **Container** : unité d'exécution qui embarque les ressources nécessaires à une tâche.

# Apache Hadoop

→ YARN (Yet Another Resource Negotiator)



# Apache Hadoop

→ **YARN (Yet Another Resource Negotiator)**

## □ Rôle de YARN

- Gérer plusieurs applications en parallèle.
- Optimiser l'utilisation du cluster.
- Permettre d'exécuter différents frameworks (pas seulement MapReduce, mais aussi Spark, Tez, Flink...).

## □ Fonctionnement de YARN (workflow simplifié)

1. Le client soumet une application à YARN.
2. Le **ResourceManager** lance un **ApplicationMaster** sur un **NodeManager**.
3. L'**ApplicationMaster** demande des conteneurs pour exécuter les tâches.
4. Les **NodeManagers** exécutent les tâches dans les conteneurs.
5. L'**ApplicationMaster** informe le **ResourceManager** lorsque tout est terminé.

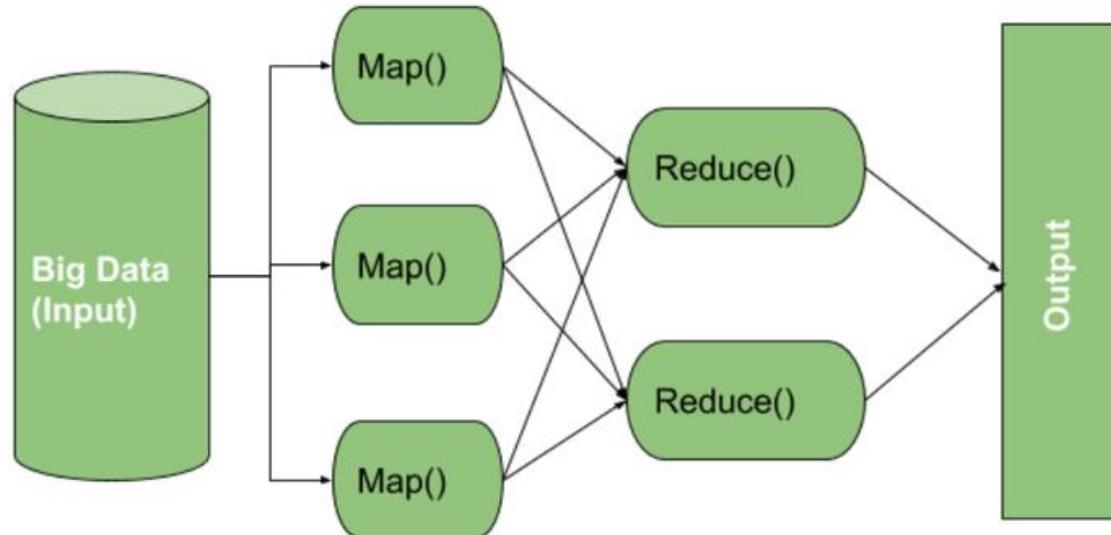
→ YARN permet ainsi d'optimiser l'utilisation du cluster et de supporter plusieurs frameworks (MapReduce, Spark...).

# Apache Hadoop

## → MapReduce

**MapReduce** est un **modèle de programmation parallèle** inventé par Google, repris dans Hadoop. Il permet de traiter des grandes masses de données en deux étapes :

- **Map** : diviser le travail en petites tâches → chaque nœud traite sa part de données.
- **Reduce** : regrouper les résultats partiels et produire un résultat global.

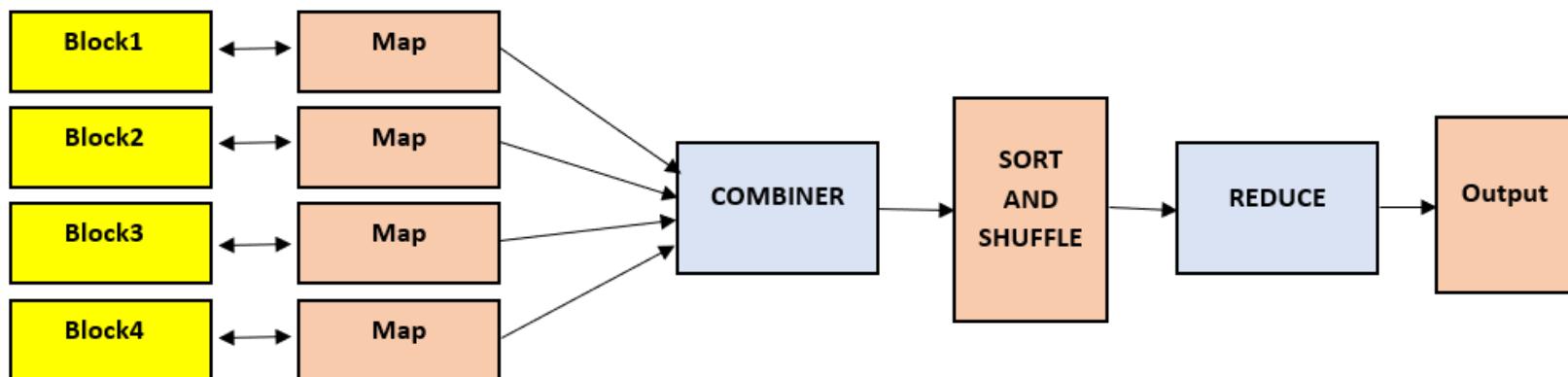


# Apache Hadoop

→ MapReduce

## □ Étapes d'un job MapReduce

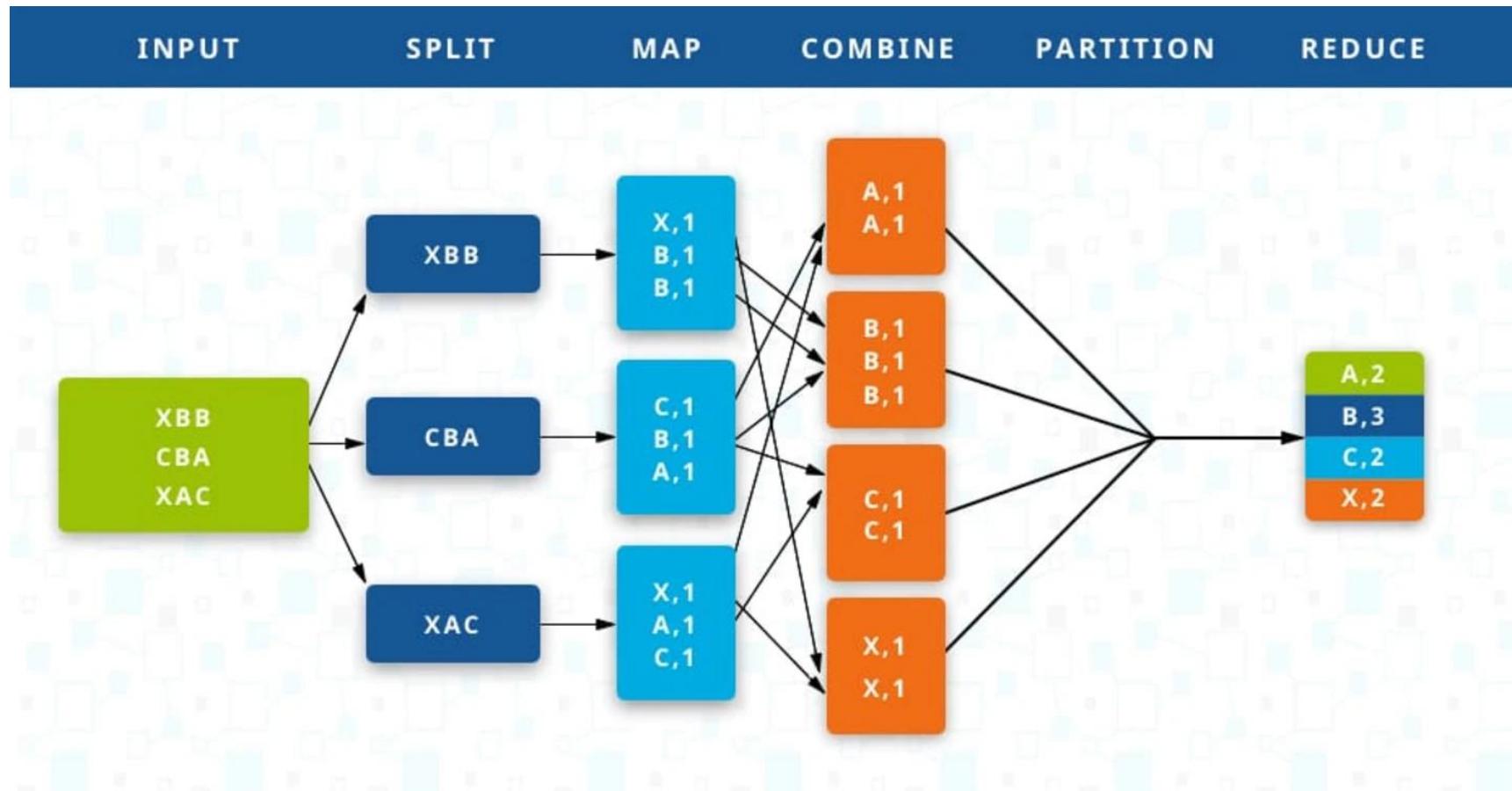
1. **Input** : lecture des données depuis HDFS.
2. **Map** : transformation en paires clé/valeur.
3. **Shuffle & Sort** : regroupement par clé.
4. **Reduce** : agrégation finale.
5. **Output** : écriture du résultat dans HDFS.



