

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.

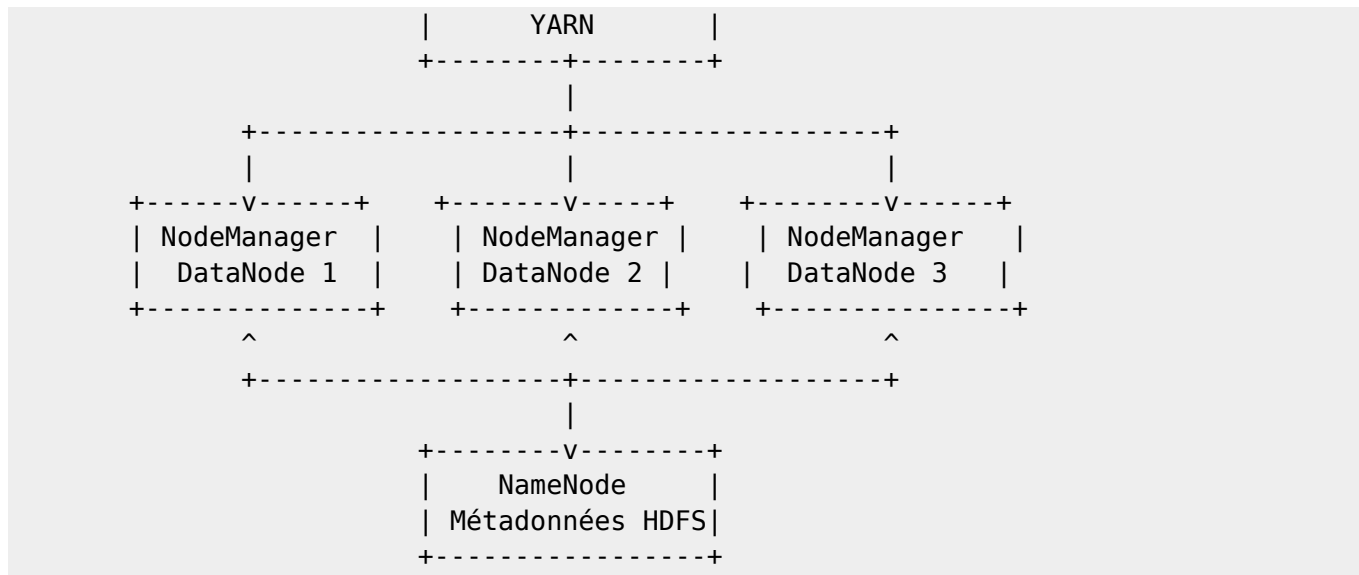
2. Objectifs

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 :

```
<code bash> docker -version docker compose version git -version </sxh>
```

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.

```
<code bash> git clone https://github.com/big-data-europe/docker-hadoop.git cd docker-hadoop </sxh>
```

Démarrer l'environnement :

```
<code bash> docker compose up -d </sxh>
```

Vérifier l'état des conteneurs :

```
<code bash> docker compose ps </sxh>
```

Vous devez retrouver notamment des services proches de :

```
namenode
datanode
resourcemanager
nodemanager
historyserver
```

4.3 Interfaces web

Service	Adresse
NameNode	http://localhost:9870
ResourceManager	http://localhost:8088
NodeManager	http://localhost:8042

5. Accéder aux conteneurs

Pour ouvrir un terminal dans le NameNode :

```
<code bash> docker compose exec namenode bash </sxh>
```

Pour ouvrir un terminal dans le ResourceManager :

```
<code bash> docker compose exec resourcemanager bash </sxh>
```

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
```

Répondre aux questions suivantes :

1. Que se passe-t-il si le disque du serveur est plein ?
2. Que se passe-t-il si le serveur devient indisponible ?
3. Peut-on augmenter facilement la capacité de stockage ?

4. Plusieurs traitements peuvent-ils utiliser efficacement cette machine ?
5. Quel mécanisme permettrait de répartir les données sur plusieurs machines ?

Conserver vos réponses dans un fichier `reponses.md`.

Mission 2 — Préparer les données

7. Générer un fichier de ventes

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

```
<code python> import csv import random from datetime import date, timedelta
```

