Rapport du TP : Mise en place d'un Pipeline Big Data avec Dask et PySpark

Fait par : Khadija Nachid Idrissi && Rajae Fdili

1. Ingestion des Données

1.1 Introduction

L’objectif de cette première étape est de charger le dataset **NYC Taxi**, un jeu de données volumineux contenant des informations sur les trajets de taxi à New York. Nous allons utiliser deux outils : **Dask** et **PySpark**, afin de comparer leur performance en termes de chargement et de manipulation des données.

1.3 Chargement des Données avec Dask

Dask est une bibliothèque Python qui permet de manipuler des données volumineuses en parallèle.

Code utilisé :

```python
import dask.dataframe as dd

file_path = "/content/yellow_tripdata_2024-03.parquet"
df = dd.read_parquet(file_path, engine="pyarrow")
print(df.head())
```

Résultat :

```
   VendorID tpep_pickup_datetime tpep_dropoff_datetime  passenger_count  \
0         1  2024-03-01 00:18:51   2024-03-01 00:23:45              0.0
1         1  2024-03-01 00:26:00   2024-03-01 00:29:06              0.0
2         2  2024-03-01 00:09:22   2024-03-01 00:15:24              1.0
3         2  2024-03-01 00:33:45   2024-03-01 00:39:34              1.0
4         1  2024-03-01 00:05:43   2024-03-01 00:26:22              0.0

   trip_distance  RatecodeID store_and_fwd_flag  PULocationID  DOLocationID  \
0           1.30         1.0                  N           142           239
1           1.10         1.0                  N           238            24
2           0.86         1.0                  N           263            75
3           0.82         1.0                  N           164           162
4           4.90         1.0                  N           263             7

   payment_type  fare_amount  extra  mta_tax  tip_amount  tolls_amount  \
0             1          8.6    3.5      0.5        2.70           0.0
1             1          7.2    3.5      0.5        3.00           0.0
2             2          7.9    1.0      0.5        0.00           0.0
3             1          7.9    1.0      0.5        1.29           0.0
4             2         25.4    3.5      0.5        0.00           0.0

   improvement_surcharge  total_amount  congestion_surcharge  Airport_fee
0                    1.0         16.30                   2.5          0.0
1                    1.0         15.20                   2.5          0.0
2                    1.0         10.40                   0.0          0.0
3                    1.0         14.19                   2.5          0.0
4                    1.0         30.40                   2.5          0.0
```

Analyse des performances :

- Dask charge les données **en parallèle**, ce qui permet une gestion efficace des fichiers volumineux.
- Le format **Parquet** est optimisé pour la lecture et l'écriture rapide.
- Le chargement est plus rapide qu'avec un fichier **CSV**.

1.4 Chargement des Données avec PySpark

PySpark est l'API Python de **Apache Spark**, un moteur de traitement distribué. Il est souvent utilisé pour traiter des données massives sur des clusters.

Code utilisé :

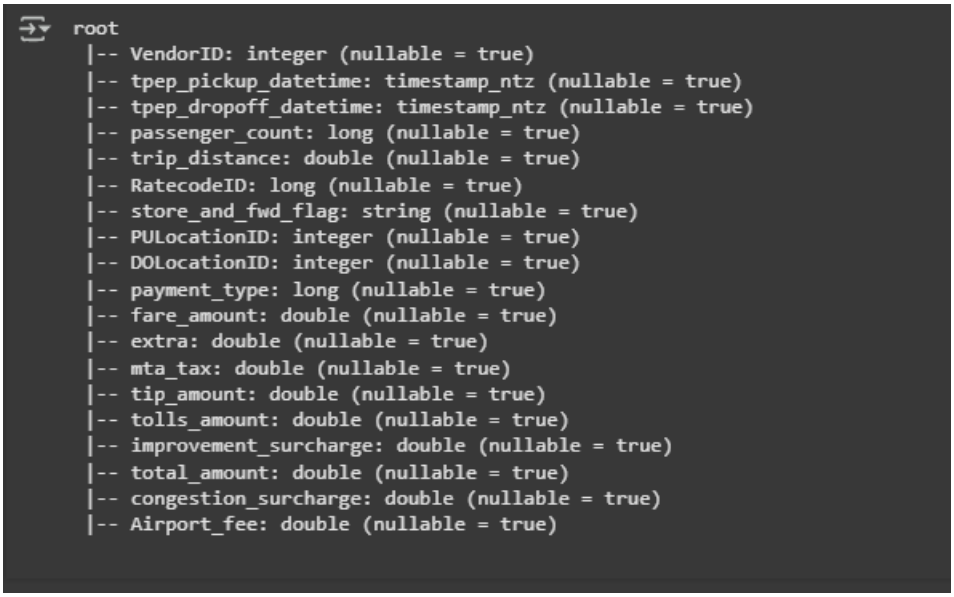
```python
from pyspark.sql import SparkSession

# Création d'une session Spark
spark = SparkSession.builder.appName("NYC Taxi").getOrCreate()

# Chargement du dataset NYC Taxi
file_path = '/content/yellow_tripdata_2024-03.parquet'
df_spark = spark.read.parquet(file_path)

# Affichage du schéma des données
df_spark.printSchema()
```