# Apache Hadoop

→ MapReduce

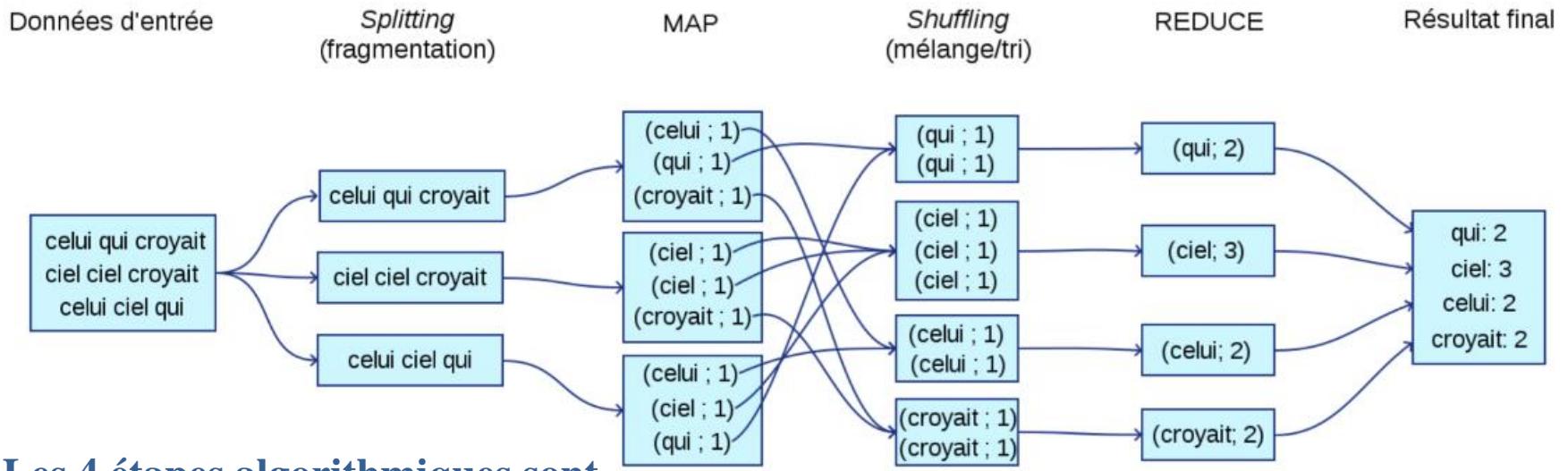
## Exemple 01



# Apache Hadoop

## → MapReduce

### Exemple 02



### □ Les 4 étapes algorithmiques sont

1. **Split** : est une fonction qui divise l'ensemble des données en blocs de données,
2. **Map** : est une fonction classique en programmation fonctionnelle qui applique une fonction sur chaque bloc de données. Le résultat de la fonction appliquée à chaque bloc est un ensemble de couples (clé, valeur),
3. **Shuffle** : est l'opération qui consiste à regrouper les couples (clé ; valeur) qui ont des clés identiques,
4. **Reduce**: est la fonction classique de programmation fonctionnelle (aussi appelé fold) qui réduit un ensemble de données en une donnée finale.

# Apache Hadoop

→ MapReduce

## Exemple

Compter le nombre d'occurrences de chaque mot dans un fichier texte.

### 1. Map

- *Input* : “big data hadoop big”
- *Output* : (big,1), (data,1), (hadoop,1), (big,1)

### 2. Shuffle & Sort

- (big,1), (big,1), (data,1), (hadoop,1)

### 3. Reduce

- big → 2, data → 1, hadoop → 1

### Output

- (big,2), (data,1), (hadoop,1)

# Apache Hadoop

→ MapReduce

## □ Les avantages et les limites

### • Les avantages

- Très simple conceptuellement.
- Scalabilité, tolérance aux pannes, simplicité conceptuelle.
- Permet un traitement **batch massif** (pétas de données).

### • Les limites

- Lent car écrit/lecture sur disque à chaque étape.
- Peu flexible pour certains traitements (analyse temps réel, interactif).

→ C'est pourquoi **Spark** a été introduit comme alternative plus rapide.

# Apache Hadoop

- **Hadoop** offre une infrastructure puissante et évolutive pour stocker et traiter de très grandes quantités de données.
- Son architecture basée sur **HDFS**, **YARN** et **MapReduce** garantit performance, résilience et flexibilité.

## *Dans la pratique*

- HDFS est responsable du stockage distribué.
  - YARN gère les ressources et la planification des tâches.
  - MapReduce effectue le traitement parallèle des données.
- Aujourd’hui, même si d’autres technologies comme **Apache Spark** sont souvent préférées pour les traitements modernes, Hadoop reste une base solide et incontournable dans tout écosystème Big Data.

# Apache Hadoop

## → Fonctionnement général de Hadoop

**Hadoop** est une plateforme distribuée qui permet de stocker et traiter de très grandes quantités de données sur un ensemble de machines (un cluster). Il repose sur trois composants complémentaires qui collaborent étroitement :

**1. HDFS (Hadoop Distributed File System)** → C'est le système de stockage.

Il divise les fichiers en blocs et les distribue sur plusieurs machines, avec réPLICATION pour éviter la perte de données en cas de panne.

**2. YARN (Yet Another Resource Negotiator)** → C'est le gestionnaire de ressources.

Il attribue les ressources (processeur, mémoire) aux différentes tâches et supervise leur exécution sur le cluster.

**3. MapReduce** → C'est le **moteur de traitement**.

Il exécute les programmes de traitement de données en deux étapes :

- Map** : diviser les données et les traiter en parallèle.
- Reduce** : agréger les résultats partiels pour produire le résultat final.

# Apache Hadoop

## → Fonctionnement général de Hadoop

### □ Collaboration des trois composants

1. L'utilisateur soumet une **tâche MapReduce** à Hadoop.
2. **YARN** reçoit la demande et attribue les ressources nécessaires (machines, mémoire, CPU).
3. Les mappeurs et réducteurs exécutés par **YARN** lisent et écrivent les données dans **HDFS**.
4. **HDFS** assure la réplication et la disponibilité des données tout au long du processus.
5. Une fois le calcul terminé, le résultat est stocké à nouveau dans **HDFS**.

# Apache Hadoop

## → Fonctionnement général de Hadoop

### Exemple

Supposons qu'une entreprise veuille **analyser 1 To de logs web** pour compter le nombre de visites par page :

- **HDFS** découpe les fichiers de logs et les stocke sur plusieurs machines.
- **YARN** distribue les ressources et lance plusieurs tâches MapReduce.
- Les **tâches Map** lisent chaque bloc et comptent les visites par page localement.
- Les **tâches Reduce** regroupent les résultats et produisent le total global des visites.
- Le **résultat final** est sauvegardé dans HDFS sous forme d'un fichier agrégé.

# Apache Hadoop

## → Hadoop: Installation et environnement de travail Hadoop

### □ Choix de l'environnement d'installation

Hadoop peut être installé de plusieurs manières, selon les besoins et les ressources disponibles :

- **Mode pseudo-distribué (single-node local)**
  1. Installation sur une seule machine (Linux ou VM).
  2. Recommandé pour l'apprentissage.
- **Mode distribué (multi-nodes)**
  1. Cluster réel avec plusieurs machines.
  2. Plus réaliste mais complexe.
- **Docker / Conteneurs**
  1. Hadoop tourne dans des conteneurs Docker prêts à l'emploi.
  2. Facile à installer, rapide à lancer et réinitialiser.
  3. Très utilisé en formation et pour des environnements temporaires.

# Apache Hadoop

→ **Hadoop: Installation et environnement de travail Hadoop**

## □ Pré-requis logiciels

### 1. Système d'exploitation

- Linux est recommandé (Ubuntu ou CentOS).
- Windows est possible avec WSL (Windows Subsystem for Linux) ou via une VM.

### 2. Java

- Hadoop est écrit en Java.
- Il faut installer Java JDK 8 (pas JDK 11 ou 17 qui causent parfois des incompatibilités).

### 3. SSH : communication entre nœuds.

### 4. Hadoop binaire : à télécharger depuis [Apache Hadoop](#)

Ou **Docker** ([Lien](#)) pour exécuter Hadoop dans des conteneurs.

# Apache Hadoop

## Exercice 03

Un fichier de 800 Mo doit être stocké dans HDFS avec une taille de bloc de 128 Mo et un facteur de réPLICATION de 3.

## Exercice 04

Réalisez un schéma clair et annoté représentant le fonctionnement général d'HDFS.

Votre schéma doit montrer :

- Le NameNode et son rôle (gestion des métadonnées)
- Les DataNodes et leur rôle (stockage des blocs de données)
- La répartition des blocs d'un fichier entre plusieurs DataNodes
- Le principe de réPLICATION des blocs (ex. 3 copies)
- Les flèches de communication entre le client, le NameNode et les DataNodes

# Apache Hadoop

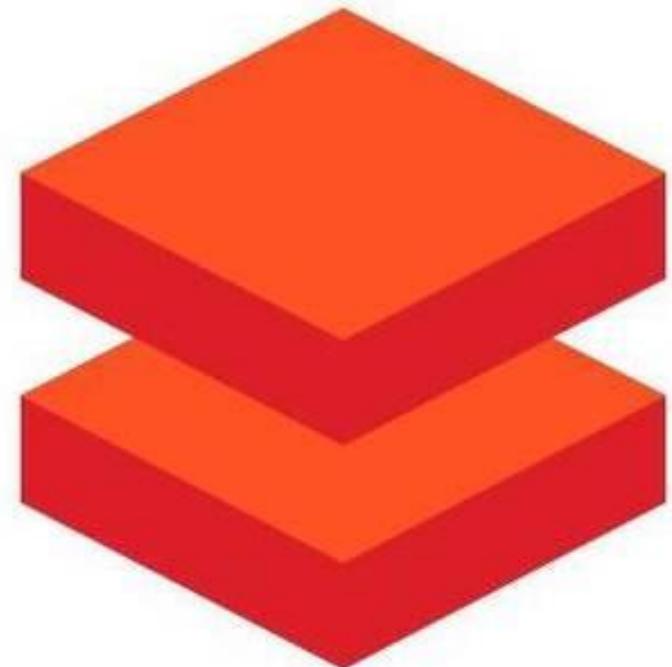
## Exercice 05

On dispose d'un cluster Hadoop composé de 20 nœuds (machines). Chaque machine possède 4 To de capacité de stockage. On veut stocker un jeu de données de 50 To dans ce cluster. La taille d'un bloc HDFS est de 128 Mo et le facteur de réPLICATION est 3.

1. Quelle est la capacité totale brute du cluster (sans réPLICATION) ?
2. Quelle est la capacité réelle utilisable si on prend en compte la réPLICATION par défaut (3 copies) ?
3. Combien de blocs HDFS seront nécessaires pour stocker les 50 To de données ?
4. Quel est l'espace total occupé dans le cluster (en To) à cause de la réPLICATION ?
5. Quelle fraction (%) de la capacité du cluster sera réellement utilisée par ces données ?

Si on augmentait le facteur de réPLICATION à 4, serait-il encore possible de stocker ces données dans le cluster ?

# Apache Spark



# Apache Spark

**Apache Spark** est un framework open source pour le traitement distribué de grandes quantités de données.

- Il a été développé en 2009 à l'Université de Berkeley (projet AMP Lab) avant d'être repris par la Fondation Apache.
- Spark a été conçu pour dépasser les limites de Hadoop MapReduce, notamment sa lenteur due aux écritures fréquentes sur disque.
- **Son objectif** : fournir une plateforme rapide, flexible et unifiée pour le traitement de données massives, en mémoire plutôt que sur disque.



