TD1 - De Spark à Hadoop : stocker et traiter des données avec HDFS et YARN

- **Durée :** 4 heures
- **Travail :** binôme
- **Environnement :** Docker Compose, Hadoop, HDFS, YARN

1. Contexte

Une plateforme de commerce en ligne collecte quotidiennement :

- des commandes ;
- des événements de navigation ;
- des journaux applicatifs.

Les données sont actuellement stockées sur une seule machine. Cette architecture pose plusieurs problèmes :

- capacité de stockage limitée ;
- risque de perte en cas de panne ;
- traitements concurrents difficiles à gérer ;
- impossibilité de répartir efficacement les calculs.

Votre mission consiste à mettre en place une petite plateforme Hadoop permettant de :

```
1. stocker les données dans HDFS ;
2. observer leur distribution dans le cluster ;
3. vérifier la réplication et la tolérance aux pannes ;
4. comprendre le rôle de YARN ;
5. exécuter un traitement distribué.
```

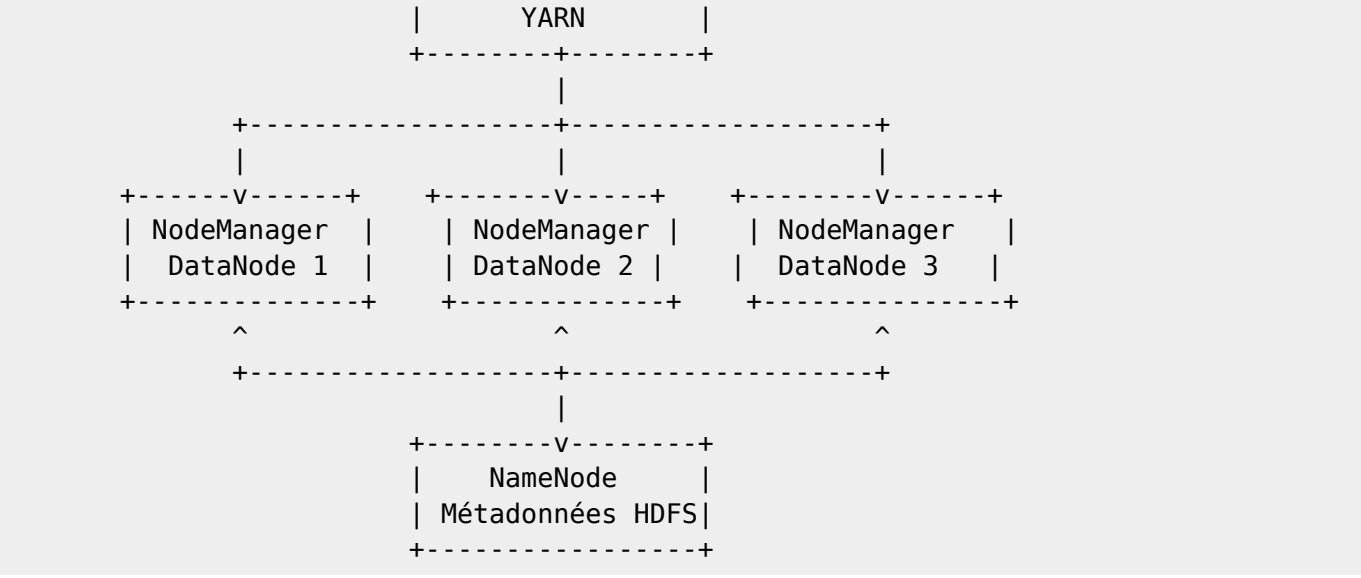
2. Objectifs

À la fin du TD, vous devez être capables de :

- distinguer Hadoop, HDFS, YARN et MapReduce ;
- expliquer le rôle du NameNode et des DataNodes ;
- expliquer le découpage d'un fichier en blocs ;
- observer la réplication d'un fichier ;
- vérifier le comportement d'HDFS lorsqu'un DataNode devient indisponible ;
- identifier le rôle du ResourceManager et des NodeManagers ;
- soumettre un traitement à YARN ;
- relier HDFS, YARN et un moteur de traitement.