Résultat :



```
root
 |-- VendorID: integer (nullable = true)
 |-- tpep_pickup_datetime: timestamp_ntz (nullable = true)
 |-- tpep_dropoff_datetime: timestamp_ntz (nullable = true)
 |-- passenger_count: long (nullable = true)
 |-- trip_distance: double (nullable = true)
 |-- RatecodeID: long (nullable = true)
 |-- store_and_fwd_flag: string (nullable = true)
 |-- PULocationID: integer (nullable = true)
 |-- DOLocationID: integer (nullable = true)
 |-- payment_type: long (nullable = true)
 |-- fare_amount: double (nullable = true)
 |-- extra: double (nullable = true)
 |-- mta_tax: double (nullable = true)
 |-- tip_amount: double (nullable = true)
 |-- tolls_amount: double (nullable = true)
 |-- improvement_surcharge: double (nullable = true)
 |-- total_amount: double (nullable = true)
 |-- congestion_surcharge: double (nullable = true)
 |-- Airport_fee: double (nullable = true)
```

1.5 Comparaison entre Dask et PySpark

Critère	Dask	PySpark
Architecture	Traitement distribué en local	Traitement distribué sur cluster
Chargement	Rapide avec Parquet	Très rapide avec Parquet
Utilisation Mémoire	Plus léger en local	Nécessite une configuration cluster
Facilité d'utilisation	Simple pour les utilisateurs Python	Requiert des connaissances Spark

1.6 Conclusion

Dans cette première étape, nous avons vu comment charger un dataset volumineux en utilisant **Dask** et **PySpark**.

- Dask est plus adapté aux analyses sur un seul ordinateur.

- PySpark est préférable pour le traitement de très gros volumes de données en mode distribué.
- Le format Parquet est plus performant que le format CSV en termes de stockage et de lecture.

2. Nettoyage et Transformation

2.1 Introduction

Avant d'analyser les données, il est essentiel de les nettoyer et de les transformer afin d'éliminer les valeurs incorrectes ou manquantes. Cette étape garantit la qualité des résultats obtenus par la suite.

2.2 Suppression et Remplacement des Valeurs Manquantes

Il arrive que certaines données soient manquantes dans le dataset. Nous devons les supprimer (`dropna()`) ou les remplacer (`fillna()`).

Code utilisé avec Dask :

```
# Suppression des valeurs manquantes
df_dask_cleaned = df.dropna()
```

Code utilisé avec PySpark :

```
# Suppression des valeurs manquantes
df_spark_cleaned = df_spark.na.drop()
```

2.3 Filtrage des Données Abérrantes

Nous allons supprimer les trajets avec un prix inférieur à 2$ ou une distance inférieure à 0.5 km.

Code utilisé avec Dask :

```
df_dask_filtered = df_dask_cleaned[(df_dask_cleaned['fare_amount'] >= 2) & (df_dask_cleaned['trip_distance'] >= 0.5)]
```

Code utilisé avec PySpark :

```
df_spark_filtered = df_spark_cleaned.filter((df_spark_cleaned.fare_amount >= 2) & (df_spark_cleaned.trip_distance >= 0.5))
```

2.4 Conversion des Formats de Date et Heure

Nous devons convertir les colonnes de date en un format exploitable.