# Apache Spark

## □ Pourquoi Spark ?

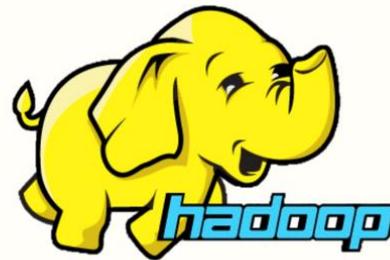
- Hadoop MapReduce oblige à écrire les résultats intermédiaires sur disque entre chaque phase, ce qui ralentit le traitement.
- Spark, au contraire, conserve les données **en mémoire (RAM)**, permettant un traitement jusqu'à **100 fois plus rapide** pour certaines charges de travail.

## Exemples d'utilisation

- Analyse de logs ou de flux de données.
- Traitement d'images et de vidéos.
- Machine Learning (classification, recommandation, clustering).
- Requêtes SQL sur des volumes de données massifs.
- Calculs en temps réel (streaming).

# Apache Spark

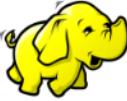
→ Comparaison avec Hadoop



Aspect	Hadoop MapReduce	Apache Spark
<b>Stockage intermédiaire</b>	Sur disque	En mémoire
<b>Langages supportés</b>	Java	Scala, Python, Java, R
<b>Rapidité</b>	Lente (E/S disque)	Très rapide
<b>Type de traitement</b>	Batch	Batch + Streaming + ML + SQL
<b>API haut niveau</b>	Non	Oui (DataFrame, SparkSQL, MLlib, GraphX)

# Apache Spark

## → Comparaison avec Hadoop

		
<b>Storage</b>	HDFS	<i>Leverage Existing</i>
<i>MapReduce</i>	✓	✓
<b>Speed</b>	<i>Fast</i>	<i>10 - 100 X Faster</i>
<b>Resource Management</b>	YARN	<i>Standalone</i>

Le mode **Standalone** est le mode natif de déploiement d'Apache Spark. Autrement dit, c'est le gestionnaire de cluster intégré à Spark lui-même, sans dépendre d'autres outils comme YARN (de Hadoop)

# Apache Spark

## → Comparaison avec Hadoop

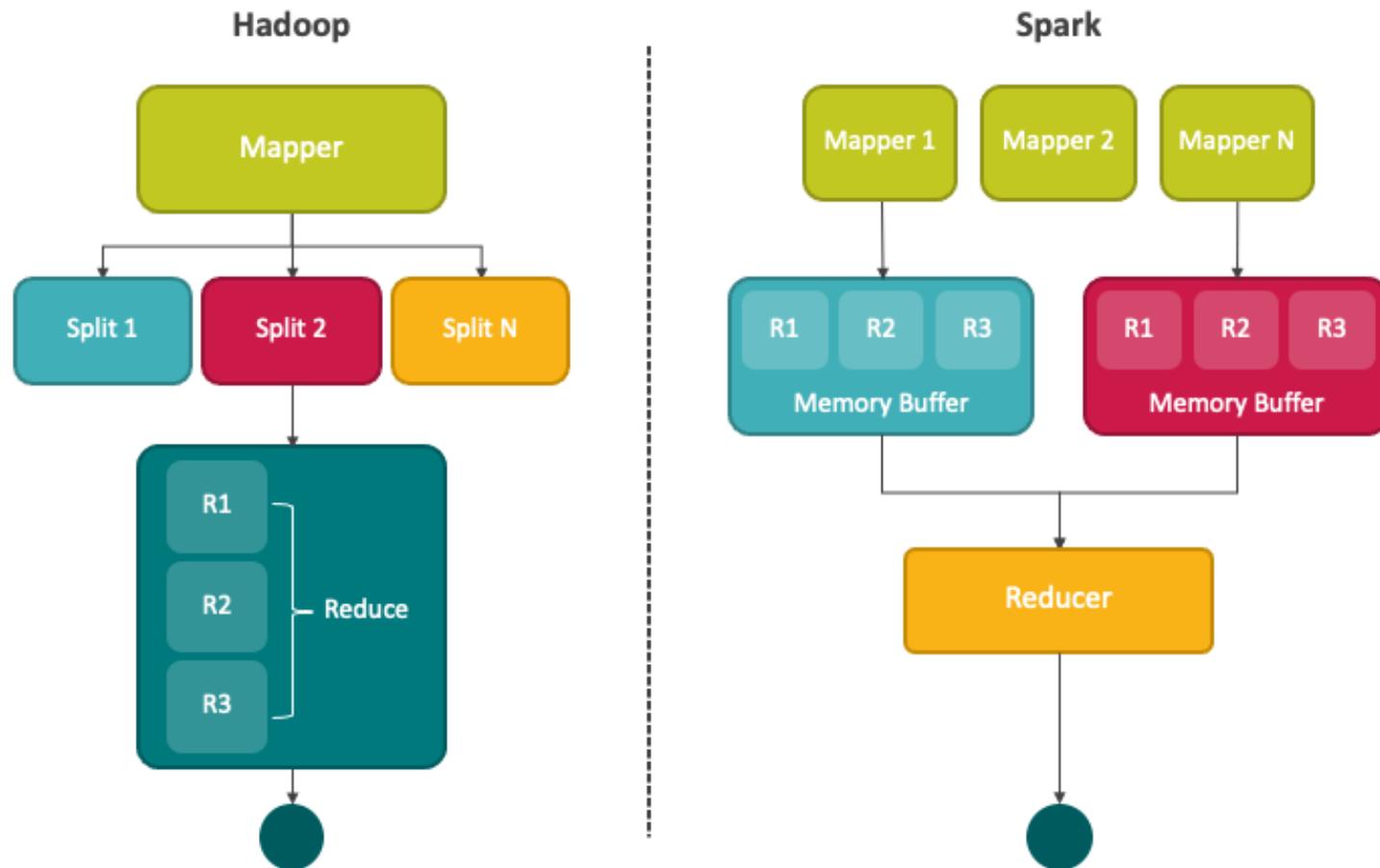
FACTORS	SPARK	HADOOP MAPREDUCE
Speed	100x times than MapReduce	Faster than traditional system
Written In	Scala	Java
Data Processing	Batch/real-time/iterative/interactive/graph	Batch processing
Ease of Use	Compact & easier than Hadoop	Compact & lengthy
Caching	Chaches the data in-memory & enhances the system performance	Doesn't support caching of data

**Scala** est un langage de programmation moderne, conçu pour combiner :

- la syntaxe concise et expressive de Python,
- avec la puissance et la robustesse de Java.

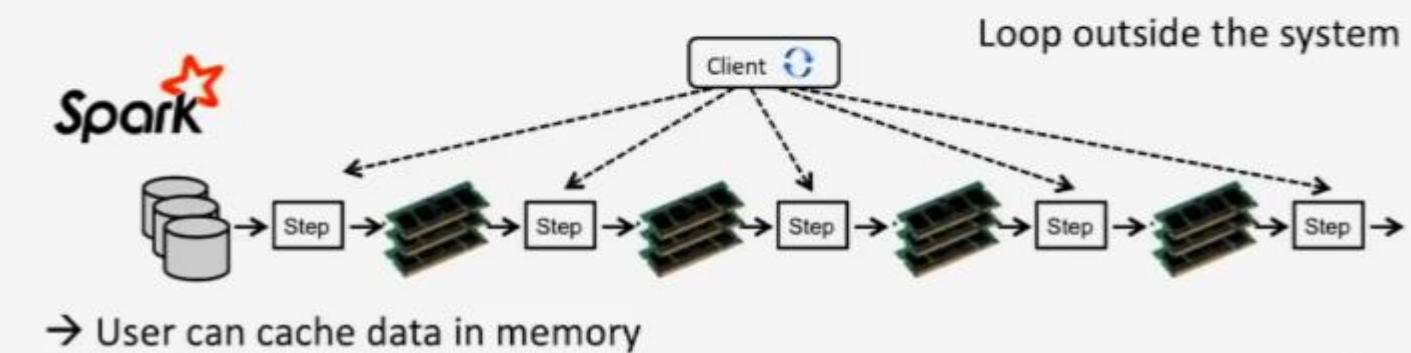
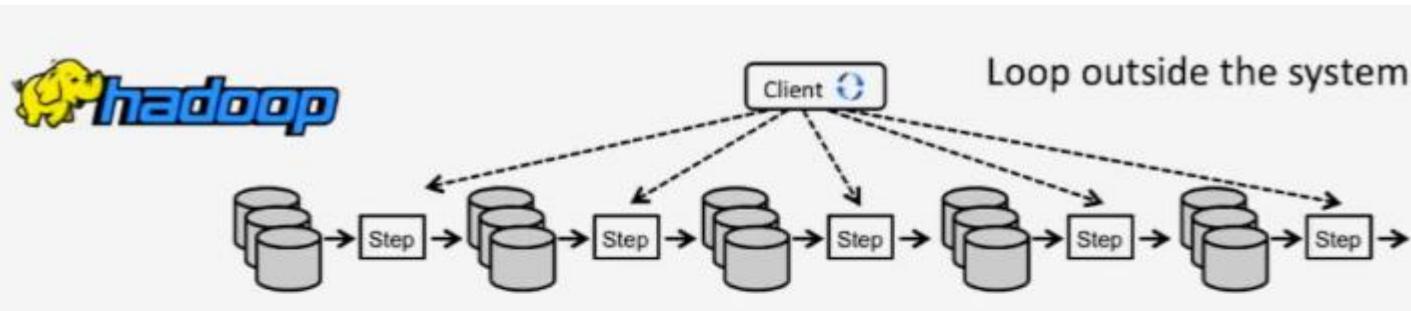
# Apache Spark

## → Comparaison avec Hadoop



# Apache Spark

## → Comparaison avec Hadoop



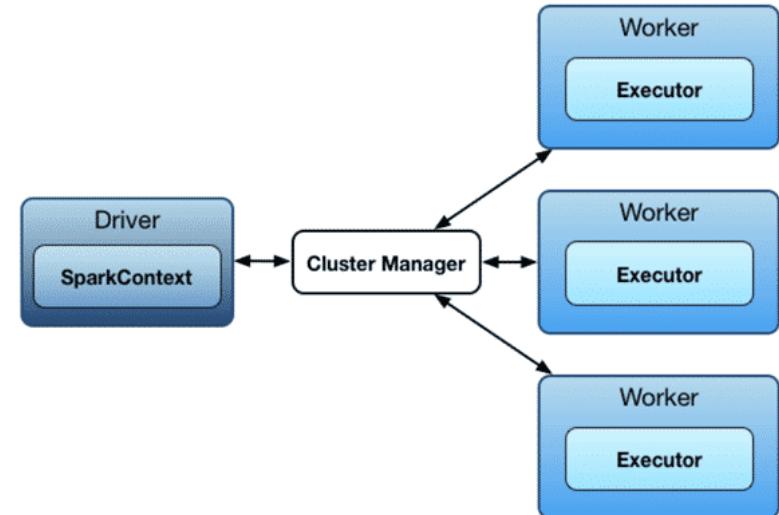
Spark est donc une **évolution naturelle** de Hadoop MapReduce : même philosophie (traitement distribué), mais bien plus rapide et flexible.

