# Rapport de TP 5 - Spark avec Python (PySpark)


## Réalisé par : LAABID ABDESSAMAD

## github: https://github.com/aplusInDev/hadoop_tps


# Introduction à PySpark

PySpark est une interface Python pour Apache Spark, un framework de traitement distribué conçu pour le traitement de grandes quantités de données. Cette bibliothèque permet d'exécuter des applications Python en exploitant les capacités de traitement parallèle et distribué d'Apache Spark.

PySpark offre plusieurs avantages :

- Traitement de données à grande échelle
- Exécution parallèle sur des clusters distribués
- API intuitive en Python pour les opérations de données
- Intégration avec l'écosystème Python pour l'analyse de données et le machine learning

L'intégration entre Python et Spark est réalisée grâce à la bibliothèque Py4J, qui permet à Python d'interagir dynamiquement avec les objets Java de la JVM (Java Virtual Machine).

# Architecture de PySpark

PySpark fonctionne sur une architecture maître-esclave :

- **Driver (Maître)** : C'est le processus principal qui coordonne l'exécution des applications Spark. Il crée le SparkContext qui sert de point d'entrée à l'application.
- **Workers (Esclaves)** : Ce sont les nœuds où s'exécutent les opérations de traitement des données.
- **Cluster Manager** : Il gère l'allocation des ressources entre les différentes applications Spark.

Le flux de travail général est le suivant :

1. L'application PySpark crée un SparkSession/SparkContext
2. Le Driver planifie les tâches et les distribue aux Workers
3. Les Workers exécutent les opérations sur les données
4. Les résultats sont retournés au Driver

L'écosystème PySpark comprend plusieurs modules spécialisés :

- RDD (Resilient Distributed Dataset)
- DataFrame & SQL
- Streaming
- MLlib pour le machine learning
- GraphFrames pour le traitement de graphes

# Installation et configuration

Pour utiliser PySpark, les composants suivants doivent être installés :

1. **Java 8 ou ultérieur** : Nécessaire pour exécuter Spark
2. **Python** : Le langage de programmation utilisé
3. **Apache Spark** : Le moteur de traitement distribué
4. **Distribution Anaconda** : Recommandée pour l'environnement de développement Python

La procédure d'installation est la suivante :

```
# Installation de PySpark via conda
conda install -c conda-forge pyspark
```

Configuration nécessaire :

- Définir la variable d'environnement PYTHONPATH :

```
%SPARK_HOME%/python;%SPARK_HOME%/python/lib/py4j-0.10.9-src.zip;%PYTHONPATH%
```

# Concepts fondamentaux de PySpark

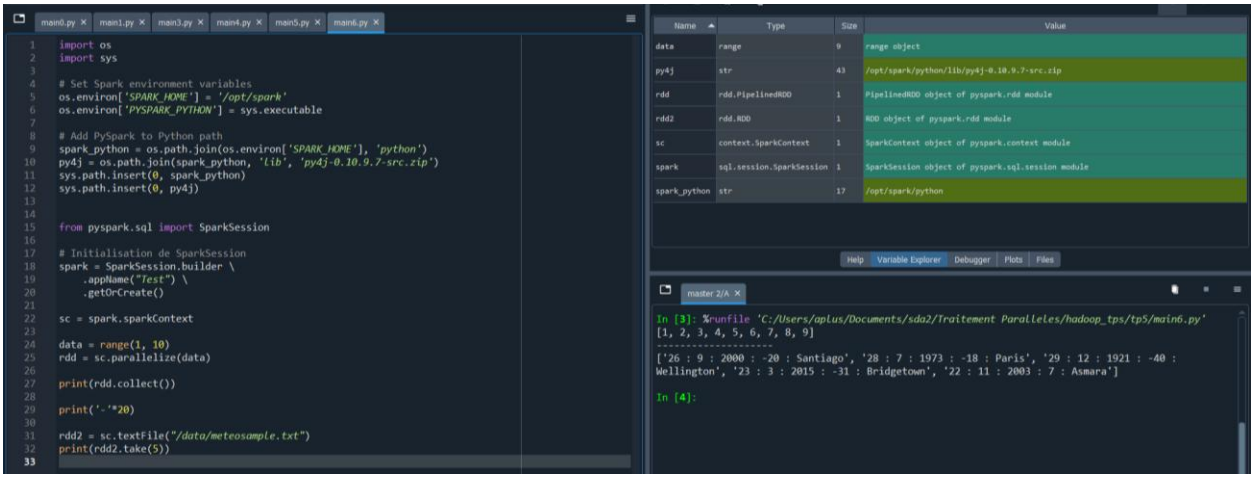
## RDD (Resilient Distributed Dataset)