```
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(200000):
        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
        ])
```

```
</sxh>
```

Exécuter :

```
<code bash> python generer_donnees.py </sxh>
```

Observer la taille du fichier :

```
<code bash> ls -lh ventes.csv wc -l ventes.csv head ventes.csv </sxh>
```

Répondre :

1. Quelle est la taille du fichier ?
2. Combien contient-il de lignes ?
3. Est-il suffisamment volumineux pour observer plusieurs blocs HDFS ?
4. Comment produire un fichier beaucoup plus volumineux ?

Mission 3 — Stocker les données dans HDFS

8. Créer une organisation de fichiers

Dans le conteneur namenode :

```
<code bash> hdfs dfs -mkdir -p /data/ecommerce/raw/ventes hdfs dfs -mkdir -p /data/ecommerce/raw/logs hdfs  
dfs -mkdir -p /data/ecommerce/processed </sxh>
```

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

```
<code bash> docker cp ventes.csv docker-hadoop-namenode-1:/tmp/ventes.csv </sxh>
```

Le nom exact du conteneur peut être vérifié avec :

```
<code bash> docker ps </sxh>
```

Dans le conteneur namenode :

```
<code bash> hdfs dfs -put /tmp/ventes.csv /data/ecommerce/raw/ventes/ </sxh>
```

Vérifier la présence du fichier :

```
<code bash> hdfs dfs -ls -h /data/ecommerce/raw/ventes </sxh>
```

Afficher quelques lignes :

```
<code bash> hdfs dfs -cat /data/ecommerce/raw/ventes/ventes.csv | head </sxh>
```

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 :

```
<code bash> hdfs fsck /data/ecommerce/raw/ventes/ventes.csv \
```

1. files \
2. blocks \
3. locations

```
</sxh>
```

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 :

```
<code bash> docker compose down -v docker compose up -d </sxh>
```

Cette étape supprime les volumes Docker associés au cluster.

Recréer les répertoires HDFS et importer à nouveau le fichier.

Relancer :

```
<code bash> hdfs fsck /data/ecommerce/raw/ventes/ventes.csv \
```

1. files \
2. blocks \
3. locations

```
</sxh>
```

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 :

```
<code bash> hdfs fsck /data/ecommerce/raw/ventes/ventes.csv \
```

1. files \
2. blocks \
3. locations

```
</sxh>
```

Modifier la réplication du fichier :

```
<code bash> hdfs dfs -setrep -w 3 \
```

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

```
</sxh>
```

Vérifier le résultat :

```
<code bash> hdfs fsck /data/ecommerce/raw/ventes/ventes.csv \
```

1. files \
2. blocks \
3. locations

```
</sxh>
```

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 :

```
<code bash> hdfs dfsadmin -report </sxh>
```

Relever :

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

Depuis la machine hôte :

```
<code bash> docker compose ps </sxh>
```

Identifier le conteneur correspondant à un DataNode.

13. Arrêter un DataNode

Arrêter un DataNode :

```
<code bash> docker compose stop datanode </sxh>
```

Si plusieurs DataNodes portent un suffixe :

```
<code bash> docker compose stop datanode1 </sxh>
```

ou :

```
<code bash> docker compose stop datanode2 </sxh>
```

Vérifier l'état :

```
<code bash> docker compose ps </sxh>
```

Puis :

```
<code bash> hdfs dfsadmin -report </sxh>
```

Examiner le fichier :

```
<code bash> hdfs fsck /data/ecommerce/raw/ventes/ventes.csv \
```

1. files \
2. blocks \
3. locations

```
</sxh>
```

Lire le fichier :

```
<code bash> hdfs dfs -cat /data/ecommerce/raw/ventes/ventes.csv | head </sxh>
```

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 :

```
<code bash> docker compose start datanode </sxh>
```

Puis vérifier :

```
<code bash> hdfs dfsadmin -report </sxh>
```

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 :

```
<code bash> yarn node -list </sxh>
```

Afficher les applications :

```
<code bash> yarn application -list </sxh>
```

Afficher les informations du cluster :

```
<code bash> yarn cluster </sxh>
```

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 :

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

Créer un petit fichier texte dans HDFS :