# Apache Spark

## → Architecture et composants de Spark

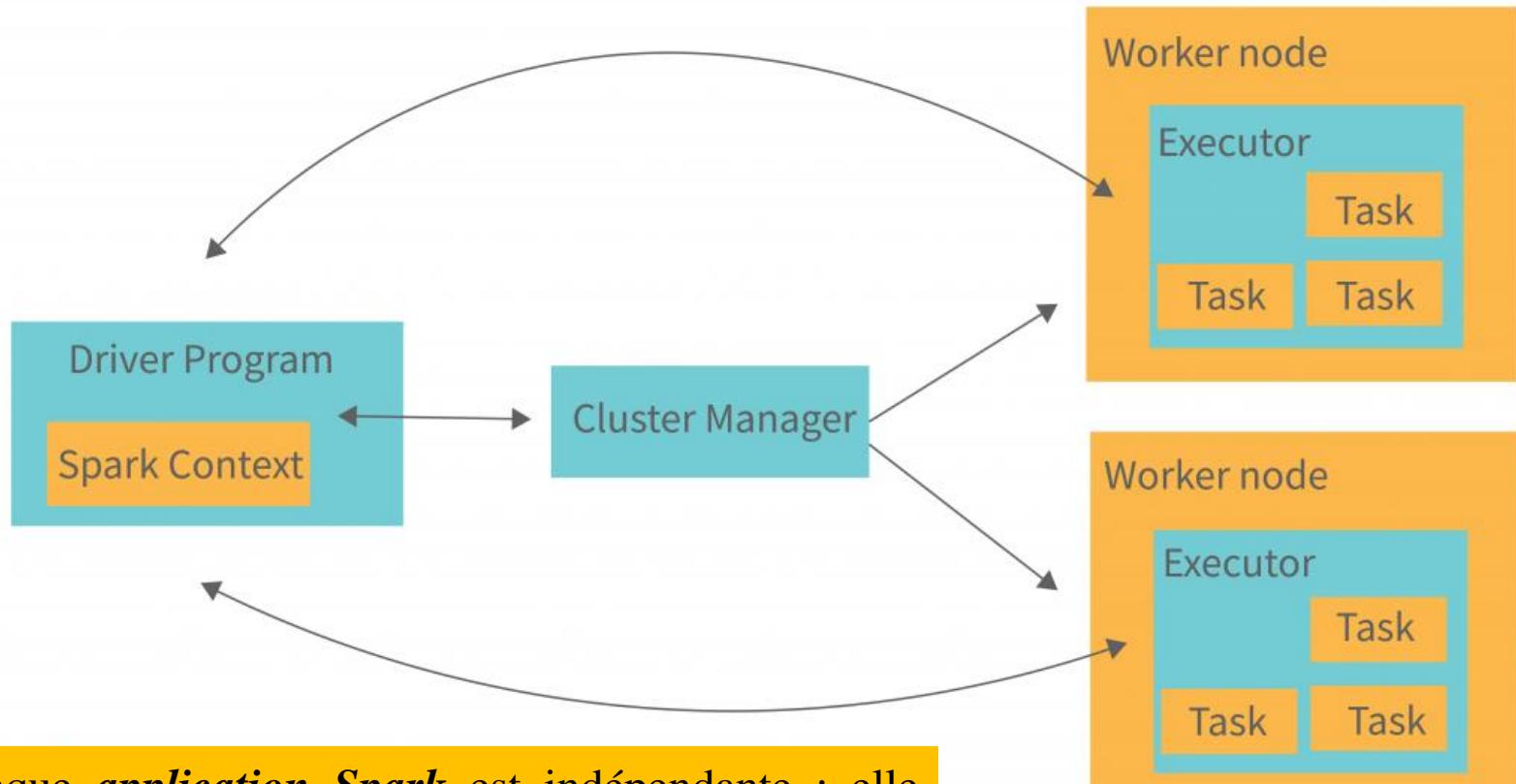
Spark suit une architecture **Master/Worker** :

- **Driver (Master)** : programme principal qui gère l'application Spark. Il crée un *SparkContext*, planifie les tâches et collecte les résultats.
- **Cluster Manager** : outil qui gère les ressources du cluster (CPU, mémoire). Il peut s'agir de YARN, Mesos ou du gestionnaire Spark propre (*Standalone*).
- **Executors (Workers)** : processus distribués sur différents nœuds qui exécutent les tâches planifiées et stockent les données en mémoire.



# Apache Spark

## → Architecture et composants de Spark



Chaque **application Spark** est indépendante : elle démarre son propre **Driver** et ses **Executors**.

# Apache Spark

## → Composants principaux de l'écosystème Spark

### 1. Spark Core

- Le moteur central qui gère la planification, la distribution et la tolérance aux pannes.
- Il introduit le concept fondamental des **RDD** (Resilient Distributed Dataset).

### 2. Spark SQL

Permet d'exécuter des requêtes SQL et de manipuler des **DataFrames** (structures tabulaires comme dans une base de données).

### 3. Spark Streaming

Gère les flux de données en temps réel (ex. : tweets, logs applicatifs).

### 4. MLlib

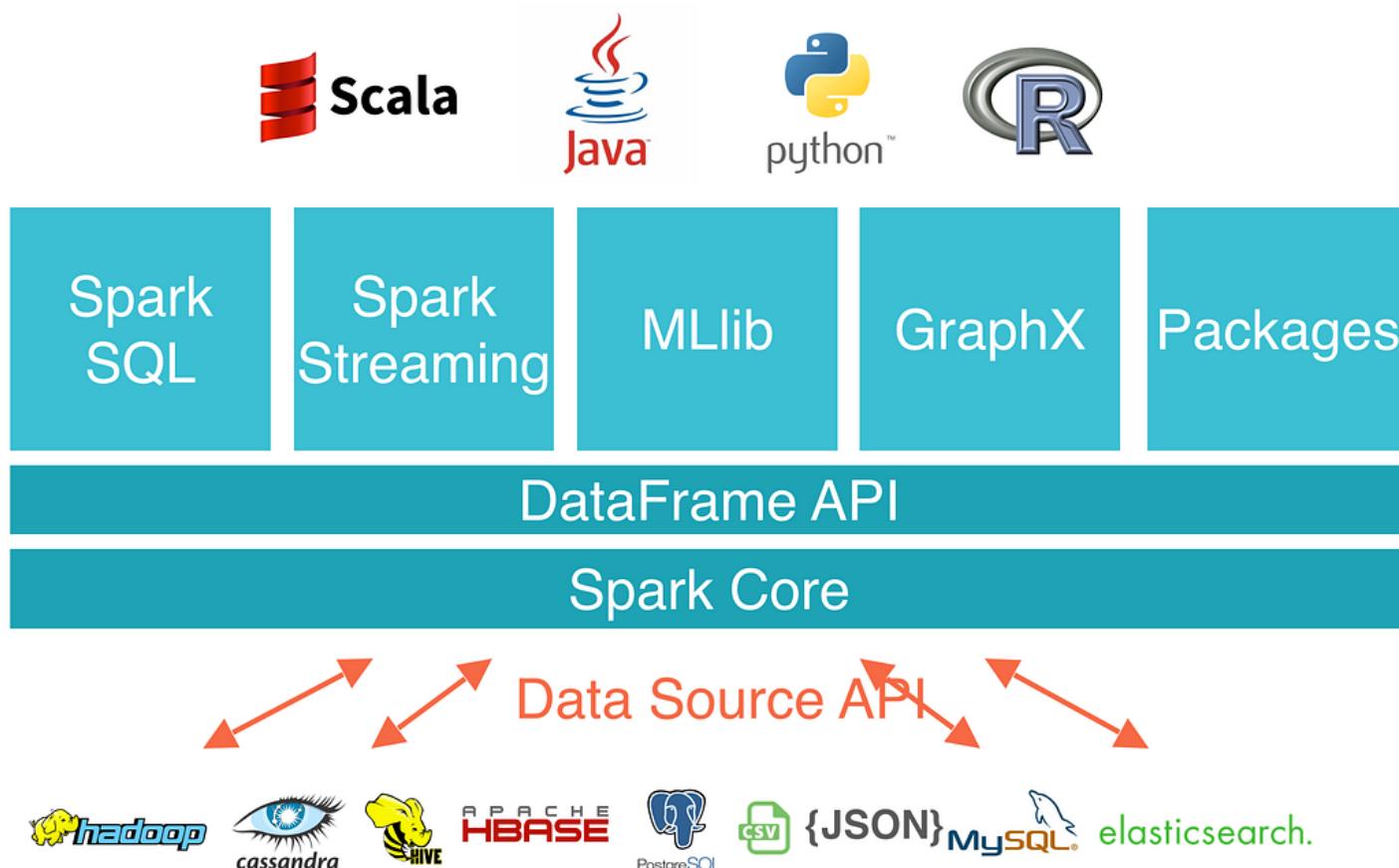
Bibliothèque de Machine Learning distribuée : régression, arbres de décision, clustering...

### 5. GraphX

Pour le calcul sur des graphes (ex. : analyse de réseaux sociaux).

# Apache Spark

→ Composants principaux de l'écosystème Spark



# Apache Spark

## → Les RDD: le cœur du modèle de données Spark

Un **RDD** (Resilient Distributed Dataset) est la structure de données de base de Spark.

- Il s'agit d'une **collection distribuée d'objets immuables**, partitionnée sur plusieurs nœuds du cluster.
- Chaque RDD peut être :
  - créé à partir d'un fichier dans HDFS ou un autre système de stockage
  - dérivé d'un autre RDD via des **transformations** (map, filter, join...).