Le RDD est la structure de données fondamentale de PySpark. C'est une collection distribuée, immuable et tolérante aux pannes d'objets. Les caractéristiques principales des RDD sont :

- **Immutabilité** : Une fois créé, un RDD ne peut pas être modifié
- **Distribution** : Les données sont partitionnées sur les nœuds du cluster
- **Résilience** : Capacité à reconstruire les données en cas de panne d'un nœud

Création d'un RDD :

```
# Création d'un RDD par parallélisation
data = range(1, 10)
rdd = spark.sparkContext.parallelize(data)

# Création d'un RDD à partir d'un fichier
rdd = spark.sparkContext.textFile("/data/meteosample.txt")
```




## DataFrame

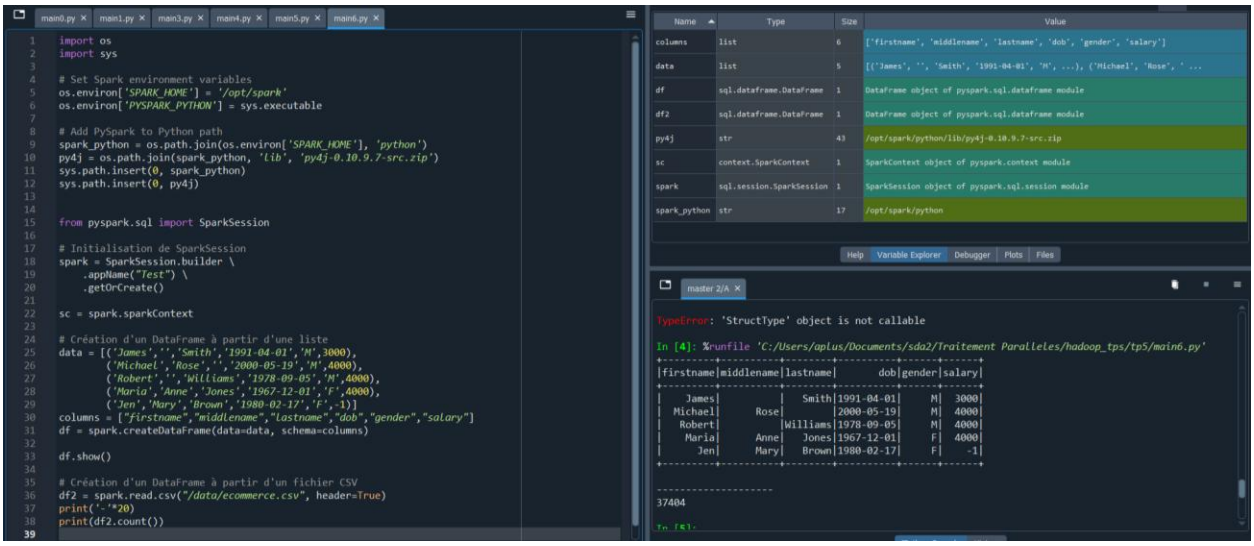
Le DataFrame est une abstraction de plus haut niveau que le RDD. C'est une collection distribuée de données organisées en colonnes nommées, similaire à une table dans une base de données relationnelle ou à un DataFrame dans pandas.