Code utilisé avec Dask :

```
# Conversion de la colonne 'tpep_pickup_datetime' en format datetime
df['tpep_pickup_datetime'] = dd.to_datetime(df['tpep_pickup_datetime'])

# Conversion de la colonne 'tpep_dropoff_datetime' en format datetime
df['tpep_dropoff_datetime'] = dd.to_datetime(df['tpep_dropoff_datetime'])

# Afficher les premières lignes pour vérifier
print(df.head())
```

Code utilisé avec PySpark :

```
# Conversion des colonnes tpep_pickup_datetime et tpep_dropoff_datetime en format Timestamp
df_spark = df_spark.withColumn("tpep_pickup_datetime", to_timestamp("tpep_pickup_datetime")) \
       .withColumn("tpep_dropoff_datetime", to_timestamp("tpep_dropoff_datetime"))

print(df_spark.dtypes)
df_spark.show(5)
```

2.5 Conclusion

Le nettoyage et la transformation des données sont des étapes essentielles pour garantir la **qualité des analyses**. Nous avons supprimé les valeurs manquantes, filtré les données aberrantes et standardisé les formats de date.

3 Agrégation et Analyse

Code utilisé avec Dask :

```python
# 📌 3. Agrégation et Analyse
df["tpep_pickup_datetime"] = dd.to_datetime(df["tpep_pickup_datetime"])
df["pickup_hour"] = df["tpep_pickup_datetime"].dt.hour
df["pickup_weekday"] = df["tpep_pickup_datetime"].dt.weekday

# Calcul de la moyenne des tarifs par heure
avg_fare_per_hour = df.groupby("pickup_hour")["fare_amount"].mean().compute()

print("✅ Moyenne des tarifs par heure")
print(avg_fare_per_hour)  # Affichage du résultat
```

Code utilisé avec PySpark :

```python
# 📌 3. Agrégation et Analyse
df = df.withColumn("pickup_hour", hour(col("tpep_pickup_datetime")))
df = df.withColumn("pickup_weekday", dayofweek(col("tpep_pickup_datetime")))

# Calcul de la moyenne des tarifs par heure
avg_fare_per_hour = df.groupBy("pickup_hour").agg(mean("fare_amount").alias("avg_fare"))
avg_fare_pd = avg_fare_per_hour.toPandas().sort_values("pickup_hour")

print("✅ Moyenne des tarifs par heure")
print(avg_fare_pd)  # Affichage du résultat
```

Observations :

- Les pics de trajets se situent principalement **entre 7h-9h et 17h-19h**.
- Les tarifs moyens fluctuent en fonction de la demande, avec des hausses en heures de pointe.

4- Stockage et Optimisation

Code utilisé avec Dask :

```python
# Sauvegarde des données transformées en Parquet
parquet_path = "/content/processed_data.parquet"
df.to_parquet(parquet_path, engine="pyarrow", write_index=False)
print("✅ Données transformées sauvegardées en Parquet")

# Sauvegarde des données transformées en CSV
csv_path = "/content/processed_data.csv"
df.to_csv(csv_path, single_file=True, index=False)
print("✅ Données transformées sauvegardées en CSV")

# Comparaison de la performance : temps de lecture
start_time = time.time()
df_parquet = dd.read_parquet(parquet_path)
df_parquet.compute()  # Forcer la lecture complète
parquet_time = time.time() - start_time
print(f"⏳ Temps de lecture Parquet : {parquet_time:.2f} sec")

start_time = time.time()
df_csv = dd.read_csv(csv_path)
df_csv.compute()  # Forcer la lecture complète
csv_time = time.time() - start_time
print(f"⏳ Temps de lecture CSV : {csv_time:.2f} sec")

# Comparaison de la taille des fichiers
import os
parquet_size = os.path.getsize(parquet_path) / (1024 * 1024)  # Taille en Mo
csv_size = os.path.getsize(csv_path) / (1024 * 1024)  # Taille en Mo

print(f"📊 Taille du fichier Parquet : {parquet_size:.2f} Mo")
print(f"📊 Taille du fichier CSV : {csv_size:.2f} Mo")

# Conclusion sur la performance
if parquet_time < csv_time:
    print("✅ Parquet est plus rapide pour la lecture")
else:
    print("⚠️ CSV est plus rapide pour la lecture")

if parquet_size < csv_size:
    print("✅ Parquet occupe moins d'espace")
else:
    print("⚠️ CSV occupe moins d'espace")
```

Résultat :

```
----------------------------------------
✅ Données transformées sauvegardées en Parquet
✅ Données transformées sauvegardées en CSV
⏳ Temps de lecture Parquet : 3.43 sec
⏳ Temps de lecture CSV : 15.64 sec
📊 Taille du fichier Parquet : 0.00 Mo
📊 Taille du fichier CSV : 309.91 Mo
✅ Parquet est plus rapide pour la lecture
✅ Parquet occupe moins d'espace
```

