



Chapitre 1 : Introduction au Big Data

1. Qu'est-ce que le Big Data ?

Le **Big Data** désigne un ensemble de données :





- **Volumineuses** 📦
- **Produites en continu** ⌚
- **Hétérogènes** 🌐 (formats et sources variés)



Elles proviennent de :

- Individus (réseaux sociaux, smartphones)
- Entreprises
- Objets connectés (IoT)
- Systèmes informatiques

👉 Objectif : **collecter, stocker, traiter et analyser** ces données efficacement avec des technologies adaptées (Cloud, Hadoop, etc.).

2. Principales évolutions du Big Data

Évolution	Détail	Exemple
 Volume	Explosion de la quantité de données générées	Réseaux sociaux, IoT
 Origine	Sources variées (hors BDD classiques)	Logs, vidéos, capteurs
 Structure	Données structurées, semi-structurées, non-structurées	SQL, JSON/XML, vidéos
 Stockage	Adoption du Cloud Computing	AWS, Google Cloud

 Traitement	Nouvelles méthodes : batch & streaming	Hadoop, Kafka
 Usages	Nouvelles applications	Recommandations, alertes

Définitions utiles

- **Log** : Fichier d'événements système.
 - **XML** : Format de structuration de données (souple, hiérarchique).
 - **Cluster** : Ensemble de serveurs interconnectés.
 - **Datacenter** : Infrastructure physique hébergeant des clusters.
-

3. Les 5V du Big Data

V	Signification	Détail	Exemple
 Volume	Quantité énorme de données	> 175 ZB en 2025	
 Vélocité	Vitesse de génération/analyse	Trading, Google requêtes	
 Variété	Diversité des formats/sources	JSON, vidéos, IoT	
 Valeur	Utilité économique ou stratégique	Décisions, prédictions	
 Véracité	Fiabilité et qualité des données	Nettoyage, validation	

 Un **Plan Big Data** repose sur Volume, Vélocité, Variété, et nécessite :

- Infrastructures adaptées (Cloud, Hadoop)
- Traitements en parallèle / temps réel
- Gouvernance des données

4. Débuts du Big Data

MapReduce (Google)

- **Modèle parallèle** : Map (traitement) → Reduce (agrégation)
- **Avantage** : traitement distribué
- **Limite** : traitement batch uniquement

Google File System (GFS)

- **Stockage distribué**
- **Architecture maître-esclave**
- **Réplication des données** pour tolérance aux pannes

♦ **Résumé** : MapReduce + GFS = fondements du Big Data distribué → Inspirant Hadoop

5. L'Écosystème Hadoop

Composant	Fonction	Détail
HDFS	Stockage	Fichiers massifs, réplication
MapReduce	Traitement batch	Programmation parallèle
YARN	Orchestration	Remplace MapReduce v1
Hive	Requêtes SQL-like	HiveQL
Pig	Script haut niveau	Pig Latin
HBase	Base NoSQL	Temps réel sur HDFS
Zookeeper	Coordination	Services distribués
Oozie	Planification	Workflows Big Data

👉 Hadoop = écosystème complet pour gérer des volumes massifs **de façon distribuée et efficace**

Conclusion

Le Big Data transforme la manière dont les données sont utilisées. Avec des outils comme **Hadoop, Spark, Kafka**, et des plateformes comme **le Cloud**, les entreprises peuvent :

- Anticiper, personnaliser, automatiser
- Créer de la **valeur métier**
- Gagner un **avantage concurrentiel**

MAGUEMOUN SAMY

📖 Chapitre 2 : CONCEPTS DE BASE DE L'ENVIRONNEMENT HADOOP

✅ 1. Architectures logicielles & fichiers distribués

📌 Types d'architectures :

- **1-Tier (Monolithique)** : Tout est centralisé → ❌ **Pas adaptée** au stockage distribué.
- **2-Tier (Client/Serveur)** : Base de données et interface séparées → ❌ **Limité** pour HDFS.
- **3-Tier (Multicouche)** : Interface, logique métier, et stockage séparés → ✅ **Adaptée** au système distribué comme Hadoop avec YARN.