Avantages des DataFrames :

- Structure des données plus explicite
- Optimisations de requêtes via le moteur Catalyst
- API plus intuitive et plus proche de SQL

Exemple de création d'un DataFrame :

```
# Création d'un DataFrame à partir d'une liste
data = [('James','','Smith','1991-04-01','M',3000),
        ('Michael','Rose','','2000-05-19','M',4000),
        ('Robert','','Williams','1978-09-05','M',4000),
        ('Maria','Anne','Jones','1967-12-01','F',4000),
        ('Jen','Mary','Brown','1980-02-17','F',-1)]
columns = ["firstname","middlename","lastname","dob","gender","salary"]
df = spark.createDataFrame(data=data, schema=columns)

# Création d'un DataFrame à partir d'un fichier CSV
df = spark.read.csv("/chemin/vers/fichier.csv", header=True,
inferSchema=True)
```




## PySpark SQL

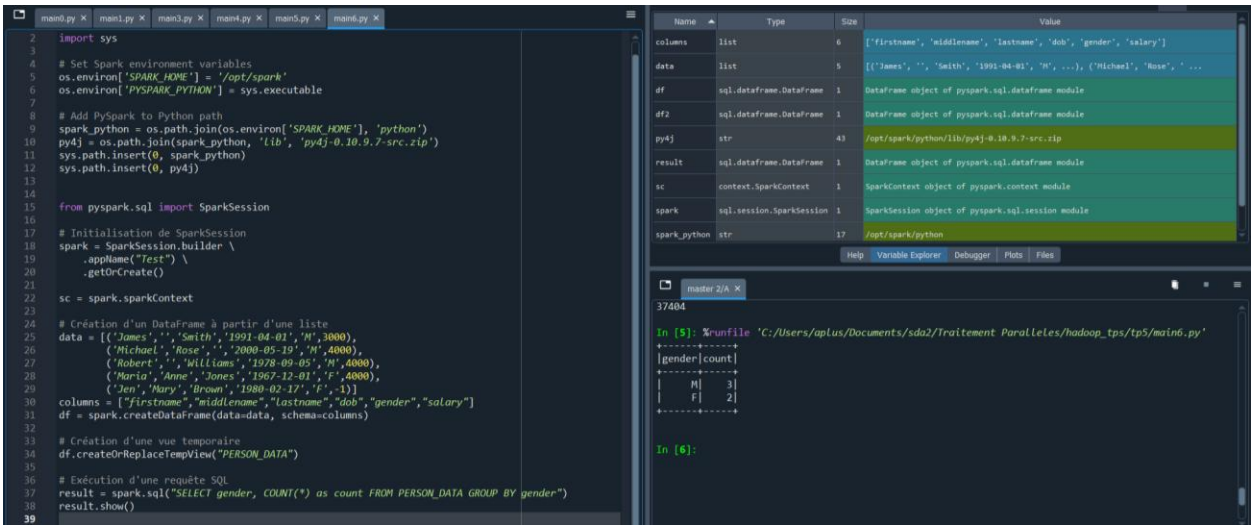
PySpark SQL permet d'exécuter des requêtes SQL sur les données stockées dans les DataFrames. Ce module facilite l'interaction avec les données structurées en utilisant une syntaxe familière.

Fonctionnalités principales :

- Exécution de requêtes SQL sur les DataFrames
- Création de vues temporaires pour les requêtes
- Intégration avec les sources de données externes

Exemple d'utilisation :

```
# Création d'une vue temporaire
df.createOrReplaceTempView("PERSON_DATA")

# Exécution d'une requête SQL
result = spark.sql("SELECT gender, COUNT(*) as count FROM PERSON_DATA GROUP
BY gender")
result.show()
```




# Analyse de données avec PySpark

## Présentation du jeu de données

Le jeu de données utilisé dans ce TP provient d'une base de données de boutique et se compose de trois tables :

