

ANALYSE DES DONNEES AVEC HADDOP

NB : Étant donné que nous travaillons sur le terminal de la machine, nous ne pouvons pas fournir de lien vers les codes sources. C'est pourquoi nous avons rédigé ce document avec des captures d'écran pour illustrer nos actions et pour pouvoir revenir plus en détail sur les codes lors de la présentation.

Objectif du TP :

La base de données "**purchases.txt**" contient plus de 4 millions de lignes et 6 variables, représentant des transactions commerciales enregistrées avec la date, l'heure, la ville, la catégorie de produit acheté, le montant de la transaction et le moyen de paiement utilisé. Chaque ligne correspond à une vente effectuée dans une ville donnée, avec des catégories variées telles que les vêtements, la musique ou encore l'électronique, et des paiements réalisés via différents moyens comme Visa, MasterCard ou en espèces. L'objectif de l'analyse est de traiter cette base avec **Hadoop Streaming**, en utilisant des scripts Python pour calculer les ventes totales par ville, ou par catégories, de calculer la moyenne des ventes etc... Il sera donc question de répondre aux cinq questions qui suivent :

- **Total des ventes par ville** : Combien d'argent a été dépensé dans chaque ville ?
- **Total des ventes par catégorie de produit** : Quel est le total des ventes pour chaque catégorie ?
- **Total des ventes par méthode de paiement** : Quelles sont les méthodes de paiement les plus utilisées ?
- **Ventes moyennes par transaction** : Quelle est la dépense moyenne par transaction dans chaque ville et par catégorie ?
- **Analyse de la répartition temporelle** : sur quelle période de la journée on note plus de ventes ?

1- Lancement des conteneurs Hadoop dans Docker

Avant de charger les données dans HDFS, il est nécessaire de démarrer les conteneurs Docker contenant les services Hadoop. Cette étape permet d'exécuter Hadoop dans un environnement isolé et reproductible. Pour cela, on utilise une image préconfigurée d'Hadoop et on lance un

conteneur qui inclut HDFS et YARN. Une fois les conteneurs en cours d'exécution, on peut interagir avec le système de fichiers HDFS pour charger et traiter les données efficacement.

Pour le faire, on ouvre Docker et on active les conteneurs comme le montre l'image suivante :

Une fois les conteneurs lancés nous pouvons vérifier s'ils fonctionnent en exécutant : « `docker ps` ». Voici le résultat :

```
C:\Users\Lenovo>docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
af7ea6e21ffa	liliasfaxi/hadoop-cluster:latest	"sh -c 'service ssh ..."	34 hours ago	Up 2 hours	0.0.0.0:8041->8042/tcp
de6dc92e5786	liliasfaxi/hadoop-cluster:latest	"sh -c 'service ssh ..."	34 hours ago	Up 2 hours	0.0.0.0:8040->8042/tcp
e914dabfc81f	liliasfaxi/hadoop-cluster:latest	"sh -c 'service ssh ..."	34 hours ago	Up 2 hours	0.0.0.0:7077->7077/tcp, 0.0.0.0:8088->8088/tcp, 0.0.0.0:9870->9870/tcp, 0.0.0.0:16010->16010/tcp

2- Entrer dans le conteneur

Dans ce cas pratique, nous allons entrer dans le conteneur principal (master) pour commencer à utiliser Hadoop et YARN. Pour cela, nous devons exécuter la commande suivante dans notre terminal : « `docker exec -it hadoop-master bash` »







Une fois cette commande exécutée, nous serons connectés au conteneur, et nous verrons un prompt semblable à celui-ci :