### □ Propriétés des RDD

- **Résilience** : Spark garde la trace des transformations (via un *DAG*, Directed Acyclic Graph). Si une partition est perdue, elle peut être recomputée automatiquement.
- **Immutabilité** : les RDD ne changent pas ; chaque transformation produit un nouveau RDD.
- **Distribution** : les données sont réparties sur plusieurs nœuds pour un traitement parallèle.

# Apache Spark

## → Les RDD: le cœur du modèle de données Spark

### □ Création de RDD

#### Exemples en PySpark

```
from pyspark import SparkContext
sc = SparkContext("local", "ExempleRDD")

# Créer un RDD à partir d'une liste
rdd1 = sc.parallelize([1, 2, 3, 4, 5])

# Créer un RDD à partir d'un fichier
rdd2 = sc.textFile("hdfs:///user/etudiant/texte.txt")
```

**PySpark** est l'interface Python d'Apache Spark.  
Elle permet aux développeurs d'écrire du code Spark en langage Python.

# Apache Spark

## → Les DataFrames et Spark SQL

Un **DataFrame** est une abstraction de plus haut niveau que les RDD, introduite pour simplifier la manipulation de données structurées.

Il est similaire à une *table SQL* ou un DataFrame *Pandas*.

### □ Les caractéristiques

- Colonnes typées (nom, type, schéma).
- Optimisation automatique grâce au moteur Catalyst Optimizer.
- Compatible avec SQL : on peut écrire directement des requêtes sur les DataFrames.

### Exemple

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("DataFrameEx").getOrCreate()

# Charger un fichier CSV
df = spark.read.csv("hdfs://user/etudiant/ventes.csv", header=True, inferSchema=True)

# Afficher les premières lignes
df.show(5)

# Sélection et calcul
df.groupBy("region").sum("montant").show()
```

# Apache Spark

## → Les DataFrames et Spark SQL

### □ Les transformations courantes

Opération	Exemple
Sélection	<code>df.select("colonne")</code>
Filtrage	<code>df.filter(df["prix"] &gt; 100)</code>
Groupement	<code>df.groupBy("region").count()</code>
Tri	<code>df.orderBy(df["montant"].desc())</code>

# Apache Spark

## → Les transformations et actions principales dans Spark

Spark est fondé sur deux types d'opérations :

- **Transformations** : Crée un **nouveau RDD** ou **DataFrame** à partir d'un existant, sans calculer immédiatement le résultat.

- Ex. : **map()** , **filter()** , **flatMap()** , **join()** , **groupBy()** .

- **Actions** : opérations qui retournent un résultat au **Driver**.

- Ex. : **count()** , **collect()** , **saveAsTextFile()** .

### Exemple

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
rdd2 = rdd.map(lambda x: x * 2)      # Transformation
print(rdd2.collect())                 # Action
```

Les transformations sont *paresseuses* (lazy) : Spark ne les exécute qu'au moment où une action est demandée. Cela permet d'optimiser le plan d'exécution complet.

# Apache Spark

## → Les transformations et actions principales dans Spark

### ❑ Transformations Spark

Fonction / Méthode	Description / Utilité
<b><i>map (func)</i></b>	Applique une fonction à chaque élément et retourne un nouveau RDD/DataFrame
<b><i>filter (func)</i></b>	Garde uniquement les éléments qui respectent la condition
<b><i>flatMap (func)</i></b>	Applique une fonction à chaque élément et "aplatis" le résultat
<b><i>reduceByKey (func)</i></b>	Combine les valeurs par clé pour les RDD de paires (key, value)
<b><i>groupByKey ()</i></b>	Regroupe les valeurs par clé dans un RDD de paires
<b><i>join ()</i></b>	Fait la jointure de deux RDD/DataFrame basés sur une clé commune
<b><i>withColumn ()</i></b>	Crée ou remplace une colonne dans un DataFrame
<b><i>select ()</i></b>	Sélectionne certaines colonnes d'un DataFrame
<b><i>orderBy ()</i></b>	Trie les lignes d'un DataFrame selon une ou plusieurs colonnes

# Apache Spark

## → Les transformations et actions principales dans Spark

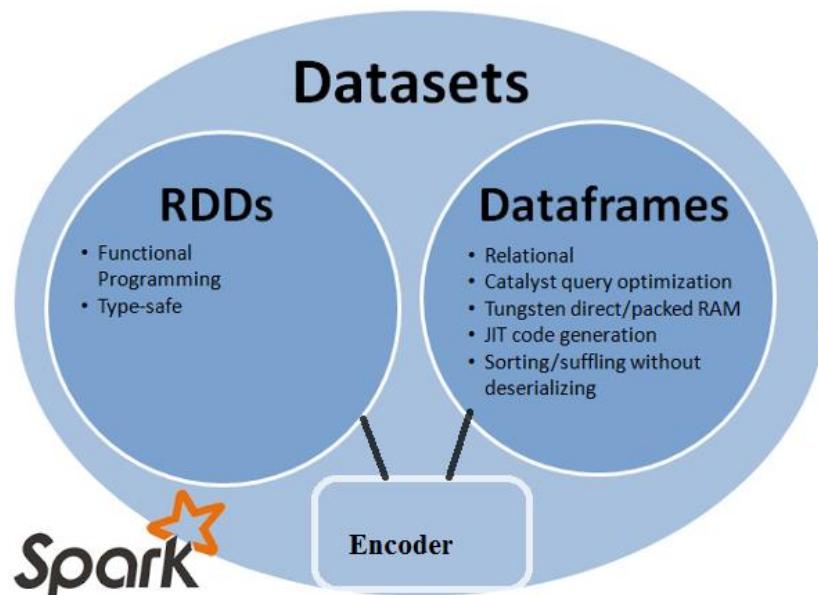
### □ Actions Spark

Fonction / Méthode	Description / Utilité
<b><code>collect()</code></b>	Récupère tous les éléments d'un RDD/DataFrame côté driver
<b><code>count()</code></b>	Compte le nombre d'éléments
<b><code>take(n)</code></b>	Récupère les n premiers éléments
<b><code>saveAsTextFile(path)</code></b>	Sauvegarde le RDD dans un fichier texte
<b><code>show(n)</code></b>	Affiche les n premières lignes d'un DataFrame
<b><code>reduce(func)</code></b>	Combine tous les éléments d'un RDD selon une fonction de réduction
<b><code>first()</code></b>	Récupère le premier élément d'un RDD/DataFrame

# Apache Spark

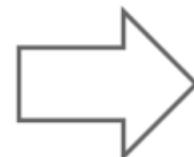
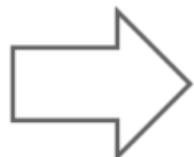
## → RDD / DataFrames

- **RDD** plus flexible mais nécessite plus de code et peut être plus complexe ; adapté aux données non structurées.
- **DataFrame** plus simple, structuré, performant, et adapté aux analyses et rapports sur de grandes données.



# Apache Spark

## → RDD / DataFrames



Distribute collection  
of JVM objects

Functional Operators (map,  
filter, etc.)

Distribute collection  
of Row objects

Expression-based operations  
and UDFs

Logical plans and optimizer

Fast/efficient internal  
representations

Internally rows, externally  
JVM objects

Almost the “Best of both  
worlds”: type safe + fast

But slower than DF  
Not as good for interactive  
analysis, especially Python

# Apache Spark

→ RDD / DataFrames

Critère	RDD	DataFrame
<b>Niveau d'API</b>	Bas niveau (low-level)	Haut niveau (high-level)
<b>Organisation</b>	Non structuré, éléments bruts	Structuré, colonnes avec types définis
<b>Support SQL</b>	Non	Oui, on peut faire select, groupBy, agg...
<b>Optimisation</b>	Moins optimisé	Optimisé automatiquement (Catalyst Optimizer)
<b>Facilité d'utilisation</b>	Plus compliqué, beaucoup de code	Plus simple, fonctions intégrées pour l'analyse
<b>Type de données</b>	Peut contenir tout type d'objet	Colonnes homogènes avec types précis
<b>Exemple d'opérations</b>	map(), filter(), reduceByKey(), flatMap()	withColumn(), groupBy(), agg(), orderBy(), show()
<b>Quand l'utiliser</b>	Données non structurées ou transformations complexes	Analyses, rapports, données tabulaires ou structurées

# Apache Spark

## → Exemple complet de traitement rapide

- Objectif : compter le nombre de ventes par région à partir d'un CSV sur HDFS.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("VentesRegion").getOrCreate()

# Charger les données
df = spark.read.csv("hdfs://user/etudiant/ventes.csv", header=True, inferSchema=True)

# Calcul du chiffre d'affaires par région
resultat = df.groupBy("region").sum("montant")

# Affichage et sauvegarde
resultat.show()
resultat.write.csv("hdfs://user/etudiant/resultats_ventes")
```

En interne, Spark lit les blocs HDFS en parallèle, stocke les partitions en mémoire, et exécute les transformations sans repasser par le disque.

# Apache Spark

## → Installation et environnement Spark

### □ Mode local

Télécharger Spark depuis [ce lien](#), décompresser, et lancer :

```
./bin/pyspark
```

L'interface interactive (*shell PySpark*) permet d'exécuter directement des commandes Python sur Spark.

### □ Docker

Utiliser une image prête :

```
docker run -it -p 8888:8888 -p 4040:4040 jupyter/pyspark-notebook
```

Accéder ensuite au notebook via <http://localhost:8888>

### □ Cluster (avec YARN)

Sur un cluster Hadoop existant, Spark peut être soumis comme une application :