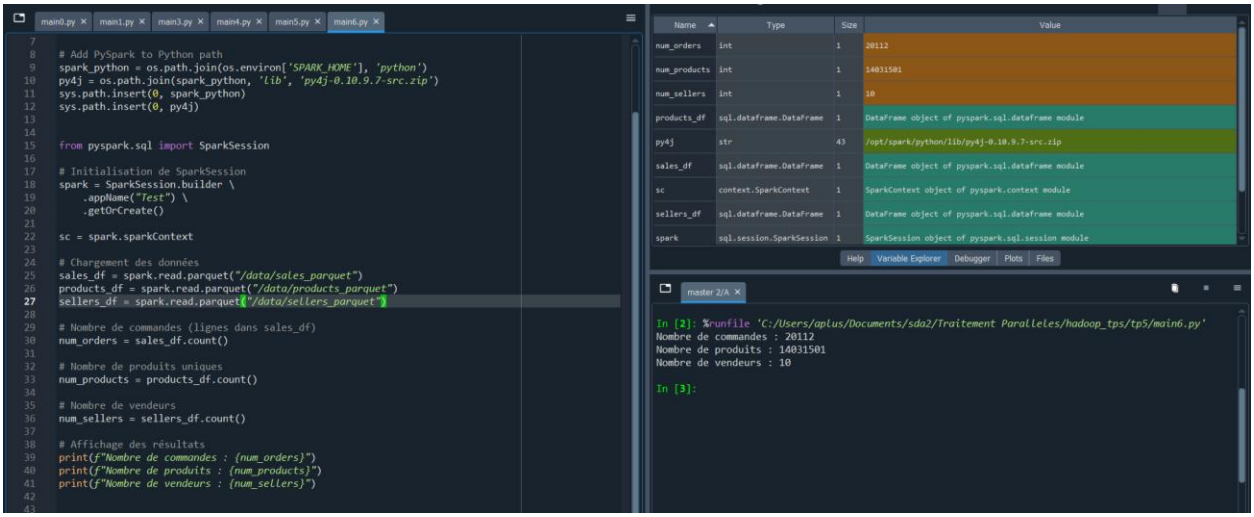
1. **Table Sales (Ventes)** :
   - `order_id` : Identifiant de la commande
   - `product_id` : Produit vendu (un seul par commande)
   - `seller_id` : Identifiant du vendeur
   - `num_pieces_sold` : Nombre d'unités vendues
   - `bill_raw_text` : Texte brut de la facture
   - `date` : Date de la commande
2. **Table Products (Produits)** :
   - Contient les informations sur les produits disponibles
   - Inclut `product_id` et `price`
3. **Table Sellers (Vendeurs)** :
   - `seller_id` : Identifiant du vendeur
   - `seller_name` : Nom du vendeur
   - `daily_target` : Quota quotidien en nombre d'articles

## Question 1 : Analyse des volumes de données

Pour compter le nombre de commandes, de produits et de vendeurs dans les données, nous utilisons le code suivant :

```
# Chargement des données
sales_df = spark.read.parquet("/data/sales_parquet")
products_df = spark.read.parquet("/data/products_parquet")
sellers_df = spark.read.parquet("/data/sellers_parquet")
```