```
Lenovo@Baba-AS MINGW64 ~  
$ docker exec -it hadoop-master bash  
root@hadoop-master:~#
```

Nous nous retrouvons ainsi dans le shell du **namenode** du cluster Hadoop, ce qui nous permet de manipuler et de gérer l'ensemble de l'infrastructure Hadoop.

3- Lancer Hadoop et YARN

Une fois dans le conteneur, la première action que nous devons effectuer est de démarrer Hadoop et YARN. Un script d'initialisation, « `start-hadoop.sh` », nous permet de lancer ces services de manière simple et rapide.

Name	Container ID	Image	Port(s)	CPU (%)	Actions
hadoop-worker2	af7ea6e21ffa	liliasfaxi/hadoop-cluster:latest	8041:8042 ↗	0.69%	 
hadoop-worker1	de6dc92e5786	liliasfaxi/hadoop-cluster:latest	8040:8042 ↗	0.77%	 
hadoop-master	e914dabfc81f	liliasfaxi/hadoop-cluster:latest	16010:16010 ↗ Show all ports (4)	1.36%	 

```

root@hadoop-master:~# ./start-hadoop.sh

Starting namenodes on [hadoop-master]
hadoop-master: warning: Permanently added 'hadoop-master' (ED25519) to the list of known hosts.
hadoop-master: WARNING: HADOOP_NAMENODE_OPTS has been replaced by HDFS_NAMENODE_OPTS. Using value of HADOOP_NAMENODE_OPTS.
hadoop-master: namenode is running as process 171. Stop it first and ensure /tmp/hadoop-root-namenode.pid file is empty before retry.
Starting datanodes
WARNING: HADOOP_SECURE_DN_LOG_DIR has been replaced by HADOOP_SECURE_LOG_DIR. Using value of HADOOP_SECURE_DN_LOG_DIR.
hadoop-worker1: warning: Permanently added 'hadoop-worker1' (ED25519) to the list of known hosts.
hadoop-worker2: warning: Permanently added 'hadoop-worker2' (ED25519) to the list of known hosts.
hadoop-worker1: WARNING: HADOOP_SECURE_DN_LOG_DIR has been replaced by HADOOP_SECURE_LOG_DIR. Using value of HADOOP_SECURE_DN_LOG_DIR.
hadoop-worker1: WARNING: HADOOP_DATANODE_OPTS has been replaced by HDFS_DATANODE_OPTS. Using value of HADOOP_DATANODE_OPTS.
hadoop-worker1: datanode is running as process 76. Stop it first and ensure /tmp/hadoop-root-datanode.pid file is empty before retry.
hadoop-worker2: WARNING: HADOOP_SECURE_DN_LOG_DIR has been replaced by HADOOP_SECURE_LOG_DIR. Using value of HADOOP_SECURE_DN_LOG_DIR.
hadoop-worker2: WARNING: HADOOP_DATANODE_OPTS has been replaced by HDFS_DATANODE_OPTS. Using value of HADOOP_DATANODE_OPTS.
hadoop-worker2: datanode is running as process 75. Stop it first and ensure /tmp/hadoop-root-datanode.pid file is empty before retry.

```

Ce script lancera les composants essentiels de Hadoop, y compris le **namenode** et les services **YARN** nécessaires pour la gestion des ressources du cluster. Après l'exécution du script, nous pourrions vérifier que tout fonctionne correctement en consultant l'interface web de YARN ou en analysant les logs du conteneur. Pour vérifier si les composants sont bien lancés, nous allons utiliser la commande « **jps** » :

```

root@hadoop-master:~# jps
632 ResourceManager
171 NameNode
1966 Jps
366 SecondaryNameNode
root@hadoop-master:~#

```

On voit ici que toutes les composantes sont bien lancées. Maintenant on peut manipuler des fichiers dans HDFS.

4- Manipuler des fichiers dans HDFS : Cas Pratique

Dans ce cas pratique, nous allons interagir avec le système de fichiers HDFS en utilisant des commandes Hadoop. Nous commencerons par créer un répertoire, puis chargerons notre fichier `purchages.txt` pour effectuer un traitement MapReduce.

4-1. Créer un répertoire dans HDFS

Tout d'abord, nous devons créer un répertoire appelé **input** dans HDFS ou nous allons mettre le fichier. Pour ce faire, nous utilisons la commande suivante : « `hdfs dfs -mkdir -p input` »

On peut vérifier que le répertoire est bien créé avec la commande : « `hdfs dfs -ls/` ». Cette commande liste tous les dossiers de HDFS.

Le résultat donne ceci :