```
<code bash> hdfs dfs -mkdir -p /data/ecommerce/test hdfs dfs -put /etc/hosts /data/ecommerce/test/ </sxh>
```

Lancer wordcount :

```
<code bash> hadoop jar \ /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-*.jar \  
wordcount \ /data/ecommerce/test \ /data/ecommerce/processed/wordcount </sxh>
```

Si le répertoire existe déjà :

```
<code bash> hdfs dfs -rm -r /data/ecommerce/processed/wordcount </sxh>
```

Afficher le résultat :

```
<code bash> hdfs dfs -cat /data/ecommerce/processed/wordcount/part-r-00000 </sxh>
```

Pendant l'exécution, consulter :

```
<code bash> yarn application -list </sxh>
```

Puis :

```
<code bash> yarn application -list -appStates ALL </sxh>
```

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 :

```
<code python> df = spark.read \
```

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

```
df.printSchema() df.show(5) </sxh>
```

Comparer avec :

```
<code python> spark.read.csv("ventes.csv") </sxh>
```

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` :

```
<code python> 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() </sxh>
```

Selon l'environnement Spark disponible :

```
<code bash> spark-submit \
```

1. -master yarn \
2. -deploy-mode cluster \
3. -num-executors 2 \
4. -executor-memory 1G \

```
analyse_ventes.py </sxh>
```

Observer l'application :

```
<code bash> yarn application -list </sxh>
```

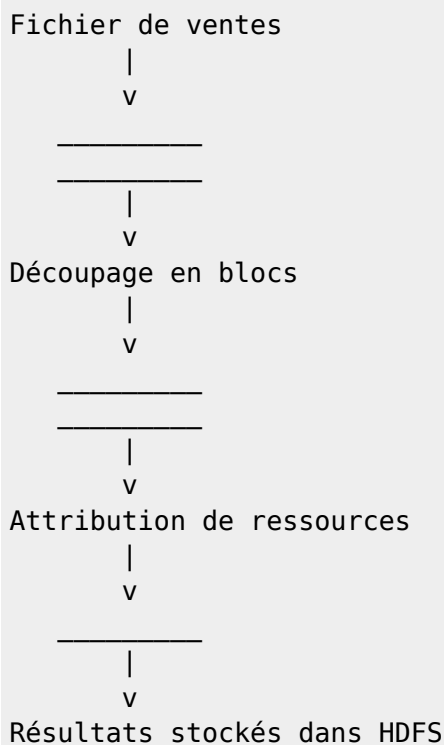
Vérifier le résultat :

```
<code bash> hdfs dfs -ls /data/ecommerce/processed/ca_par_categorie </sxh>
```

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

Mission 10 — Synthèse de l'architecture

Compléter :



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

```
<code bash> # 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

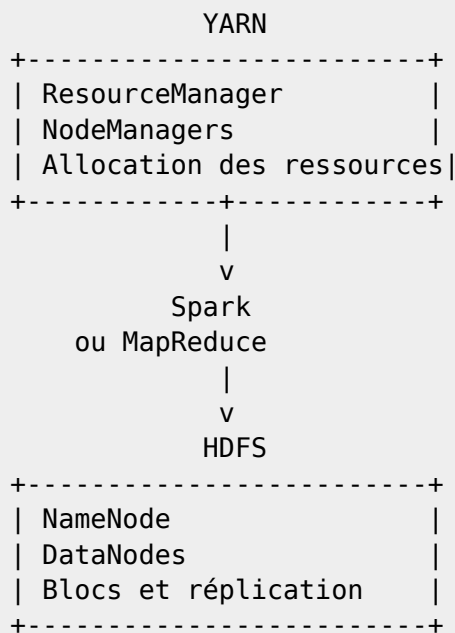
# État des DataNodes hdfs dfsadmin -report

# Nœuds YARN yarn node -list

# Applications YARN yarn application -list

# Applications terminées yarn application -list -appStates ALL </sxh>
```

Schéma final à retenir



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

From:

<http://slamwiki2.kobject.net/> - SlamWiki 2.1

Permanent link:

<http://slamwiki2.kobject.net/eadl/bloc5/fm3/td1?rev=1789165143>

Last update: 2026/09/12 00:19