3. Architecture utilisée

```
                    +-----------------+
                    | ResourceManager |
```



Composant	Rôle
NameNode	Gère les métadonnées HDFS
DataNode	Stocke les blocs de données
ResourceManager	Gère les ressources YARN
NodeManager	Gère les ressources d’un nœud
HDFS	Stocke les données
YARN	Attribue les ressources aux applications
MapReduce	Modèle de traitement distribué

4. Préparation de l’environnement

4.1 Prérequis

Installer :

- Docker ;
- Docker Compose ;
- Git.

Vérifier l’installation :

```
docker --version
docker compose version
git --version
```

Une dizaine de gigaoctets d’espace disque libre est recommandée.

4.2 Récupération de l’environnement Hadoop

Nous utilisons une image Docker Hadoop préconfigurée pour un environnement pédagogique.

```
git clone https://github.com/big-data-europe/docker-hadoop.git
```

```
cd docker-hadoop
```

Créer le fichier docker-compose-v2.yml :

```
version: "3"

services:
  namenode:
    image: bde2020/hadoop-namenode:2.0.0-hadoop3.2.1-java8
    container_name: namenode
    restart: always
    ports:
      - 9870:9870
      - 9000:9000
    volumes:
      - hadoop_namenode:/hadoop/dfs/name
    environment:
      - CLUSTER_NAME=test
    env_file:
      - ./hadoop.env
  datanode1:
    image: bde2020/hadoop-datanode:2.0.0-hadoop3.2.1-java8
    container_name: datanode1
    restart: always
    ports:
      - "9864:9864"
    volumes:
      - hadoop_datanode1:/hadoop/dfs/data
    environment:
      SERVICE_PRECONDITION: "namenode:9870"
    env_file:
      - ./hadoop.env
  datanode2:
    image: bde2020/hadoop-datanode:2.0.0-hadoop3.2.1-java8
    container_name: datanode2
    restart: always
    ports:
      - "9865:9864"
    volumes:
      - hadoop_datanode2:/hadoop/dfs/data
    environment:
      SERVICE_PRECONDITION: "namenode:9870"
    env_file:
      - ./hadoop.env
  datanode3:
    image: bde2020/hadoop-datanode:2.0.0-hadoop3.2.1-java8
    container_name: datanode3
    restart: always
    ports:
      - "9866:9864"
    volumes:
      - hadoop_datanode3:/hadoop/dfs/data
    environment:
      SERVICE_PRECONDITION: "namenode:9870"
```

```
    env_file:
      - ./hadoop.env
  resourcemanager:
    image: bde2020/hadoop-resourcemanager:2.0.0-hadoop3.2.1-java8
    container_name: resourcemanager
    restart: always
    ports:
      - "8088:8088"
    environment:
      SERVICE_PRECONDITION: "namenode:9000 namenode:9870 datanode1:9864"
    env_file:
      - ./hadoop.env
  nodemanager1:
    image: bde2020/hadoop-nodemanager:2.0.0-hadoop3.2.1-java8
    container_name: nodemanager
    restart: always
    ports:
      - "8042:8042"
    environment:
      SERVICE_PRECONDITION: "namenode:9000 namenode:9870 datanode1:9864
resourcemanager:8088"
    env_file:
      - ./hadoop.env
  historyserver:
    image: bde2020/hadoop-historyserver:2.0.0-hadoop3.2.1-java8
    container_name: historyserver
    restart: always
    ports:
      - "8188:8188"
    environment:
      SERVICE_PRECONDITION: "namenode:9000 namenode:9870 datanode1:9864
resourcemanager:8088"
    volumes:
      - hadoop_historyserver:/hadoop/yarn/timeline
    env_file:
      - ./hadoop.env
volumes:
  hadoop_namenode:
  hadoop_datanode1:
  hadoop_datanode2:
  hadoop_datanode3:
  hadoop_historyserver:
```

Démarrer l'environnement :

```
docker compose -f docker-compose-v2.yml up -d
```

Vérifier l'état des conteneurs :

```
docker compose -f docker-compose-v2.yml ps
```

Vous devez retrouver notamment des services proches de :

```
namenode
datanode1
datanode2
datanode3
resourcemanager
nodemanager
historyserver
```

Vérifier les hosts créés :

```
docker exec -it namenode hdfs dfsadmin -report
```