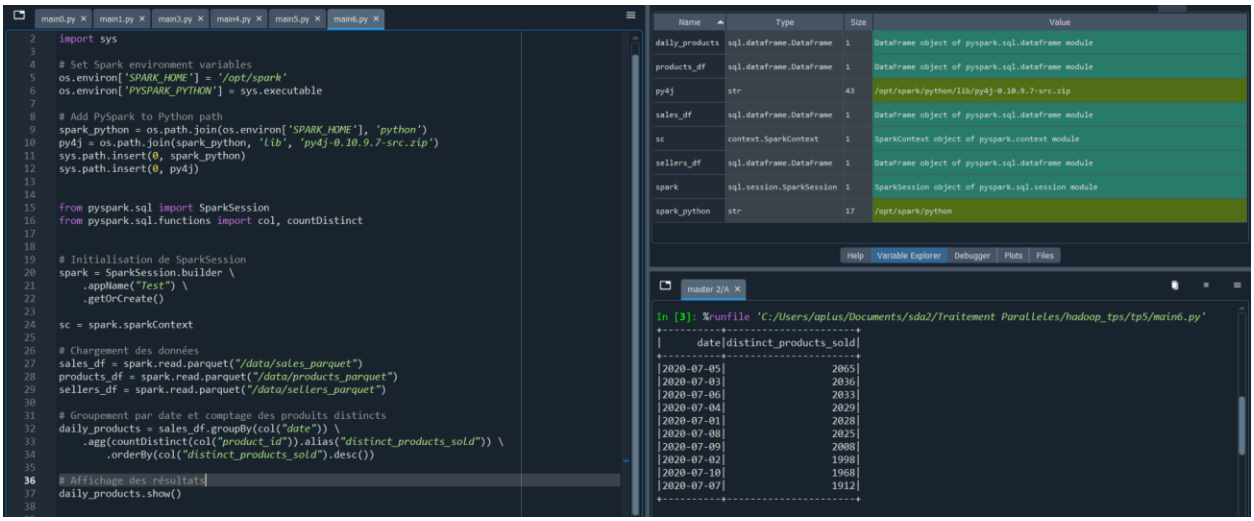
```
# Nombre de commandes (lignes dans sales_df)
num_orders = sales_df.count()

# Nombre de produits uniques
num_products = products_df.count()

# Nombre de vendeurs
num_sellers = sellers_df.count()

# Affichage des résultats
print(f"Nombre de commandes : {num_orders}")
print(f"Nombre de produits : {num_products}")
print(f"Nombre de vendeurs : {num_sellers}")
```




## Question 2 : Analyse des produits vendus par jour

Pour déterminer le nombre de produits distincts vendus chaque jour, nous utilisons l'agrégation avec la fonction `countDistinct` :

```
from pyspark.sql.functions import col, countDistinct

# Groupement par date et comptage des produits distincts
daily_products = sales_df.groupBy(col("date")) \

.agg(countDistinct(col("product_id")).alias("distinct_products_sold")) \
                          .orderBy(col("distinct_products_sold").desc())

# Affichage des résultats
daily_products.show()
```