📌 Peer-to-Peer (P2P) :

- Chaque nœud agit comme **client et serveur**.
- Utilisé dans des systèmes comme **BitTorrent**.
- ✅ Adapté pour le stockage distribué, mais ❌ **pas pour Hadoop**, qui repose sur un modèle **maître-esclave**.

📌 Fichiers distribués :

- Un **DFS (Distributed File System)** stocke des fichiers sur **plusieurs machines**.
- **HDFS** utilise une architecture **3-Tier** en **maître-esclave**.
- ❌ Le P2P pur manque de contrôle centralisé → inadapté aux entreprises.

📌 Scalabilité :

- **Verticale** 📈 : Ajouter des ressources à une seule machine → ❌ limité.
- **Horizontale** ➡️ : Ajouter plusieurs machines → ✅ **Recommandé avec Hadoop**.

✓ 2. L'écosystème Hadoop

Composants clés :

- **HDFS** : Stockage distribué.
 - **MapReduce** : Traitement batch en parallèle.
 - **YARN** : Gestion des ressources et planification des tâches.
-

✓ 3. Systèmes de fichiers

Un système de fichiers gère les données sur un disque. Exemples : **FAT16**, **FAT32**, **NTFS**, **Ext4**...

✓ 4. HDFS : Hadoop Distributed File System

🏗️ **Architecture :**

- **NameNode** (maître) : Gère les métadonnées, pas les données.
- **DataNodes** (esclaves) : Stockent les blocs de données, exécutent les requêtes.
- **Secondary NameNode** : Sauvegarde régulière des métadonnées (pas un remplaçant du NameNode).

📌 **Fonctionnement :**

- Un fichier est **découpé en blocs**, stockés sur les **DataNodes**.
- Le **NameNode** garde une **carte d'emplacement**.
- Lors de la lecture, le **client** contacte le NameNode pour localiser les blocs, puis les récupère auprès des DataNodes.

🔍 **Caractéristiques :**

- **Blocs** : 128 Mo par défaut (ajustable).

- **Réplication** : Chaque bloc copié sur 3 nœuds (valeur par défaut = 3).
- **Scalabilité horizontale** : Ajout facile de nouveaux DataNodes.
- **Haute disponibilité** : Si un DataNode échoue, une copie est accessible ailleurs.

Lecture :

- Le client interroge le **NameNode**, puis contacte les **DataNodes**.
- Métadonnées stockées dans **FSImage** + modifications dans **EditLog**.

Écriture :

- Le client envoie le fichier → découpé en blocs → réparti & répliqué.
- Le NameNode met à jour les métadonnées.

Image explicative :

- Les fichiers sont **fragmentés, répliqués et répartis** dans le cluster.
- Le système garantit **résilience et performance**.

5. Système de fichiers Hadoop FS

Hadoop FS permet d'interagir avec différents types de systèmes :

- **HDFS** : Stockage natif distribué.
- **Local FS** : Stockage local pour test.
- **Amazon S3 FS** : Intégration cloud.
- **HFTP FS** : Transfert via HTTP entre clusters.
- **S3A FS** : Version optimisée avec gestion de permissions et performance.

Image illustrée :

Les données sont stockées sur différents nœuds avec réplication.
Même en cas de panne, les fichiers restent accessibles grâce à la redondance.

🧠 Concepts clés :

- **Scalabilité** : Ajout de machines.
 - **Résilience** : Données disponibles même en cas de défaillance.
-

✅ 6. Les Daemons d'HDFS

⚙️ Composants :

- **NameNode** :
 - Gère les métadonnées.
 - Stocke dans `fsimage` et `edits`.
- **Secondary NameNode** :
 - Fusionne `fsimage` et `edits` périodiquement.
 - Permet la récupération en cas de panne.
- **DataNode** :
 - Stocke les blocs de données.
 - Effectue lecture/écriture.
 - Envoie un **heartbeat** toutes les 3 secondes au NameNode.

♦ Résumé :

- **NameNode** : cerveau du cluster.
 - **SNN** : backup.
 - **DataNode** : muscle du stockage.
-

✓ 7. Paradigme MapReduce avec WordCount

🧠 Objectif :

Modèle distribué pour **traiter en parallèle** d'immenses volumes de données.