4.3 Interfaces web

Service	Adresse
NameNode	http://localhost:9870
ResourceManager	http://localhost:8088
NodeManager	http://localhost:8042
DataNode1	http://localhost:9864
DataNode2	http://localhost:9865
DataNode3	http://localhost:9866



Par défaut, seul le port du NameNode (9870) est exposé sur l'hôte. Pour accéder aux interfaces ResourceManager (8088) et NodeManager (8042), il faut ajouter manuellement les mappings ports: dans le docker-compose.yml avant de démarrer les conteneurs, ou les ajouter puis exécuter docker compose up -d pour recréer les conteneurs concernés avec les nouveaux ports.

5. Accéder aux conteneurs

Pour ouvrir un terminal dans le NameNode :

```
docker compose exec namenode bash
```

Pour ouvrir un terminal dans le ResourceManager :

```
docker compose exec resourcemanager bash
```

Les commandes Hadoop sont exécutées dans le conteneur `namenode`, sauf indication contraire.

Mission 1 - Comprendre le problème

6. Architecture initiale

On considère l'architecture suivante :

```
Serveur unique
│
├── ventes_j1.csv
├── ventes_j2.csv
├── clics_j1.csv
└── application.log
```



Cette architecture atteint rapidement ses limites : le disque finit par saturer, la panne d'une seule machine rend toutes les données inaccessibles, et plusieurs traitements lancés simultanément se disputent les mêmes ressources (CPU, disque, mémoire). Augmenter la capacité signifie changer de machine ou ajouter du matériel, ce qui ne se fait ni rapidement ni indéfiniment. C'est exactement ce type de limite qu'Hadoop a été conçu pour dépasser, en répartissant à la fois le stockage (HDFS) et le calcul (YARN) sur plusieurs machines.

Mission 2 - Préparer les données

7. Générer un fichier de ventes

Sur votre machine hôte, créer un fichier `generer_donnees.py` :

```
import argparse
import csv
import random
from datetime import date, timedelta

parser = argparse.ArgumentParser()
parser.add_argument("--lignes", type=int, default=200000, help="Nombre de lignes à
générer")
args = parser.parse_args()
categories = ["informatique", "maison", "sport", "livres"]
villes = ["Paris", "Lyon", "Lille", "Nantes", "Bordeaux"]
with open("ventes.csv", "w", newline="") as fichier:
    writer = csv.writer(fichier)
    writer.writerow([
        "date",
        "commande_id",
```

```
        "client_id",
        "categorie",
        "ville",
        "montant"
    ])
date_depart = date(2026, 1, 1)

for i in range(args.lignes):
    jour = date_depart + timedelta(days=random.randint(0, 30))
    commande_id = f"C{i:07d}"
    client_id = f"CL{random.randint(1, 50000):05d}"
    categorie = random.choice(categories)
    ville = random.choice(villes)
    montant = round(random.uniform(5, 500), 2)

    writer.writerow([
        jour,
        commande_id,
        client_id,
        categorie,
        ville,
        montant
    ])
```

Exécuter :

```
python3 generer_donnees.py
```

Observer la taille du fichier :

```
ls -lh ventes.csv
wc -l ventes.csv
head ventes.csv
```



Par défaut, HDFS découpe les fichiers en blocs de 128 Mo. Le fichier généré ici est volontairement petit (quelques Mo au plus) : il tiendra donc dans un seul bloc, et c'est normal.

L'objectif de cette étape est seulement de vérifier que le fichier est bien généré et de comprendre son contenu — l'observation des blocs se fera en Mission 4, sur un fichier généré en plus grande quantité. Pour générer un fichier plus volumineux (utile pour la Mission 4) :

```
python3 generer_donnees.py --lignes 5000000
```

Mission 3 - Stocker les données dans HDFS

8. Créer une organisation de fichiers

Dans le conteneur namenode :

```
hdfs dfs -mkdir -p /data/ecommerce/raw/ventes
hdfs dfs -mkdir -p /data/ecommerce/raw/logs
hdfs dfs -mkdir -p /data/ecommerce/processed
```

Depuis la machine hôte, copier le fichier dans le conteneur :

```
docker cp ventes.csv docker-hadoop-namenode-1:/tmp/ventes.csv
```

Le nom exact du conteneur peut être vérifié avec :

```
docker ps
```

Dans le conteneur namenode :

```
hdfs dfs -put /tmp/ventes.csv /data/ecommerce/raw/ventes/
```

Vérifier la présence du fichier :

```
hdfs dfs -ls -h /data/ecommerce/raw/ventes
```

Afficher quelques lignes :

```
hdfs dfs -cat /data/ecommerce/raw/ventes/ventes.csv | head
```

Répondre :

1. La commande `hdfs dfs -put` ressemble-t-elle à une copie classique ?
2. Qu'est-ce qui est différent après son exécution ?