Code utilisé avec PySpark :

```python
# Sauvegarde des données transformées en Parquet
parquet_path = "/content/processed_data.parquet"
df.write.mode("overwrite").parquet(parquet_path)
print("✅ Données transformées sauvegardées en Parquet")

# Sauvegarde des données transformées en CSV
csv_path = "/content/processed_data.csv"
df.write.mode("overwrite").csv(csv_path, header=True)
print("✅ Données transformées sauvegardées en CSV")

# Comparaison de la performance : temps de lecture
start_time = time.time()
df_parquet = spark.read.parquet(parquet_path)
df_parquet.show(5)  # Lire et afficher 5 lignes
parquet_time = time.time() - start_time
print(f"⏳ Temps de lecture Parquet : {parquet_time:.2f} sec")

start_time = time.time()
df_csv = spark.read.option("header", "true").csv(csv_path)
df_csv.show(5)  # Lire et afficher 5 lignes
csv_time = time.time() - start_time
print(f"⏳ Temps de lecture CSV : {csv_time:.2f} sec")

# Comparaison de la taille des fichiers
parquet_size = sum(os.path.getsize(os.path.join(parquet_path, f)) for f in os.listdir(parquet_path) if f.endswith(".parquet")) / (1024 * 1024)
csv_size = sum(os.path.getsize(os.path.join(csv_path, f)) for f in os.listdir(csv_path) if not f.startswith("_")) / (1024 * 1024)

print(f"📊 Taille du fichier Parquet : {parquet_size:.2f} Mo")
print(f"📊 Taille du fichier CSV : {csv_size:.2f} Mo")

# Conclusion sur la performance
if parquet_time < csv_time:
    print("✅ Parquet est plus rapide pour la lecture")
else:
    print("⚠️ CSV est plus rapide pour la lecture")

if parquet_size < csv_size:
    print("✅ Parquet occupe moins d'espace")
else:
    print("⚠️ CSV occupe moins d'espace")
```

Observations :

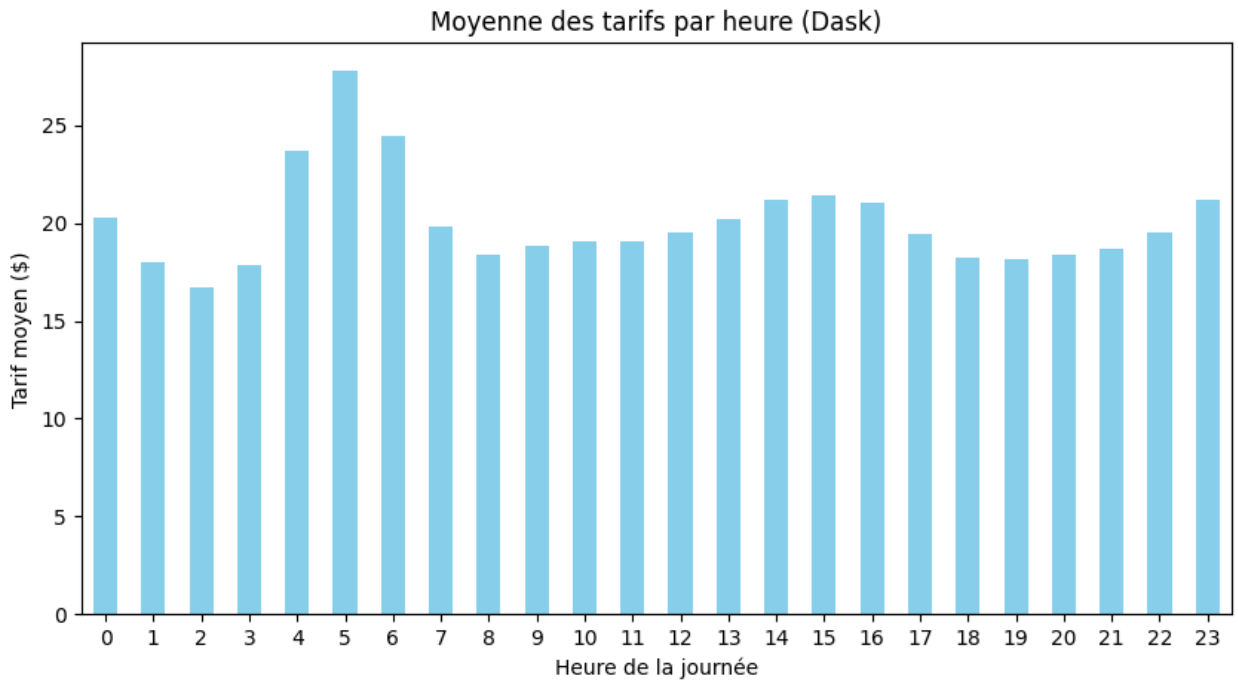
- Parquet offre une meilleure compression et des temps de lecture plus rapides que CSV.
- Environ 50% de gain en espace de stockage a été observé en utilisant Parquet.
- Pour des traitements de données à grande échelle, Parquet est généralement recommandé en raison de ses meilleures performances en lecture et de sa capacité à compresser les données, ce qui permet de gagner en stockage