```

root@hadoop-master:~# hdfs dfs -ls
Found 2 items
drwxr-xr-x - root supergroup          0 2025-02-19 13:57 input
drwxr-xr-x - root supergroup          0 2025-02-19 14:46 output1

```

Pour mettre le fichier purchases.txt dans le dossier input, on utilise la commande suivante :
« `hdfs dfs -put purchases.txt input` »

```
root@hadoop-master:~# hdfs dfs -ls input/
Found 4 items
-rw-r--r--  2 root supergroup      65529 2025-02-19 12:38 input/billets.csv
-rw-r--r--  2 root supergroup       878 2025-02-19 13:57 input/mapper.py
-rw-r--r--  2 root supergroup 211312924 2025-02-18 13:51 input/purchases.txt
-rw-r--r--  2 root supergroup       878 2025-02-19 13:57 input/reducer.py
root@hadoop-master:~#
```

Le fichier purchases.txt est bien visible.

4-2. Afficher les premières lignes du fichier purchases.txt

Si nous souhaitons consulter le contenu du fichier **purchases.txt**, en particulier ses premières lignes, nous pouvons utiliser la commande suivante : « `hdfs dfs -cat input/purchases.txt | head` »

Le résultat donne ceci :

```
root@hadoop-master:~# hdfs dfs -cat input/purchases.txt | head
2012-01-01    09:00    San Jose      Men's Clothing  214.05    Amex
2012-01-01    09:00    Fort Worth    Women's Clothing  153.57    Visa
2012-01-01    09:00    San Diego     Music    66.08    Cash
2012-01-01    09:00    Pittsburgh    Pet Supplies  493.51    Discover
2012-01-01    09:00    Omaha        Children's Clothing  235.63    MasterCard
2012-01-01    09:00    Stockton      Men's Clothing  247.18    MasterCard
2012-01-01    09:00    Austin        Cameras  379.6    Visa
2012-01-01    09:00    New York      Consumer Electronics  296.8    Cash
2012-01-01    09:00    Corpus Christi Toys    25.38    Discover
2012-01-01    09:00    Fort Worth    Toys    213.88    Visa
```

Une fois que notre fichier est dans HDFS, nous pouvons maintenant commencer à répondre aux questions posées un peu plus haut.

5- Réponses aux questions :

Nous allons réaliser le cas pratique pour la première question, car le procédé est identique pour les autres. Les résultats des autres questions ainsi que leurs traitements seront inclus dans les livrables finaux.

Total des ventes par ville : Combien d'argent a été dépensé dans chaque ville ?

Pour répondre à la question "Combien d'argent a été dépensé dans chaque ville ?", nous avons utilisé Hadoop MapReduce. Le processus comprend plusieurs étapes clés : écriture du code Mapper et Reducer, exécution du job MapReduce et affichage des résultats. Voici un aperçu de ce que nous avons fait.

5-1- Le code Mapper écrit en python.

Le rôle du Mapper est de lire chaque ligne du fichier purchases.txt, d'extraire la ville et le montant, puis d'émettre une paire clé-valeur. La clé est la ville, et la valeur est le montant dépensé. Voici le code :

```
#!/usr/bin/env python3
import sys

# Lecture de l'entrée ligne par ligne
for line in sys.stdin:
    # Supprimer les espaces en début et fin de ligne
    line = line.strip()

    # Séparer les colonnes en utilisant plusieurs espaces comme séparateur
    columns = line.split()

    # Vérifier que la ligne contient au moins 6 colonnes (pour éviter les erreurs d'index)
    if len(columns) < 6:
        continue # Ignorer les lignes mal formatées

    try:
        # Extraire la ville (3e colonne) et le montant (5e colonne)
        city = columns[2]
        amount = float(columns[4])