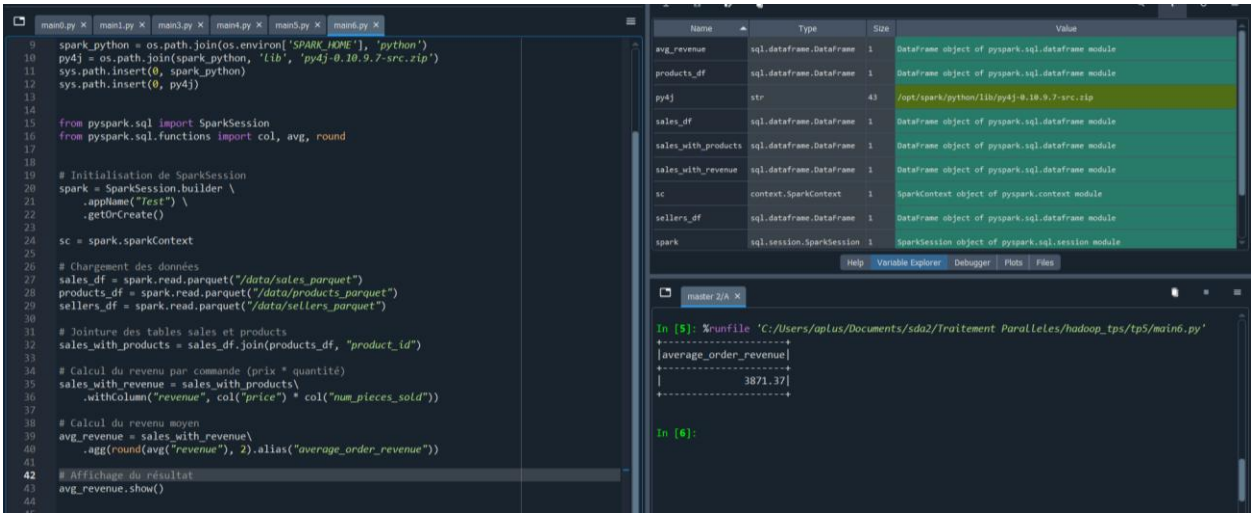



# Travail demandé

## Question 1 : Revenu moyen des commandes

Pour calculer le revenu moyen des commandes, nous devons joindre les tables des ventes et des produits afin d'obtenir le prix unitaire de chaque produit vendu, puis calculer le revenu total par commande.

```
from pyspark.sql.functions import col, avg, round

# Jointure des tables sales et products
sales_with_products = sales_df.join(products_df, "product_id")

# Calcul du revenu par commande (prix * quantité)
sales_with_revenue = sales_with_products.withColumn("revenue", col("price") *
col("num_pieces_sold"))

# Calcul du revenu moyen
avg_revenue = sales_with_revenue.agg(round(avg("revenue"),
2).alias("average_order_revenue"))

# Affichage du résultat
avg_revenue.show()
```

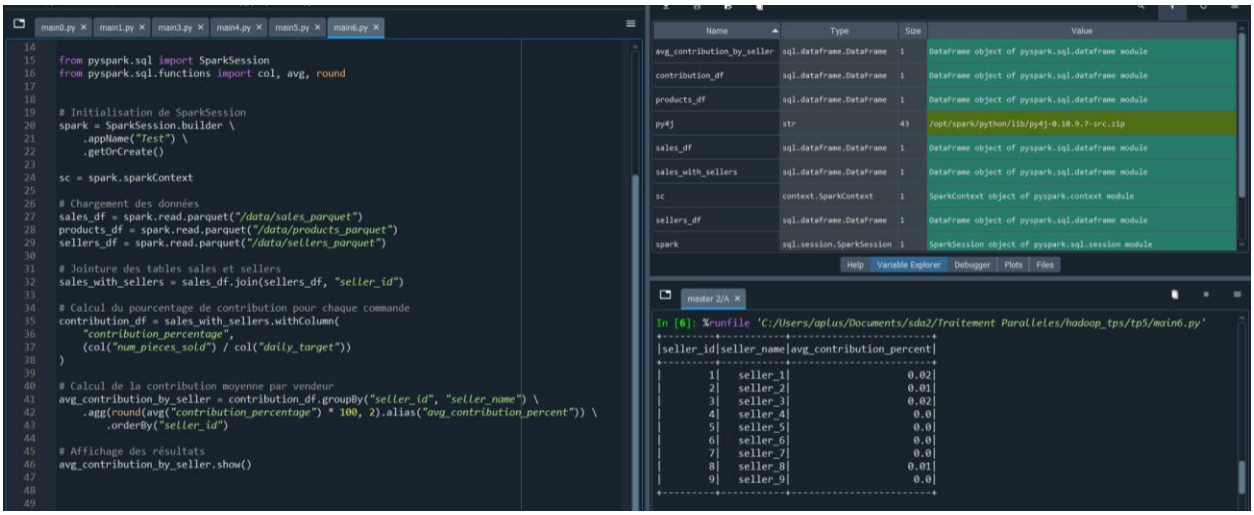



## Question 2 : Contribution moyenne aux quotas des vendeurs

Pour calculer la contribution moyenne en pourcentage de chaque commande au quota quotidien du vendeur, nous devons :