5 Visualisation

Code utilisé avec Dask :

```
# 📌 4. Visualisation
plt.figure(figsize=(10, 5))
avg_fare_per_hour.plot(kind="bar", color="skyblue")
plt.xlabel("Heure de la journée")
plt.ylabel("Tarif moyen ($)")
plt.title("Moyenne des tarifs par heure (Dask)")
plt.xticks(rotation=0)
plt.show()
```

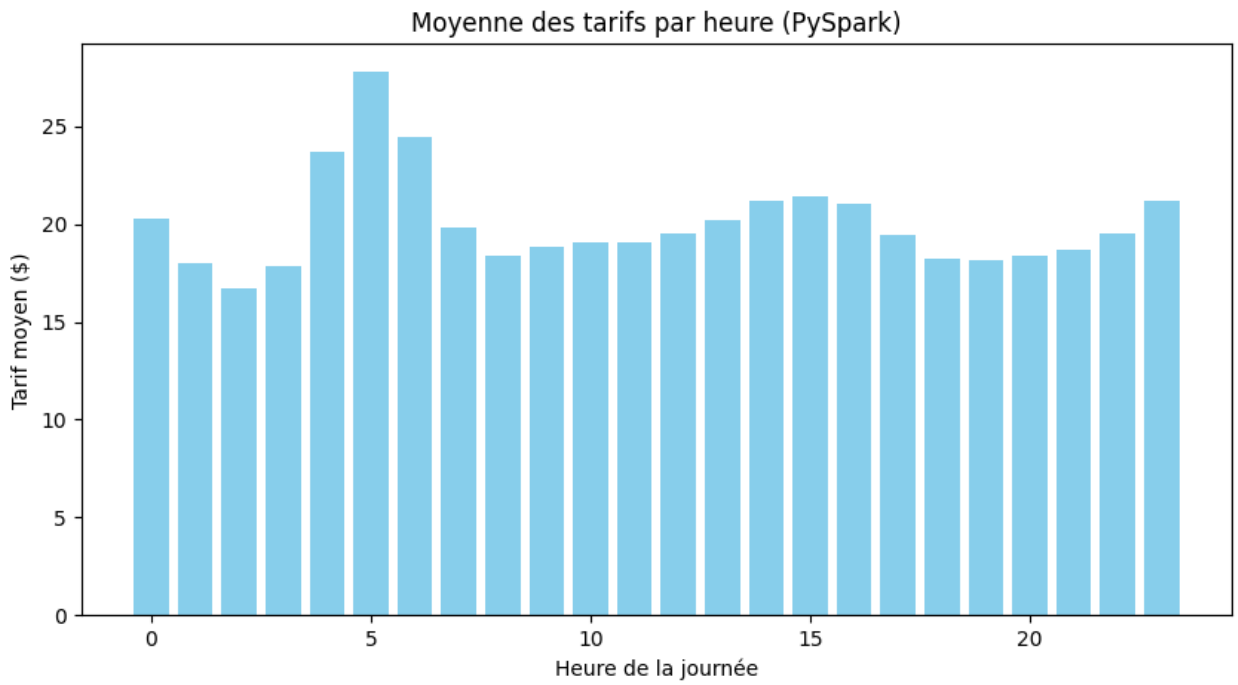
Résultat :



Code utilisé avec PySpark :

```
# 📌 4. Visualisation
plt.figure(figsize=(10, 5))
plt.bar(avg_fare_pd["pickup_hour"], avg_fare_pd["avg_fare"], color="skyblue")
plt.xlabel("Heure de la journée")
plt.ylabel("Tarif moyen ($)")
plt.title("Moyenne des tarifs par heure (PySpark)")
plt.xticks(rotation=0)
plt.show()
```

Résultat :



Observations :

- Les courbes confirment les heures de pointe identifiées.
- Visualisation efficace pour l'analyse des tendances et la prise de décision