```
spark-submit --master yarn script.py
```

# Apache Spark

## Exercice 06

Dans PySpark, crée un RDD contenant les nombres [1, 2, 3, 4, 5, 6, 7, 8, 9, 10].

1. Affiche tous les éléments.
2. Calcule la somme des éléments pairs.
3. Multiplie chaque élément par 2.
4. Affiche la moyenne des valeurs obtenues.

# Apache Spark

## Exercice 06

Étape	Fonction PySpark
<code>SparkSession.builder.appName(...).getOrCreate()</code>	Crée ou récupère une session Spark, nécessaire pour utiliser des DataFrames ou des RDD
<code>parallelize()</code>	crée un RDD à partir d'une liste Python
<code>collect()</code>	récupère tous les éléments du RDD
<code>filter()</code>	garde seulement les éléments qui respectent une condition
<code>sum()</code>	calcule la somme
<code>map()</code>	applique une fonction à chaque élément
<code>count()</code>	compte le nombre d'éléments

[Lien de la correction](#)

# Apache Spark

## Exercice 07

Soit un RDD contenant les mots d'un texte.

1. Découpe chaque ligne en mots.
2. Supprime les mots vides ("le", "la", "et", "de", ...).
3. Calcule le nombre d'occurrences de chaque mot.
4. Affiche les 10 mots les plus fréquents.

# Apache Spark

## Exercice 07 – Correction

Étape	Fonction
<code>flatMap()</code>	transforme chaque ligne en plusieurs mots
<code>filter()</code>	supprime les mots indésirables
<code>map()</code>	crée des paires (mot, 1)
<code>reduceByKey()</code>	additionne les valeurs associées à chaque mot
<code>sortBy()</code>	trie les résultats par fréquence
<code>take(10)</code>	affiche seulement les 10 premiers

[Lien de la correction](#)

# Apache Spark

## Exercice 08

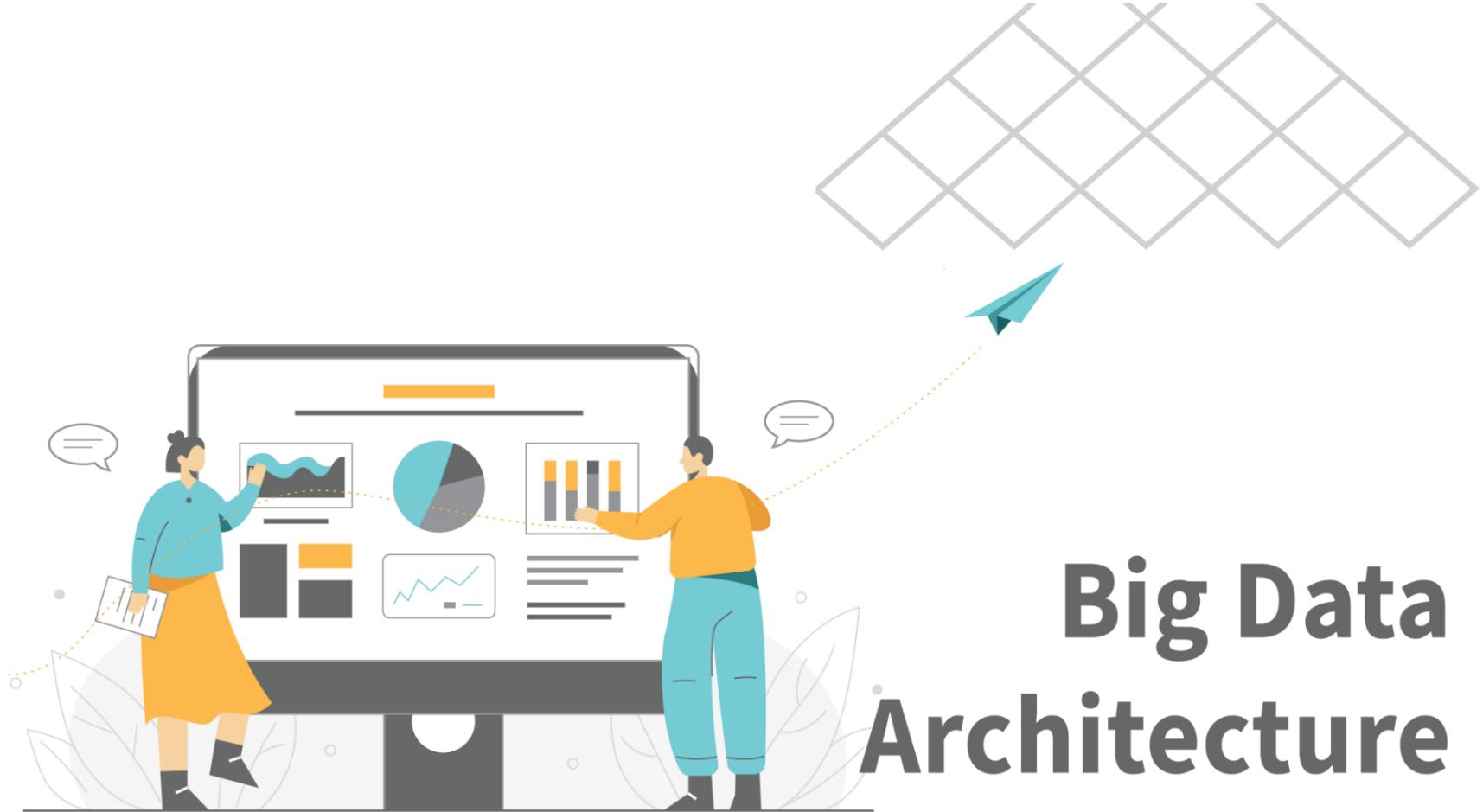
On dispose du fichier ventes.csv contenant :

*region,produit,quantite,prix\_unitaire*

1. Charge le fichier dans un DataFrame Spark.
2. Affiche les 5 premières lignes.
3. Crée une nouvelle colonne montant = quantite × prix\_unitaire.
4. Calcule le total des ventes par région.
5. Trie les résultats par montant décroissant.

[Lien de la correction](#)

# Notions de base sur l'architecture Big Data



# **Notions de base sur l'architecture Big Data**

Depuis une quinzaine d'années, les entreprises font face à une explosion du volume, de la diversité et de la vitesse des données.

Les informations proviennent de **sources multiples** :

- Transactions en ligne,
- Capteurs iot,
- Applications mobiles,
- Logs de serveurs,
- Réseaux sociaux, etc.

Les bases de données classiques (relationnelles) ne suffisent plus pour gérer **autant de données, aussi variées et changeantes**. Elles ont été conçues pour des données structurées, pas pour des flux massifs et hétérogènes.

→ **Problème** : comment collecter, stocker, transformer et analyser toutes ces données efficacement ?

C'est pour répondre à cette question qu'a émergé **l'architecture Big Data**, une organisation composée de plusieurs couches et technologies complémentaires.

# Notions de base sur l'architecture Big Data

## → Une architecture à plusieurs niveaux

Une architecture Big Data repose généralement sur quatre grandes briques :

- **Le Data Lake:** espace où l'on stocke toutes les données, dans leur forme brute.
- **Les processus ETL/ELT:** mécanismes qui extraient, transforment et chargent les données.
- **Le Data Warehouse:** entrepôt où les données sont structurées et prêtes à être analysées.
- **Les moteurs de traitement** (comme Hadoop et Spark): qui assurent le stockage distribué et le calcul parallèle.

On peut donc voir l'écosystème Big Data comme une chaîne :

*Sources → Stockage (Data Lake / Hadoop) → Traitement (ETL / Spark) → Analyse (Data Warehouse).*

# Notions de base sur l'architecture Big Data

→ Une architecture à plusieurs niveaux

Élément	Type	Rôle principal	Relation
<b>Hadoop</b>	Outil	Stocker et traiter de gros volumes de données	Base du Big Data
<b>Spark</b>	Outil	Traitement distribué rapide (batch & streaming)	Peut tourner sur Hadoop
<b>Data Lake</b>	Concept / Architecture	Stocker toutes les données (brutes et semi-structurées)	Souvent basé sur HDFS
<b>ETL / ELT</b>	Processus	Préparer et intégrer les données	Alimente le Data Lake ou le DW
<b>Data Warehouse</b>	Outil / Concept	Analyser les données structurées et propres	S'appuie sur le Data Lake ou ETL

# **Notions de base sur l'architecture Big Data**

→ **Le rôle de Hadoop et Spark dans cette architecture**

## **□ Hadoop : la fondation**

- Hadoop est la base technique sur laquelle repose une grande partie des architectures Big Data.
- Hadoop est la “colonne vertébrale” qui permet de stocker les données du Data Lake de manière fiable et évolutive.

## **□ Spark : le moteur de traitement rapide**

- Spark est un moteur moderne de traitement de données. Il fonctionne souvent au-dessus de Hadoop (en lisant depuis HDFS) et exécute les transformations nécessaires (nettoyage, calculs, agrégations, apprentissage machine).
- Spark prend les données du Data Lake, les nettoie, et alimente le Data Warehouse.
- Il joue un rôle central dans les processus ETL et ELT.