Mission 4 - Observer les blocs HDFS

9. Examiner la structure physique du fichier

Exécuter :

```
hdfs fsck /data/ecommerce/raw/ventes/ventes.csv \
  -files \
  -blocks \
  -locations
```

Relever :

- la taille logique du fichier ;
- le nombre de blocs ;
- la taille de chaque bloc ;
- les DataNodes utilisés ;
- le facteur de réplication.

Élément	Valeur
Taille du fichier	
Nombre de blocs	
Facteur de réplication	
DataNodes utilisés	
Taille du bloc	

Répondre :

1. Le fichier est-il stocké comme un objet unique sur un seul serveur ?
2. Les blocs sont-ils tous stockés sur le même DataNode ?
3. Pourquoi le fichier peut-il être lu si un DataNode devient indisponible ?
4. La taille logique correspond-elle à l'espace physique utilisé ?
5. Quelle est la relation entre découpage en blocs et traitement parallèle ?

10. Rendre les blocs plus visibles

Dans un environnement réel, les blocs HDFS ont généralement une taille importante. Pour observer plus facilement le découpage, vous pouvez utiliser un fichier plus volumineux ou modifier la configuration du cluster avant son démarrage.

Dans `hadoop.env`, repérer ou ajouter :

```
HDFS_CONF_dfs_blocksize=1048576
HDFS_CONF_dfs_replication=2
```

Cette configuration correspond à :

- une taille de bloc de 1 MiB ;
- deux copies de chaque bloc.

Reconstruire l’environnement :

```
docker compose down -v
docker compose up -d
```

> Cette étape supprime les volumes Docker associés au cluster.

Recréer les répertoires HDFS et importer à nouveau le fichier.

Relancer :

```
hdfs fsck /data/ecommerce/raw/ventes/ventes.csv \
  -files \
  -blocks \
  -locations
```

Dessiner un schéma représentant la répartition des blocs.

Exemple :

```
ventes.csv
|
+-- Bloc 0 -> DataNode 1, DataNode 2
+-- Bloc 1 -> DataNode 2, DataNode 3
+-- Bloc 2 -> DataNode 1, DataNode 3
```

Mission 5 - Comprendre la réplication

11. Modifier le facteur de réplication

Vérifier le facteur de réplication :

```
hdfs fsck /data/ecommerce/raw/ventes/ventes.csv \
  -files \
  -blocks \
  -locations
```

Modifier la réplication du fichier :

```
hdfs dfs -setrep -w 3 \
```

```
  /data/ecommerce/raw/ventes/ventes.csv
```

Vérifier le résultat :

```
hdfs fsck /data/ecommerce/raw/ventes/ventes.csv \
  -files \
  -blocks \
  -locations
```

Répondre :

1. Que signifie un facteur de réplication égal à 3 ?
2. La réplication crée-t-elle trois fichiers visibles ?
3. La taille logique change-t-elle ?
4. Pourquoi la réplication améliore-t-elle la disponibilité ?
5. Quel est son coût en espace disque ?
6. Pourquoi ne pas utiliser un facteur très élevé pour tous les fichiers ?

Mission 6 - Simuler une panne

12. Observer l’état initial du cluster

Dans le conteneur `namenode` :

```
hdfs dfsadmin -report
```

Relever :

- le nombre de DataNodes disponibles ;
- la capacité totale ;
- l’espace utilisé ;
- le nombre de blocs présents.

Depuis la machine hôte :

```
docker compose ps
```

Identifier le conteneur correspondant à un DataNode.