        # Émettre la paire clé-valeur (ville, montant) séparée par une tabulation
        print(f"{city}\t{amount}")
    except ValueError:
        continue # Ignorer les lignes où le montant n'est pas un nombre valide
```

5-2 - Le code Reducer écrit en Python

Le rôle du Reducer est de recevoir les paires clé-valeur émises par le Mapper, de regrouper les montants par ville, puis de calculer le total des ventes pour chaque ville. Voici le code :

```

current_city = None
total_sales = 0.0

# Lecture des résultats du Mapper ligne par ligne
for line in sys.stdin:
    # Supprimer les espaces en début et fin de ligne
    line = line.strip()

    # Vérifier que la ligne contient une tabulation avant de séparer
    if '\t' not in line:
        continue # Ignorer les lignes mal formatées

    try:
        city, amount = line.split('\t')
        amount = float(amount) # Convertir en float
    except ValueError:
        continue # Ignorer les lignes où le montant est invalide

    # Si la ville change, afficher l'ancienne et réinitialiser
    if current_city and current_city != city:
        print(f"{current_city}\t{total_sales}")
        total_sales = 0.0 # Réinitialiser le total

    current_city = city
    total_sales += amount # Ajouter au total de la ville

# Afficher la dernière ville après la boucle
if current_city:
    print(f"{current_city}\t{total_sales}")

```

5-3- Exécution du Job.

Après la rédaction des programmes mapper et reducer, nous allons lancer le job mapreduce en utilisant les codes suivants :

```

root@hadoop-master:~# hadoop jar /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-*.jar \
> -input input/purchases.txt \
> -output /output/ventes_villes \
> -mapper "python3 mapper1.py" \
> -reducer "python3 reducer1.py" \
> -file /home/mapper1.py \
> -file /home/reducer1.py

```

Ce code exécute un job Hadoop MapReduce en utilisant le package hadoop-streaming pour traiter un fichier purchases.txt avec un mapper et un reducer écrits en Python. Voici un détail de chaque partie de la commande :

- ***hadoop jar /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-*.jar*** : Cette commande lance le job Hadoop en utilisant le fichier JAR hadoop-streaming, qui permet d'utiliser des scripts externes (comme Python) pour le traitement MapReduce dans Hadoop.

- **input input/purchases.txt** : Cette option spécifie le chemin du fichier d'entrée dans HDFS, purchases.txt, qui contient les données à traiter.
- **output /output/ventes_villes** : Cette option définit le répertoire de sortie dans HDFS où les résultats seront sauvegardés. Si ce répertoire existe déjà, une erreur sera générée.
- **mapper "python3 mapper3.py"** : Cette option indique que le fichier mapper3.py (un script Python) sera utilisé pour la phase de map du job. Le mapper lit chaque ligne du fichier d'entrée, extrait les informations pertinentes (par exemple, la ville et le montant), et les émet sous forme de paires clé-valeur.
- **reducer "python3 reducer3.py"** : Cette option indique que le fichier reducer3.py (un autre script Python) sera utilisé pour la phase de reduce du job. Le reducer reçoit les paires clé-valeur émises par le mapper, les regroupe par clé (ville) et agrège les valeurs associées (par exemple, en faisant la somme des montants des ventes pour chaque ville).
- **file /home/mapper3.py et -file /home/reducer3.py** : Ces options spécifient les fichiers Python mapper3.py et reducer3.py, qui sont envoyés au cluster Hadoop pour être utilisés dans les phases mapper et reducer respectivement. Cela garantit que ces scripts Python sont accessibles pendant l'exécution du job.

5-4- Résultats :

5-4-1 Résultats de la première question :

Total des ventes par ville : Combien d'argent a été dépensé dans chaque ville ?

Après l'exécution du Job, voici un aperçu des résultats obtenus :