1. Joindre les tables des ventes et des vendeurs
2. Calculer le pourcentage de contribution de chaque commande au quota du vendeur
3. Calculer la moyenne de ces pourcentages pour chaque vendeur

```
from pyspark.sql.functions import col, avg, round

# Jointure des tables sales et sellers
sales_with_sellers = sales_df.join(sellers_df, "seller_id")

# Calcul du pourcentage de contribution pour chaque commande
contribution_df = sales_with_sellers.withColumn(
    "contribution_percentage",
    (col("num_pieces_sold") / col("daily_target"))
)

# Calcul de la contribution moyenne par vendeur
avg_contribution_by_seller = contribution_df.groupBy("seller_id",
"seller_name") \
    .agg(round(avg("contribution_percentage") * 100,
2).alias("avg_contribution_percent")) \
    .orderBy("seller_id")

# Affichage des résultats
avg_contribution_by_seller.show()
```




## Question 3 : Meilleur et pire vendeur par produit

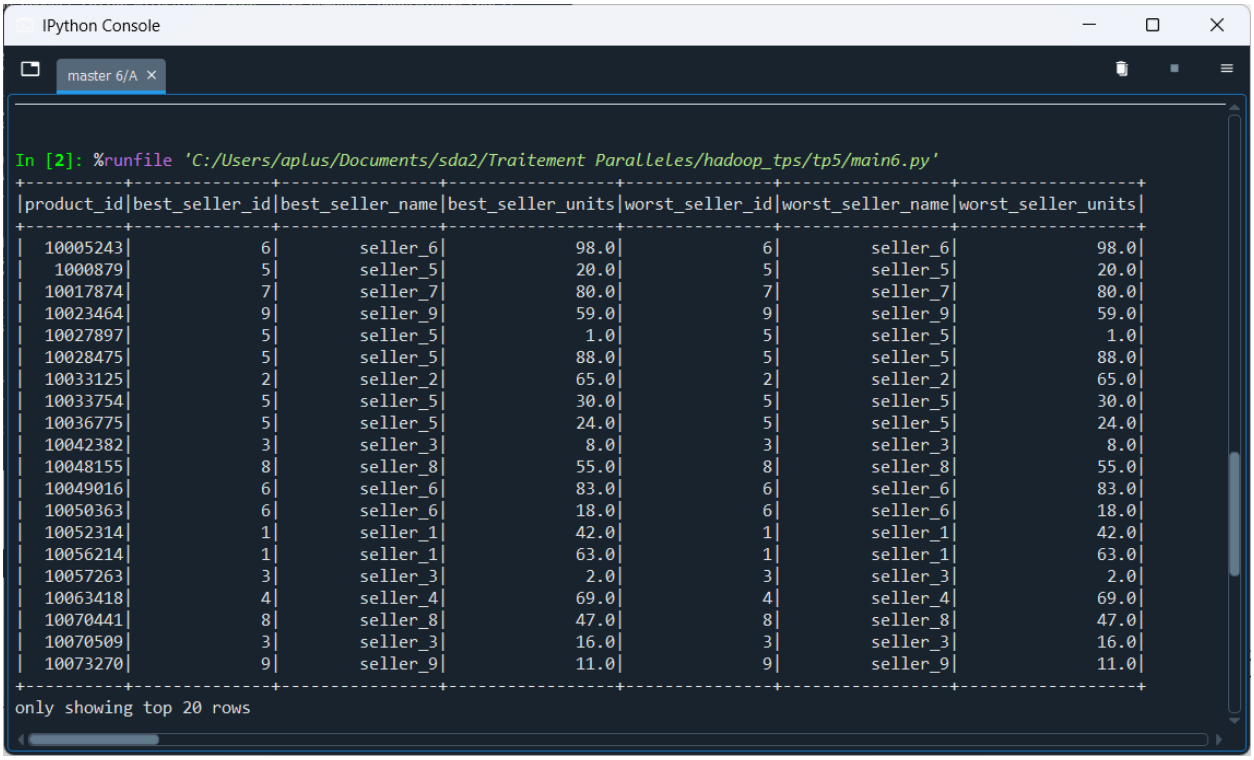
Pour identifier le meilleur et le pire vendeur pour chaque produit, nous devons :