1 Phase MAP :

- Chaque fragment de texte est transformé en paires (**mot**, 1).
 - Exemple : "A, B, C" → (A, 1) (B, 1) (C, 1)
-

2 Phase SHUFFLE & SORT :

- Les paires sont regroupées par clé (mot).
 - Exemple : A → [1, 1], B → [1, 1, 1]
-

3 Phase REDUCE :

- Les valeurs sont **agrégées** (additionnées).
 - Résultat : (A, 2), (B, 3), (C, 4)
-

✓ Avantages :

- **Traitement massif, rapide et parallélisé.**
 - Idéal pour : logs, analyse texte, indexation web.
-

✓ 8. Jobs (Daemons) de Hadoop MapReduce

♦ JobTracker :

- Gère l'exécution des jobs MapReduce.
- Reçoit les fichiers **.jar** et les chemins d'entrée/sortie HDFS.
- Communique avec le **NameNode** pour localiser les blocs.
- Unique dans le cluster (point central de contrôle).

♦ TaskTracker :

- Exécute les tâches **Map** et **Reduce** assignées.
- Accède aux blocs de données sur HDFS.
- Présent sur chaque nœud du cluster.
- Envoie des **heartbeats** au JobTracker pour signaler son état.

✚ **Résumé** : Le **JobTracker** orchestre les tâches, les **TaskTrackers** les exécutent.

✓ 9. Exécution d'un job MapReduce

1. Le client copie les données sur **HDFS**.
2. Il soumet un fichier **.jar** + fichiers d'entrée/sortie au **JobTracker**.
3. Le JobTracker consulte le NameNode pour la localisation des blocs.
4. Il assigne les tâches aux TaskTrackers **proches des données**.
5. Les TaskTrackers envoient des heartbeats au JobTracker.
6. Le JobTracker leur envoie les instructions Map/Reduce.
7. Le job est terminé quand toutes les tâches sont complétées.

✚ **Conclusion** : Le traitement est **distribué** et **parallèle**, en tenant compte de la **localisation des données**.

✓ 10. ETL & ELT

🔄 ETL (Extract, Transform, Load) :

- **Extract** : Depuis BDD, fichiers logs, APIs...
- **Transform** : Nettoyage, formatage.
- **Load** : Chargement dans un entrepôt de données.

🔄 ELT (Extract, Load, Transform) :

- Données brutes chargées dans Hadoop.
- Transformation après stockage (via Hive, Spark, etc.).

📌 Différence :

- ETL : transformation **avant** stockage.
- ELT : transformation **après**, plus adapté au Big Data.

✓ 11. Problèmes de MapReduce

1. **Surcharge du JobTracker** : Trop de responsabilités.
2. **Nombre limité de nœuds** : Capacité dépend du cluster.
3. **Slots figés** : Inefficacité dans la répartition des ressources.
4. **Single Point of Failure** : Panne du JobTracker = arrêt du traitement.

📌 Conclusion : Limitations qui ont motivé la création de **YARN**.

✓ 12. YARN – Gestionnaire de ressources Hadoop

🔄 Évolution Hadoop v1 → Hadoop v2

- v1 : JobTracker gère **tout** → surcharge.
- v2 : Introduction de **YARN** :
 - Séparation **exécution** ↔ **gestion des ressources**.
 - Prise en charge de **plusieurs moteurs** (Spark, Tez...).

📌 **Avantages** : meilleure **scalabilité**, **flexibilité**, pas de Single Point of Failure.

♦ Fonctionnement de YARN

1. Le **client** soumet un job au **Resource Manager**.
2. Le **Scheduler** lui assigne un **conteneur**.
3. L'**Application Master** est lancé dans ce conteneur.
4. Il demande les ressources nécessaires aux **Node Managers**.
5. Les tâches sont exécutées sur différents nœuds.
6. Le client récupère les résultats une fois le job terminé.

📌 **Résumé** : **YARN** alloue dynamiquement les ressources, améliore la gestion et la parallélisation.

🔄 YARN vs MapReduce

- **YARN** :
 - Sépare Application Master et gestionnaire de ressources.
 - Gère plusieurs jobs **en parallèle**.
- **MapReduce (v1)** :

- Un JobTracker centralisé.
- Moins flexible.