```
Albuquerque 5524244.309999986
Anaheim 5639296.700000043
Anchorage 5554922.3600000255
Arlington 5571053.310000012
Atlanta 5573329.290000018
Aurora 5524579.360000012
Austin 5621596.450000031
Bakersfield 5573561.269999996
Baltimore 5572059.289999976
Birmingham 5582305.480000028
Boise 5586446.409999998
Boston 5540718.989999976
Buffalo 5627585.600000015
Chandler 5490041.720000012
Charlotte 5668650.659999997
Chesapeake 5566271.600000004
Chicago 5564669.029999992
Cincinnati 5651683.529999974
Cleveland 5588794.359999961
Columbus 5560598.519999983
```

Ces résultats fournissent, pour chaque ville, le montant total des ventes. Nous pouvons exporter ces données et effectuer des analyses plus approfondies avec Python, par exemple. En utilisant Hadoop, nous avons pu traiter efficacement une base de données contenant plusieurs millions de lignes, en la réduisant à un ensemble beaucoup plus gérable, composé de moins de 30 lignes, correspondant à une ligne par ville. Grâce à la scalabilité et au traitement parallèle de Hadoop, nous avons pu gérer de grandes quantités de données de manière distribuée et rapide, ce qui rend l'analyse de données massives plus accessible et efficace.

5-4-2 Résultats de la deuxième question :

Total des ventes par catégorie de produit : Quel est le total des ventes pour chaque catégorie ?

Les résultats sont :

```
Ventes totales par catégorie :
root@hadoop-master:~# cat /root/purchases.txt | /home/mapper2.py | sort | /home/reducer2.py
Baby      43069097.199999526
Books     43035664.850000314
CDs       43029256.54000055
Cameras   42797762.740000784
Computers 42896561.850000724
Crafts    43018914.009999774
DVDs      43131865.889999375
Garden    42962977.78000005
Music     43087532.41000054
Toys      42884032.44999961
root@hadoop-master:~#
```

5-4-3 Résultats de la troisième question :

Total des ventes par méthode de paiement : Quelles sont les méthodes de paiement les plus utilisées ?

Les analyses donnent :

```
root@hadoop-master:~# echo "total des ventes par type de paiement :"
total des ventes par type de paiement :
root@hadoop-master:~# cat /root/purchases.txt | /home/mapper3.py | sort | /home/reducer3.py
Amex      85951391.07999876
Cash      85936417.65999848
Discover  85950229.74999972
MasterCard 86089744.01999825
Visa      85985883.21000075
root@hadoop-master:~#
```

L'analyse des ventes totales par mode de paiement montre que MasterCard est le moyen le plus utilisé, avec un total de 86 089 744,02. Les autres modes de paiement, comme Visa (85 985 883,21) et Amex (85 951 391,08), enregistrent des montants légèrement inférieurs mais restent proches. Discover (85 950 229,75) et Cash (85 936 417,66) affichent les volumes les plus bas.

5-4-4 Résultats de la quatrième question :

Ventes moyennes par transaction : Quelle est la dépense moyenne par transaction ?

Il s'agit ici de calculer la moyenne des ventes.

```
Résultats de la vente moyenne :  
root@hadoop-master:~# cat /root/purchases.txt | /home/mapper4.py | sort | /home/  
reducer4.py  
vente moyenne par transaction : 249.95  
root@hadoop-master:~#
```

La vente moyenne est de 249,95

5-4-5 Résultats de la cinquième question :

Analyse de la répartition temporelle : sur quelle période de la journée on note plus de ventes ?

```
Résultats de la série temporelle :  
root@hadoop-master:~# cat /root/purchases.txt | /home/mapper5.py | sort | /home/  
reducer5.py  
09:00 - 09:59 47565539.67  
10:00 - 10:59 47719578.59  
11:00 - 11:59 47675739.48  
12:00 - 12:59 47752364.15  
13:00 - 13:59 47875997.61  
14:00 - 14:59 47690445.90  
15:00 - 15:59 47835445.37  
16:00 - 16:59 47942690.79  
17:00 - 17:59 47855864.16  
root@hadoop-master:~#
```

Ces résultats montre que la tranche horaire où les ventes sont les plus élevées est **16:00 - 16:59**, avec un total de **47 942 690,79**.