13. Arrêter un DataNode

Arrêter un DataNode :

```
docker compose stop datanode
```

Si plusieurs DataNodes portent un suffixe :

```
docker compose stop datanode1
```

ou :

```
docker compose stop datanode2
```

Vérifier l'état :

```
docker compose ps
```

Puis :

```
hdfs dfsadmin -report
```

Examiner le fichier :

```
hdfs fsck /data/ecommerce/raw/ventes/ventes.csv \
  -files \
  -blocks \
  -locations
```

Lire le fichier :

```
hdfs dfs -cat /data/ecommerce/raw/ventes/ventes.csv | head
```

Répondre :

1. Le fichier est-il toujours lisible ?
2. Certains blocs ont-ils perdu une copie ?
3. Le NameNode connaît-il toujours les blocs ?
4. Quelle différence entre la perte d'un DataNode et la perte de toutes les copies d'un bloc ?
5. Dans quelles conditions HDFS ne pourrait-il plus reconstruire le fichier ?

Redémarrer le DataNode :

```
docker compose start datanode
```

Puis vérifier :

```
hdfs dfsadmin -report
```

Mission 7 - Comprendre YARN

14. Problème à résoudre

Deux équipes souhaitent lancer simultanément :

- équipe A : calcul du chiffre d'affaires ;
- équipe B : analyse des logs.

Répondre :

1. Quel composant connaît les ressources disponibles ?
2. Quel composant décide les ressources attribuées ?
3. Quel composant supervise les ressources d'un nœud ?
4. HDFS peut-il attribuer de la mémoire ou des processeurs ?
5. Spark peut-il fonctionner sans gestionnaire de ressources dans un cluster partagé ?

15. Observer les nœuds YARN

Dans le conteneur `resourcemanager` :

```
yarn node -list
```

Afficher les applications :

```
yarn application -list
```

Afficher les informations du cluster :

```
yarn cluster
```

Consulter l'interface :

```
http://localhost:8088
```

Identifier :

- les nœuds disponibles ;
- la mémoire totale ;
- la mémoire utilisée ;
- les applications en cours ;
- les applications terminées.

Compléter :

Composant	**Fonction observée**
ResourceManager	
NodeManager	
ApplicationMaster	
Conteneur YARN	

Mission 8 - Soumettre un traitement à YARN

16. Exemple MapReduce intégré

Rechercher les exemples MapReduce :

```
find / -name "hadoop-mapreduce-examples*.jar" 2>/dev/null
```

Créer un petit fichier texte dans HDFS :

```
hdfs dfs -mkdir -p /data/ecommerce/test
hdfs dfs -put /etc/hosts /data/ecommerce/test/
```

Lancer `wordcount` :

```
hadoop jar \
/usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-*.jar \
wordcount \
/data/ecommerce/test \
/data/ecommerce/processed/wordcount
```

Si le répertoire existe déjà :

```
hdfs dfs -rm -r /data/ecommerce/processed/wordcount
```

Afficher le résultat :

```
hdfs dfs -cat /data/ecommerce/processed/wordcount/part-r-00000
```

Pendant l'exécution, consulter :

```
yarn application -list
```

Puis :

```
yarn application -list -appStates ALL
```

Répondre :

1. Quelle application apparaît dans YARN ?
2. Quel est son état ?
3. Quel rôle joue le ResourceManager ?
4. Où se trouvent les données d'entrée ?
5. Où sont écrits les résultats ?
6. Le résultat est-il produit dans un fichier unique ?

Mission 9 - Relier HDFS, YARN et Spark

17. Lecture depuis HDFS

Le code suivant correspond à un traitement Spark :

```
df = spark.read \
    .option("header", True) \
    .option("inferSchema", True) \
    .csv("hdfs:///data/ecommerce/raw/ventes/ventes.csv")

df.printSchema()
df.show(5)
```

Comparer avec :

```
spark.read.csv("ventes.csv")
```

Répondre :

1. Quelle est la différence entre ces chemins ?
2. Spark stocke-t-il lui-même le fichier ?
3. Quel composant fournit les données ?
4. Quel composant attribue les ressources à Spark ?