📌 Résumé : YARN améliore l'efficacité, la scalabilité et la tolérance aux pannes.

✓ 13. MapReduce en Java – WordCount

◆ Mapper (**WordCountMapper**) :

- Reçoit une ligne de texte.
- La divise en mots avec **StringTokenizer**.
- Émet des paires (**mot**, **1**).

Exécution :

```
collector.collect(word, one);
```

📌 Objectif : Transformer le texte en paires **clé-valeur**.

◆ Reducer (**WordCountReducer**) :

- Reçoit une clé (mot) et une liste de **IntWritable** (des 1).
- Fait la somme des valeurs pour chaque mot.
- Émet (**mot**, **total**).

Exécution :

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

📌 Objectif : Additionner les occurrences de chaque mot.

♦ Illustration par l'image :

- **MAP** : Texte \rightarrow (mot, 1)
- **SHUFFLE** : Groupement par mot.
- **REDUCE** : Somme des 1 \rightarrow total final

📌 Résultat final :

cléA \rightarrow 2

cléB \rightarrow 3

cléC \rightarrow 4

✅ 14. Interface Writable

- Permet la **sérialisation** (écriture) et **désérialisation** (lecture) des données.
- Essentielle pour l'échange de données entre nœuds dans Hadoop.
- Implémentée par :
 - Text
 - IntWritable
 - LongWritable

📌 Hadoop ne peut pas échanger d'objets Java standard sans Writable \rightarrow adaptation nécessaire.

🎯 Conclusion Générale du Chapitre 2

- Hadoop repose sur un **ensemble de composants distribués**.
- **YARN** surpasse les limites de MapReduce v1.
- HDFS, MapReduce, YARN, ETL/ELT forment une base solide pour les traitements **Big Data massifs, fiables et scalables**.

Chapitre 3 – Les bases de données NoSQL

1. Modèles de gestion des données

- **Modèle hiérarchique** : Structure en arbre, chaque nœud enfant a un seul parent. Ex : IBM IMS.
 - **Modèle réseau** : Plus flexible que le hiérarchique, les nœuds peuvent avoir plusieurs parents.
 - **Modèle relationnel** : Données organisées en **tables** (relations) avec colonnes (attributs) et lignes (tuples).
 - **Modèle orienté objet** : Les données sont des objets avec classes, héritage et encapsulation.
 - **Modèle XML** : Représente des données semi-structurées sous forme de balises hiérarchiques XML.
 - **Modèle orienté graphe** : Stocke des **nœuds** (entités) et des **arêtes** (relations) pour modéliser les connexions.
 - **Modèle orienté document** : Stocke les données sous forme de documents structurés (JSON, BSON, XML).
-

2. Exemple du modèle relationnel

- Données représentées sous forme de **relations (tables)**.
 - Une **relation** contient des **tuples** (lignes) et des **attributs** (colonnes).
 - Les relations sont manipulées via le langage **SQL**.
-

3. SGBD relationnels et transactions

- Les **transactions** doivent satisfaire les propriétés **ACID** :
 - **Atomicité** : Toutes les opérations réussissent ou aucune.
 - **Cohérence** : L'état de la BD reste valide après une transaction.

- **Isolation** : Les transactions sont indépendantes entre elles.
 - **Durabilité** : Les données sont conservées même après une panne.
-

✓ 4. Caractéristiques et limites des SGBDR

- **Avantages** :
 - Bonne intégrité des données
 - Standardisation (SQL)
 - Support de transactions complexes
 - **Limites** :
 - Difficulté de passage à l'échelle horizontale
 - Moins performant pour des données non structurées ou massives
-

✓ 5. Pourquoi NoSQL ?

- Pour répondre aux besoins du **Big Data**, des données **non structurées**, et des systèmes **hautement distribués**.
 - Besoin de **flexibilité**, **scalabilité horizontale**, **haute disponibilité**, et **performance**.
-

✓ 6. Caractéristiques des bases NoSQL

- **Non relationnelles**, schéma souple
 - Haute **scalabilité** (surtout horizontale)
 - Bonne **performance en lecture/écriture**
 - **Moins strictes** en cohérence (souvent eventual consistency)
-

✓ 7. Propriétés des systèmes distribués : Théorème CAP

- **C** (Consistency - Cohérence) : Tous les nœuds voient les mêmes données en même temps.
- **A** (Availability - Disponibilité) : Chaque requête reçoit une réponse, même si certains nœuds sont en panne.
- **P** (Partition tolerance - Tolérance aux partitions) : Le système continue de fonctionner même en cas de coupure réseau.