# Notions de base sur l'architecture Big Data

## → Le Data Lake: le réservoir de données brutes

Un **Data Lake** est un espace de stockage centralisé et flexible dans lequel on conserve toutes les données de l'entreprise, dans leur forme d'origine, qu'elles soient structurées ou non.

→ On l'appelle “**lac de données**” car il contient un grand volume de données hétérogènes, comme un lac recueille différents affluents.

### ❑ Caractéristiques principales

- Stockage massif : plusieurs téraoctets ou pétaoctets de données.
- Souplesse : aucune contrainte de schéma initial (“schema-on-read”).
- Coût réduit : stockage sur des systèmes distribués comme HDFS (Hadoop) ou sur le cloud (Amazon S3, Azure Blob).
- Centralisation : toutes les données sont accessibles depuis un même espace.
- traitements Spark.

# Notions de base sur l'architecture Big Data

## → Le Data Lake: le réservoir de données brutes

### □ Types de données stockées

- *Données structurées* : bases SQL exportées (clients, ventes).
- *Données semi-structurées* : JSON, XML, logs d'applications.
- *Données non structurées* : images, vidéos, sons, documents PDF.

### □ Lien avec Hadoop

- Le Data Lake s'appuie souvent sur HDFS, le système de fichiers distribué de Hadoop.
- Chaque fichier est découpé en blocs et réparti sur plusieurs serveurs, ce qui permet de stocker d'énormes volumes avec tolérance aux pannes.

### Exemple

Une entreprise enregistre toutes ses transactions, logs et capteurs dans un Data Lake basé sur HDFS. Ces données serviront plus tard pour les traitements Spark.

# Notions de base sur l'architecture Big Data

## → Le Data Warehouse: l'entrepôt de données structurées

Le **Data Warehouse** (entrepôt de données) est une base centralisée qui contient des données propres, validées et structurées, prêtes à être analysées. Il s'agit de la couche analytique du Big Data.

### □ Le Data Warehouse sert à:

- agréger les données importantes,
- accélérer les requêtes SQL, permettre la création de rapports et de tableaux de bord.

### Exemples de technologies

**On-premise : Hive, Vertica, Teradata**

**Cloud : BigQuery (Google), Snowflake, Redshift (AWS)**

# Notions de base sur l'architecture Big Data

## → Le Data Warehouse: l'entrepôt de données structurées

### □ Différence avec le Data Lake

Critère	Data Lake	Data Warehouse
<b>Données</b>	Brutes (non transformées)	Propres et organisées
<b>Schéma</b>	À la lecture	À l'écriture
<b>Format</b>	Multi-formats (texte, vidéo, JSON)	Tables structurées
<b>Objectif</b>	Stockage et exploration	Analyse et décision
<b>Technologie</b>	HDFS, S3	SQL, BigQuery, Hive

- Le Data Lake est la réserve d'eau brute,
- le Data Warehouse est l'eau filtrée prête à être consommée

- le **Data Lake** pour stocker tout
- le **Data Warehouse** pour extraire et analyser une partie “propre” et agrégée.

# Notions de base sur l'architecture Big Data

## → Le processus ETL : Extract, Transform, Load

Le **ETL** (*Extract, Transform, Load*) est un processus de préparation des données :

- **Extract** : extraire les données depuis leurs sources (fichiers, bases, API...).
- **Transform** : nettoyer, filtrer, enrichir les données.
- **Load** : charger les données propres dans un Data Warehouse.

### Exemple

Une société d'e-commerce :

- Extrait les ventes journalières depuis son Data Lake.
- Transforme les données (suppression des doublons, conversion des devises).
- Charge les résultats agrégés dans une table “ventes\_par\_région”.

# Notions de base sur l'architecture Big Data

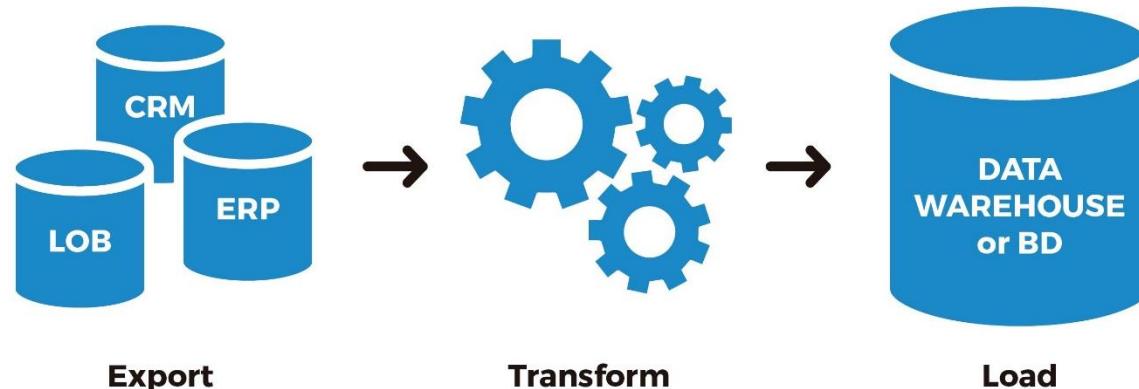
→ Le processus ETL: Extract, Transform, Load

## □ Outils utilisés

- Spark
- Talend
- Apache Airflow
- Informatica
- Pentaho

### Remarque

Avant Spark, on utilisait souvent Hadoop MapReduce pour ces traitements ETL, mais il était lent à cause des lectures/écritures fréquentes sur disque.



# Notions de base sur l'architecture Big Data

## → Le processus ELT: Extract, Load, Transform

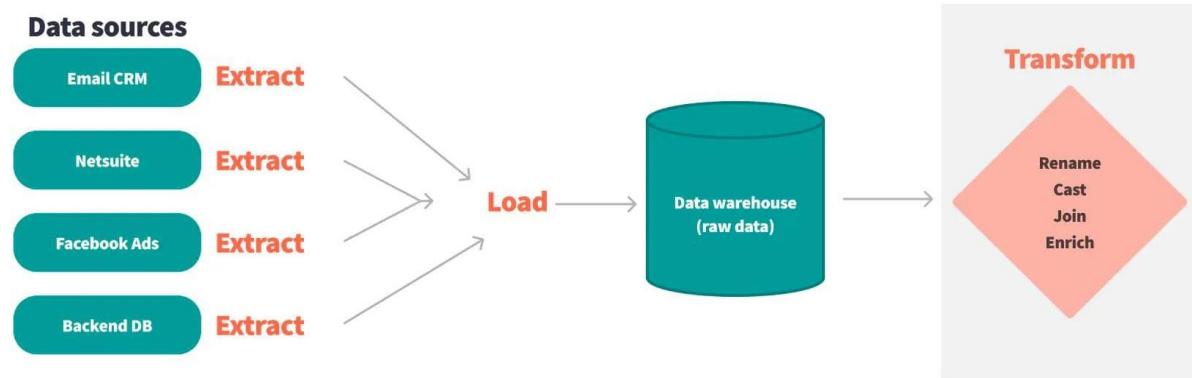
Dans l'approche **ELT**, on change l'ordre des opérations :

- **Extract** : les données sont extraites des sources.
- **Load** : elles sont directement chargées dans le Data Lake.
- **Transform** : les transformations sont faites ensuite, souvent avec Spark.

### □ Pourquoi ce changement ?

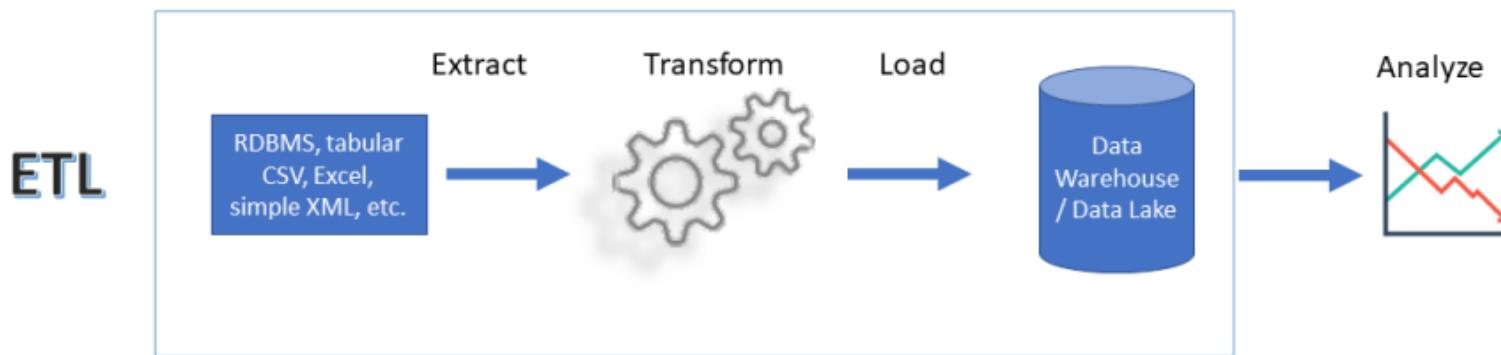
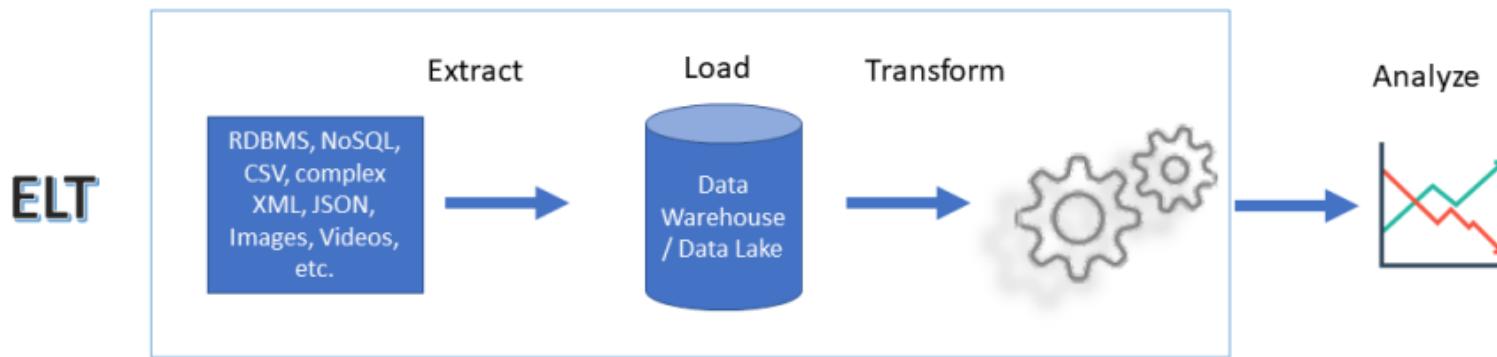
Avec le Big Data, il est plus logique de stocker d'abord les données brutes, puis de les transformer plus tard, selon les besoins des analystes.

→ Cette méthode est plus flexible, plus rapide, et mieux adaptée aux environnements distribués comme Hadoop/Spark.



# Notions de base sur l'architecture Big Data

→ ETL vs. ELT



# Notions de base sur l'architecture Big Data

→ ETL vs. ELT

## □ Comparaison entre ETL et ELT

Critère	ETL	ELT
<b>Ordre</b>	Extract → Transform → Load	Extract → Load → Transform
<b>Volume</b>	Données limitées	Données massives
<b>Lieu de traitement</b>	Serveur ETL dédié	Cluster Big Data
<b>Technologie</b>	Talend, Informatica	Spark, Databricks
<b>Utilisation</b>	Data Warehouse	Data Lake / Big Data

**ETL = ancien modèle (avant le Big Data)**

**ELT = modèle moderne adapté à Hadoop et Spark.**

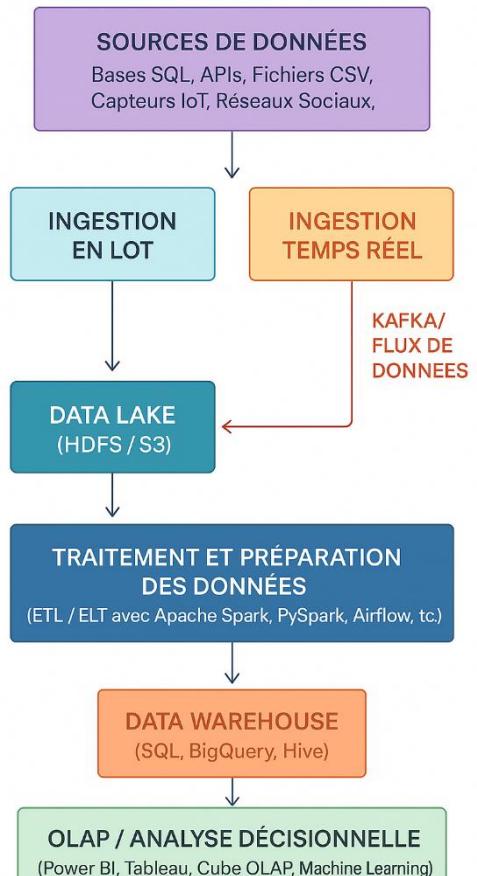