5. Que se passe-t-il si les données sont réparties sur plusieurs DataNodes ?

18. Traitement analytique

Créer `analyse_ventes.py` :

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import sum, count, avg

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

df = (
    spark.read
    .option("header", True)
    .option("inferSchema", True)
    .csv("hdfs:///data/ecommerce/raw/ventes/ventes.csv")
)

df.printSchema()

resultat = (
    df.groupBy("categorie")
    .agg(
        count("commande_id").alias("nombre_commandes"),
        sum("montant").alias("chiffre_affaires"),
        avg("montant").alias("panier_moyen")
    )
)

resultat.show()

resultat.write.mode("overwrite").parquet(
    "hdfs:///data/ecommerce/processed/ca_par_categorie"
)

spark.stop()
```

Selon l'environnement Spark disponible :

```
spark-submit \
  --master yarn \
  --deploy-mode cluster \
  --num-executors 2 \
  --executor-memory 1G \
  analyse_ventes.py
```

Observer l'application :

```
yarn application -list
```

Vérifier le résultat :

```
hdfs dfs -ls /data/ecommerce/processed/ca_par_categorie
```

Les fichiers Parquet doivent être relus avec Spark plutôt qu'avec `cat`.

Mission 10 - Synthèse de l'architecture

Compléter :

```
Fichier de ventes
       |
       v
   _________
   _________
       |
       v
Découpage en blocs
       |
       v
   _________
   _________
       |
       v
Attribution de ressources
       |
       v
   _________
       |
       v
Résultats stockés dans HDFS
```

Associer chaque composant à son rôle :

```
HDFS
YARN
NameNode
DataNode
ResourceManager
Spark
MapReduce
```

Travail à rendre

Un fichier `compte_rendu.md` contenant :

Partie 1 - Architecture

- un schéma de l'architecture Hadoop ;
- le rôle de chaque composant ;
- la différence entre Hadoop, HDFS et YARN.

Partie 2 - Blocs

- la taille du fichier ;
- le nombre de blocs ;
- le facteur de réplication ;
- les DataNodes utilisés ;
- un schéma de répartition des blocs.

Partie 3 - Tolérance aux pannes

- état du cluster avant l'arrêt d'un DataNode ;
- état du cluster après l'arrêt ;
- résultat de la lecture du fichier ;
- explication du comportement observé.

Partie 4 - YARN

- liste des nœuds ;
- observation d'une application ;
- rôle du ResourceManager ;
- rôle des NodeManagers ;
- explication de la notion de conteneur.

Partie 5 - Traitement

- commande de soumission ;
- résultat obtenu ;
- emplacement des résultats dans HDFS ;
- schéma du parcours des données.

Partie 6 - Conclusion

Répondre en une dizaine de lignes :

> En quoi HDFS et YARN répondent-ils à des problèmes différents, et pourquoi sont-ils complémentaires pour exécuter un traitement distribué ?

Commandes utiles

```
# État des conteneurs
docker compose ps

# Entrer dans un conteneur
docker compose exec namenode bash

# Lister HDFS
hdfs dfs -ls -h /chemin

# Créer un répertoire
hdfs dfs -mkdir -p /chemin

# Copier vers HDFS
hdfs dfs -put fichier /chemin/

# Lire un fichier
hdfs dfs -cat /chemin/fichier

# Supprimer
hdfs dfs -rm -r /chemin

# Examiner les blocs
hdfs fsck /chemin/fichier -files -blocks -locations

# État des DataNodes
hdfs dfsadmin -report

# Nœuds YARN
yarn node -list

# Applications YARN
yarn application -list

# Applications terminées
yarn application -list -appStates ALL
```

Schéma final à retenir

```
                    YARN
            +--------------------------+
            | ResourceManager          |
            | NodeManagers             |
```

```
              | Allocation des ressources|
              +------------+-------------+
                           |
                           v
                         Spark
                    ou MapReduce
                           |
                           v
                          HDFS
              +-------------------------+
              | NameNode                |
              | DataNodes               |
              | Blocs et réplication    |
              +-------------------------+
```

HDFS stocke. YARN organise les ressources. Spark ou MapReduce traitent.

From:
http://slamwiki2.kobject.net/ - **SlamWiki 2.1**

Permanent link:
http://slamwiki2.kobject.net/eadl/bloc5/fm3/td1?rev=1789594440

Last update: **2026/09/16 23:34**