▲ **CAP dit qu'on ne peut garantir que 2 sur les 3 en même temps.**

Exemples :

- **CP** : MongoDB (cohérence + tolérance aux partitions)
- **AP** : Cassandra (disponibilité + tolérance aux partitions)

✓ 8. Modèles de gestion des données NoSQL

♦ Clé-Valeur

Définition : Stocke les données sous forme de paires **clé** → **valeur**, chaque clé étant unique.

- **Exemples** : Redis, Riak
- **Avantages** : Ultra rapide, simple, hautement scalable
- **Inconvénients** : Pas de requêtes complexes, faible expressivité

♦ Colonnes (Wide Column Store)

Définition : Stocke les données en lignes mais organisées par **familles de colonnes**, optimisé pour les lectures par colonne.

- **Exemples** : HBase, Cassandra
- **Avantages** : Très efficace pour les analyses massives, bien adapté au Big Data
- **Inconvénients** : Moins intuitif, nécessite une bonne planification du schéma

♦ Documents

Définition : Stocke les données sous forme de **documents** structurés (souvent JSON ou BSON), chaque document pouvant avoir un schéma différent.

- **Exemples** : MongoDB, CouchDB
 - **Avantages** : Flexible, requêtes riches, supporte des données imbriquées
 - **Inconvénients** : Moins performant pour les jointures complexes
-

♦ Graphes

Définition : Stocke des données en **nœuds** (entités) reliés par des **arêtes** (relations), idéal pour modéliser les réseaux.

- **Exemples** : Neo4j, OrientDB
 - **Avantages** : Excellentes performances pour les relations complexes
 - **Inconvénients** : Moins performant pour les cas non relationnels
-

✓ 9. Cas pratique : HBase (base orientée colonnes)

HBase est un SGBD NoSQL distribué basé sur le modèle en colonnes, construit sur **Hadoop** et **HDFS**, inspiré de **BigTable** de Google.

✓ 10. SGBDR vs HBase

Critère	SGBDR	HBase
Schéma	Fixe	Flexible
Cohérence	Forte (ACID)	Eventuelle (BASE)

Langage	SQL	API Java / Shell
Stockage	Centralisé	Distribué via HDFS
Scalabilité	Verticale	Horizontale

✓ 11. Modèle de données HBase

- **Table** : Contient des lignes (comme en SQL)
 - **Clé de ligne (row key)** : Identifiant unique de la ligne
 - **Famille de colonnes** : Groupe logique de colonnes
 - **Colonne (qualifier)** : Appartient à une famille de colonnes
 - **Cellule** : Intersection d'une ligne et d'une colonne
 - **Version** : Chaque cellule peut avoir plusieurs versions (timestamp)
-

✓ 12. Architecture de HBase

- **HMaster** : Gère la distribution des régions et leur équilibrage
 - **RegionServer** : Gère les requêtes pour une ou plusieurs régions
 - **ZooKeeper** : Assure la coordination et la tolérance aux pannes
 - **Stockage** : Basé sur **HDFS**, pour la fiabilité et la distribution
-

✓ 13. Opérations HBase

■ **DDL (Data Definition Language)** :

- **create** : Créer une table
- **disable** / **enable** : Désactiver / Activer une table
- **alter** : Modifier une table
- **drop** : Supprimer une table

DML (Data Manipulation Language) :

- **put** : Ajouter ou modifier une valeur
- **get** : Lire une valeur
- **delete** : Supprimer une donnée
- **scan** : Lire plusieurs lignes

MAGUEMOUN SAMY

CHAPITRE 4 : Introduction au Cloud Computing

🧩 Partie 1 : Principes généraux du Cloud

1. 💡 Définition

Le **Cloud Computing** est une méthode de gestion des ressources informatiques via Internet. Les applications sont exécutées sur des serveurs distants interconnectés (datacenters), permettant une **scalabilité automatique** et une **gestion simplifiée** grâce à la **virtualisation**.