# Notions de base sur l'architecture Big Data

## → Synthèse : comment tout cela s'articule ?

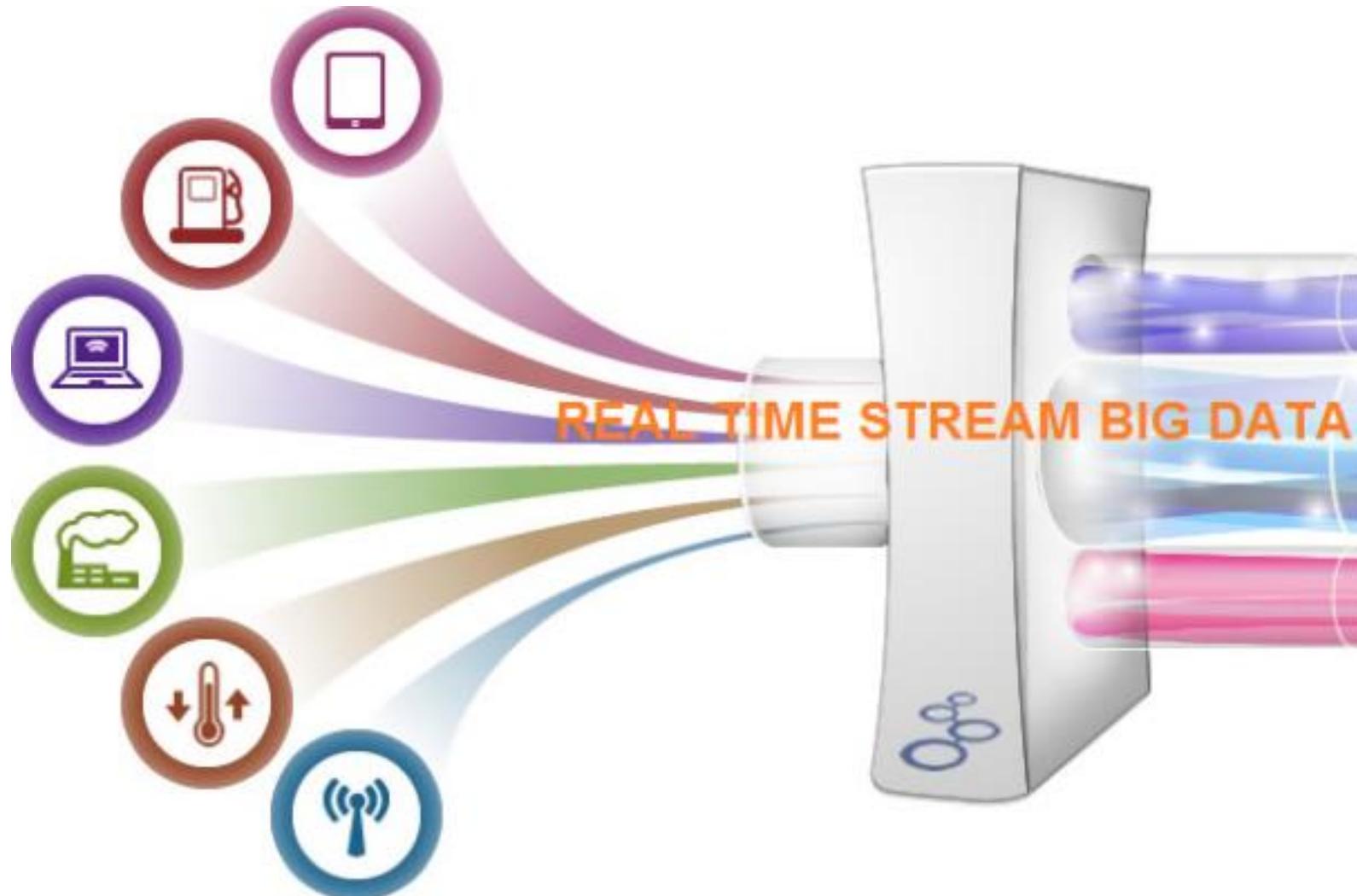
- Sources de données: les données proviennent de multiples origines.
- Ingestion :
  - En batch (fichiers, logs journaliers),
  - En temps réel via Kafka, qui transmet les flux continus.
- Data Lake (HDFS): stockage brut de toutes les données (Hadoop).
- Spark / ETL / ELT: traitement, nettoyage et transformation distribuée.
- Data Warehouse: base structurée et optimisée pour l'analyse SQL.
- OLAP / BI: analyse multidimensionnelle et visualisation.

→ **Apache Kafka** est une plateforme de streaming distribuée utilisée pour collecter, transmettre et traiter des flux de données en temps réel.

→ **OLAP** (Online Analytical Processing) est une technologie utilisée pour analyser les données sous plusieurs axes ou dimensions (temps, produit, région, client, etc.).



# Les bases du Streaming dans le Big Data



# Les bases du Streaming dans le Big Data

Traditionnellement, les traitements Big Data se faisaient en **batch** (par lots). Cela signifie que les données étaient **collectées, stockées, puis traitées périodiquement** par exemple toutes les heures, la nuit, ou en fin de journée. Mais dans de nombreux cas, les entreprises ont besoin d'informations **immédiates** :

- Surveillance de capteurs industriels,
- Détection de fraudes en ligne,
- Recommandations instantanées sur un site web,
- Suivi d'événements dans un réseau social.

→ Ces besoins ont donné naissance au **traitement de flux de données en temps réel**, appelé **stream processing** ou **data streaming**.

→ Deux technologies phares permettent de répondre à ce besoin dans l'écosystème Big Data :

- **Apache Kafka** pour la collecte et la transmission en flux continu
- **Apache Spark Streaming** pour le traitement et l'analyse en direct.

Ces deux outils sont souvent utilisés ensemble dans les architectures modernes de données.

# Les bases du Streaming dans le Big Data

→ Données batch vs données en streaming

Critère	Traitement Batch	Traitement Streaming
Type de données	Stockées (fichiers, HDFS)	En flux continu (événements, capteurs, clics)
Périodicité	Planifiée (ex : 1 fois par jour)	Continue, instantanée
Objectif	Analyse globale, rapports	Réactivité, alertes, supervision
Outils	Hadoop, Spark batch	Kafka, Spark Streaming, Flink
Exemple	Rapport des ventes journalières	Suivi des ventes en temps réel

- Le traitement **batch** s'exécute sur des données stockées.
- Le **streaming** s'exécute sur des données qui arrivent en continu.

# Les bases du Streaming dans le Big Data

→ Traitement des données en temps réel

## □ Apache Kafka

**Kafka** est un système de messagerie distribuée utilisé pour collecter et transporter les flux de données en temps réel.

- Il agit comme un intermédiaire entre les sources (sites web, capteurs, logs) et les outils de traitement (Spark, Flink...).
- Kafka stocke temporairement les messages dans des topics, accessibles par plusieurs consommateurs.



# Les bases du Streaming dans le Big Data

→ Traitement des données en temps réel

## ❑ Apache Spark Streaming

- Spark Streaming est un module de Spark dédié au traitement en temps réel.
- Il découpe les flux entrants en micro-lots de quelques secondes, puis les traite en parallèle comme des mini-jobs Spark.
- Les résultats sont ensuite envoyés vers un Data Lake, une base NoSQL ou un tableau de bord.

→ Il combine la puissance du batch et la rapidité du streaming.



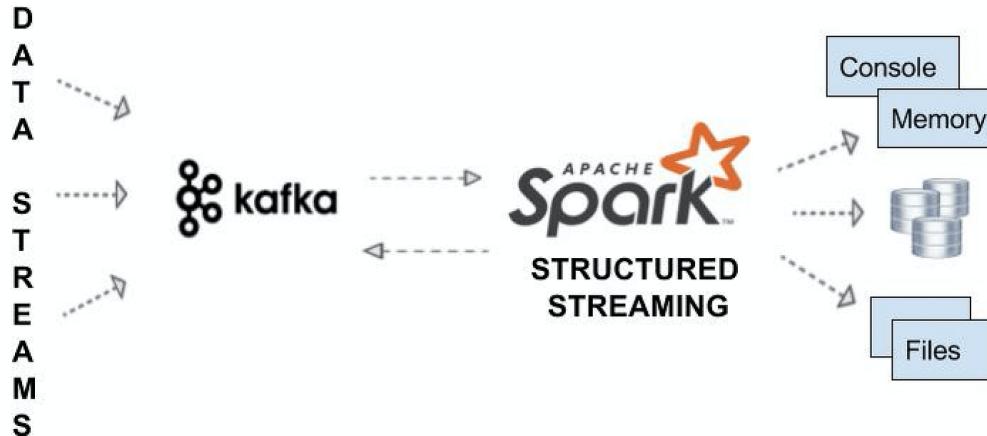
# Les bases du Streaming dans le Big Data

→ Traitement des données en temps réel

## □ Complémentarité Kafka – Spark Streaming

- Kafka reçoit et distribue les flux continus.
- Spark Streaming lit ces flux, les transforme et les agrège.  
→ Les résultats sont ensuite stockés dans HDFS ou visualisés en temps réel.

Exemple : Kafka collecte les clics d'un site web, Spark Streaming les analyse et affiche les produits les plus consultés.



# Les bases du Streaming dans le Big Data

## → Les avantages et les limites du streaming

### □ Les avantages

- Réactivité et suivi instantané
- Meilleure expérience utilisateur
- Détection rapide d'anomalies
- Complément au traitement batch

### □ Les limites

- Complexité de mise en place
- Gestion des volumes continus
- Nécessite des ressources stables
- Difficulté à rejouer les flux anciens

### □ Synthèse

Le **streaming** est devenu un pilier du Big Data moderne. Il permet de passer de l'analyse “*a posteriori*” à une **analyse en direct**.

- **Kafka** : collecte et transporte les flux continus.
- **Spark Streaming** : les traite en micro-lots rapides.
- **Hadoop / HDFS** : stocke les résultats dans le Data Lake.
- **Outils BI / OLAP** : affichent les résultats instantanés.

# Les bases du Streaming dans le Big Data

## → Cas d'usage typiques du streaming

- **E-commerce** : Recommandations produits en temps réel
- **Finance** : Détection de fraude instantanée
- **IoT / Industrie** : Surveillance d'équipements, alertes de panne
- **Réseaux sociaux** : Comptage en direct des mentions ou hashtags
- **Cybersécurité** : Détection d'intrusions ou d'anomalies réseau