2. 🎯 Intérêts du Cloud

- **Rentabilité** : paiement à l'usage, pas d'infrastructure lourde.
 - **Agilité** : déploiement rapide des services.
 - **Haute disponibilité** : tolérance aux pannes intégrée.
 - **Sécurité** : assurée par les fournisseurs.
 - **Égalité d'accès** : même services pour tous, quel que soit l'emplacement.
-

3. ★ Caractéristiques essentielles

Caractéristique	Description
Libre-service à la demande	Activation automatique des ressources.
Accès réseau étendu	Accès via Internet avec des protocoles standards.
Mutualisation des ressources	Ressources partagées entre utilisateurs.
Élasticité & scalabilité	Adaptation rapide aux variations de la charge.

Facturation à l'usage	Mesure et transparence de la consommation.
-----------------------	--

4. Avantages & Considérations

- **Haute disponibilité** : nécessite une infrastructure adaptée et des applications bien conçues.
- **Scalabilité** :
 - *Scale-up* : puissance d'un serveur.
 - *Scale-out* : nombre de serveurs.
- **Élasticité** : adaptation automatique en temps réel à la demande.

5. Modèles de services Cloud

Modèle	Vous gérez	Fournisseur gère	Usage typique
IaaS (<i>Infrastructure</i>)	OS, applis, données	Hardware, virtualisation	Déploiement personnalisé
PaaS (<i>Plateforme</i>)	Applis, données	OS, outils, infra	Développement rapide
SaaS (<i>Logiciel</i>)	Rien	Tout	Services clés en main

6. Architectures de Cloud

Architecture	Sécurité	Flexibilité	Coût	Gestion	Cas typique
Public	Moyenne	Élevée	Faible à modéré	Externalisée	Startups, web apps
Privé	Élevée	Moyenne	Élevé	Interne	Banques, santé
Hybride	Élevée	Très élevée	Variable	Complexe	Grandes entreprises

Partie 2 : Technologique du cloud computing

I/ La virtualisation :

1. Définition de la Virtualisation

Technologie qui permet d'exécuter plusieurs **machines virtuelles (VMs)** isolées sur un même serveur physique.

2. Avantages de la Virtualisation

- Réduction des coûts matériels
- Meilleure utilisation des ressources
- Facilité de gestion et de déploiement
- Isolation des environnements
- Sauvegarde et restauration facilitées
- Haute disponibilité et reprise après sinistre

3. Domaines d'Application de la Virtualisation

- Centres de données

- Cloud computing
 - Environnements de test et développement
 - Formation et laboratoires virtuels
 - Consolidation de serveurs
-

4. Techniques de Virtualisation de Serveur

4.1. Notion d'Hyperviseur

- Définition
- Fonctionnement

4.2. Types d'Hyperviseurs

- Hyperviseur de type 1 (bare metal) : VMware ESXi, Microsoft Hyper-V, Xen
- Hyperviseur de type 2 (hosted) : VMware Workstation, VirtualBox

4.3. Types de Virtualisation

- Virtualisation complète
- Paravirtualisation
- Virtualisation au niveau du système d'exploitation (containers)

4.4. Organisation des Privilèges au sein des Processeurs x86

- Niveaux de privilège (Ring 0 à Ring 3)
 - Mode utilisateur vs mode noyau
 - Extensions de virtualisation matérielle (Intel VT-x, AMD-V)
-

5. Techniques de Migration de Machines Virtuelles

5.1. Types de Migration

- **Migration à chaud (live migration)** : sans interruption du service
- **Migration à froid** : VM éteinte ou arrêtée
- **Migration à chaud avec stockage partagé ou sans stockage partagé**
- **Migration planifiée vs d'urgence**

5.2. Objectifs de la Migration

- Maintenance
 - Équilibrage de charge
 - Optimisation des performances
-

6. Virtualisation du Stockage

6.1. Concepts de Base

- Stockage en **blocs** : accès bas niveau aux disques (ex : iSCSI, FC)
- Stockage en **fichiers** : accès via un système de fichiers partagé (ex : NFS, SMB)

6.2. Types de Stockages

- DAS (Direct Attached Storage)
- NAS (Network Attached Storage)
- SAN (Storage Area Network)

6.3. Stockage RAID

- RAID 0, 1, 5, 6, 10
 - Redondance, performance, tolérance aux pannes
-

7. Protocoles de Communication des Serveurs SAN

- **Fibre Channel (FC)** : haut débit, très utilisé en entreprise
 - **FCoE (Fibre Channel over Ethernet)** : convergence réseau/stockage
 - **iSCSI (Internet Small Computer System Interface)** : utilise TCP/IP pour transporter les commandes SCSI
 - **InfiniBand** : très haute performance, faible latence
 - **NVMe over Fabrics (NVMe-oF)** : nouvelle génération, très rapide pour SSD
-

II/ Introduction à la conteneurisation (Docker)

1. Conteneur

Un **conteneur** est une unité logicielle légère et portable qui **embarque une application avec toutes ses dépendances**, s'exécutant directement sur le noyau de l'OS hôte. Contrairement à une VM, il **ne contient pas de système d'exploitation complet**, ce qui le rend plus rapide et plus léger.

🔴 **Différences principales :**

Critère	Machine virtuelle (VM)	Conteneur (Docker)
OS	OS invité + OS hôte	Partage uniquement l'OS hôte
Isolation	Forte (niveau matériel)	Moyenne (niveau logiciel)
Ressources	Lourdes	Légères
Démarrage	Lent	Rapide

✅ **Exemple :**

- VM : Ubuntu tournant sur Windows avec VirtualBox.
 - Conteneur : Application Python isolée dans un conteneur Docker sous Linux.
-

2. Avantages des conteneurs

- **Légereté** : Pas d'OS complet embarqué → démarrage rapide, taille réduite.

- **Performance** : Utilisation directe du noyau hôte → faible surcharge.
 - **Portabilité** : Fonctionne de manière identique sur tous les environnements (dev, test, prod).
 - **Uniformité** : Environnement homogène, évite le "ça marche chez moi".
 - **Modularité** : Parfait pour les architectures microservices.
 - **Optimisation des ressources** : Meilleure densité que les VMs (plus de conteneurs sur un même serveur).
-

3. Solutions de conteneurisation

Évolution Linux :

- **OpenVZ** : Ancien système de conteneurisation basé sur des **containers système** (VPS).
- **LXC** : Conteneurs Linux plus proches de l'environnement natif.
- **Docker** : Conteneurisation **standardisée**, orientée application, s'appuie sur les **cgroups** et **namespaces**.

Aujourd'hui :

- Docker est la **référence**, intégré avec Kubernetes pour l'orchestration, CI/CD pour l'automatisation, et utilisé dans le cloud, l'IA, et les microservices.
 - Alternatives : **Podman**, **CRI-O**, **containerd**, etc.
-

4. La conteneurisation sous Docker

Docker : définition

Plateforme permettant de **créer, exécuter et gérer** des conteneurs applicatifs isolés, via un moteur (Docker Engine).

Architecture Docker :

- **Docker Engine** : Cœur du système, gère l'exécution, l'isolation, la sécurité et les ressources.
- **Docker Client** : Interface en ligne de commande (**docker**) pour interagir avec l'engine.
- **Image** : Modèle de conteneur, lecture seule, contient l'app, ses dépendances, etc.
- **Conteneur** : Instance d'exécution d'une image. Chaque conteneur est isolé.
- **Registry (registre)** : Répertoire d'images Docker (ex. : Docker Hub, GitHub Container Registry).

Cycle typique :

1. Création de l'image via un **Dockerfile**.
2. Stockage dans un **registre**.
3. Lancement d'un **conteneur** basé sur cette image, en local ou distant.

CHAPITRE 5 : Les traitements de données avec SPARK

♦ 1. Types de traitements Big Data

Traitement Batch (par lots)

- **Définition** : Traitement différé sur de grandes quantités de données pré-collectées.
- **Caractéristiques** :
 - Données **entièrement disponibles** avant traitement.
 - **Haute latence**, adapté aux analyses **historiques** ou **rapports périodiques**.

- **Concepts associés :**

- **Sérialisation / Désérialisation** : conversion et reconstruction des objets (ex : JSON).
- **Réplication** : duplication pour redondance et tolérance aux pannes.

Traitement Stream (en flux)

- **Définition** : Analyse des données **en temps réel**, dès leur arrivée.
- **Caractéristiques** :
 - Traitement **immédiat**, sans attendre la fin de la collecte.
 - Adapté à la **surveillance, détection de fraudes, IoT**, etc.

Comparaison Batch vs Stream

Critère	Batch	Stream
Volume	Très grand (stocké)	Continu (en direct)
Latence	Élevée	Faible
Complexité	Moins complexe	Plus technique
Cas d'usage typique	Statistiques, rapports	Temps réel, alertes

◆ **2. Framework Apache Spark**





Présentation générale

- Créé en **2009** par Matei Zaharia.
- Plus rapide que Hadoop grâce au **traitement en mémoire (RAM)**.
- Projet Apache depuis **2013**.

Composants principaux

- **Spark Core** : gestion mémoire, planification, tolérance aux pannes, accès aux données (HDFS, etc.).
- **RDD (Resilient Distributed Dataset)** : structure fondamentale de données distribuées.
- **Spark SQL** : requêtes SQL multi-langages (Scala, Python, R, Java).
- **Spark Streaming** : traitement de flux via **DStreams** (RDDs séquentiels).
- **GraphX** : analyse de graphes (réseaux sociaux, relations complexes).
- **MLlib** : machine learning intégré.

♦ 3. Caractéristiques d'Apache Spark

Caractéristique	Description
 Traitement en mémoire	Exécution rapide sans aller-retour disque.
 Exécution parallèle	Plusieurs tâches exécutées en simultané.
 Support multi-langages	Scala, Python, Java, R.
 Composants intégrés	SQL, Streaming, MLlib, GraphX.

♦ 4. Architecture générale Spark

[Driver]

- ↳ Initialise SparkContext
- ↳ Coordonne les tâches



[Master]

- ↳ Répartit les tâches vers les nœuds du cluster



[Executors (dans chaque nœud)]

- ↳ Exécutent les tâches et stockent les données intermédiaires

- **Driver** : pilote l'application Spark.

- **SparkContext** : point d'entrée pour interagir avec le cluster.
- **Master** : coordonne le traitement global.
- **Executors** : exécutent les tâches et renvoient les résultats.

♦ 5. Avantages d'Apache Spark

- ⚡ **Vitesse** : jusqu'à 100 fois plus rapide que Hadoop grâce à l'exécution en RAM.
- 🧠 **Analyse avancée** : prend en charge Machine Learning, Streaming, graphes.
- 🗣️ **Multi-langage** : Java, Scala, Python, R.

♦ 6. RDDs (Resilient Distributed Dataset)

📌 Propriétés

- **Immutable** : ne peuvent pas être modifiés après création.
- **Lazy evaluation** : transformations exécutées seulement lors d'une **action**.
- **Cacheable** : peuvent être mis en mémoire pour accélérer les traitements.

🔧 Distribution & Partitionnement

- Les RDDs sont **automatiquement partitionnés** selon les ressources du cluster (nœuds, HDFS...).

🔧 Transformations vs Actions

Type	Description	Exemples
🔄 Transformations	Créent de nouveaux RDDs (lazy)	<code>map</code> , <code>filter</code>
🎯 Actions	Déclenchent le calcul réel	<code>collect()</code> , <code>count()</code>



Création via SparkContext

```
rdd1 = sc.textFile('/chemin/fichier.txt')      # depuis un fichier  
rdd2 = sc.parallelize(['a', 'b', 'c'])         # depuis une collection  
rdd3 = rdd2.map(lambda x: (x, 1))              # transformation
```



Actions courantes

- `collect()` : retourne tous les éléments.
- `count()` : compte les éléments.
- `first()` : retourne le premier élément.
- `take(n)` : retourne les n premiers éléments.



Résumé général

Apache Spark est une plateforme Big Data **moderne, rapide, multi-usage** et **interactive**, conçue pour les **traitements distribués, batch ou stream**, avec un large éventail de composants intégrés. Il surpasse Hadoop MapReduce en efficacité grâce à son **traitement en mémoire** et son **architecture parallèle**.