1. Calculer le nombre total d'unités vendues par vendeur pour chaque produit
2. Utiliser des fonctions de fenêtrage pour déterminer le classement des vendeurs par produit

```
from pyspark.sql.functions import col, sum as sql_sum, row_number
from pyspark.sql.window import Window

# Calcul du nombre total d'unités vendues par vendeur pour chaque produit
sales_by_product_seller = sales_df.groupBy("product_id", "seller_id") \
    .agg(sql_sum("num_pieces_sold").alias("total_sold"))

# Jointure avec les noms des vendeurs
sales_by_product_seller = sales_by_product_seller.join(sellers_df,
"seller_id")

# Définition de la fenêtre par produit
window_spec =
Window.partitionBy("product_id").orderBy(col("total_sold").desc())

# Ajout du rang pour chaque vendeur par produit
ranked_sellers = sales_by_product_seller.withColumn("rank",
row_number().over(window_spec))

# Sélection du meilleur vendeur (rang 1)
best_sellers = ranked_sellers.filter(col("rank") == 1) \
    .select("product_id",
            col("seller_id").alias("best_seller_id"),
            col("seller_name").alias("best_seller_name"),
            col("total_sold").alias("best_seller_units"))

# Tri par ordre croissant pour trouver le pire vendeur
window_spec_asc =
Window.partitionBy("product_id").orderBy(col("total_sold").asc())
```

```
ranked_sellers_asc = sales_by_product_seller.withColumn("rank",
row_number().over(window_spec_asc))

# Sélection du pire vendeur (rang 1 en ordre croissant)
worst_sellers = ranked_sellers_asc.filter(col("rank") == 1) \
    .select("product_id",
            col("seller_id").alias("worst_seller_id"),
            col("seller_name").alias("worst_seller_name"),
            col("total_sold").alias("worst_seller_units"))

# Jointure des résultats pour obtenir le meilleur et le pire vendeur par
produit
final_result = best_sellers.join(worst_sellers,
"product_id").orderBy("product_id")

# Affichage des résultats
final_result.show()
```

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import (
    col,
    sum as sql_sum,
    row_number
)
from pyspark.sql.window import Window


# Initialisation de SparkSession
spark = SparkSession.builder \
    .appName("Test") \
    .getOrCreate()

sc = spark.sparkContext

# Chargement des données
sales_df = spark.read.parquet("/data/sales_parquet")
products_df = spark.read.parquet("/data/products_parquet")
sellers_df = spark.read.parquet("/data/sellers_parquet")

# Calcul du nombre total d'unités vendues par vendeur pour chaque produit
sales_by_product_seller = sales_df.groupBy("product_id", "seller_id") \
    .agg(sql_sum("num_pieces_sold").alias("total_sold"))

# Jointure avec les noms des vendeurs
sales_by_product_seller = sales_by_product_seller.join(sellers_df, "seller_id")

# Définition de la fenêtre par produit
window_spec = Window.partitionBy("product_id").orderBy(col("total_sold").desc())

# Ajout du rang pour chaque vendeur par produit
ranked_sellers = sales_by_product_seller.withColumn("rank", row_number().over(window_spec))

# Sélection du meilleur vendeur (rang 1)
best_sellers = ranked_sellers.filter(col("rank") == 1) \
    .select("product_id",
            col("seller_id").alias("best_seller_id"),
            col("seller_name").alias("best_seller_name"),
            col("total_sold").alias("best_seller_units"))

# Tri par ordre croissant pour trouver le pire vendeur
window_spec_asc = Window.partitionBy("product_id").orderBy(col("total_sold").asc())
ranked_sellers_asc = sales_by_product_seller \
    .withColumn("rank", row_number().over(window_spec_asc))

# Sélection du pire vendeur (rang 1 en ordre croissant)
worst_sellers = ranked_sellers_asc.filter(col("rank") == 1) \
    .select("product_id",
            col("seller_id").alias("worst_seller_id"),
            col("seller_name").alias("worst_seller_name"),
            col("total_sold").alias("worst_seller_units")
    )

# Jointure des résultats pour obtenir le meilleur et le pire vendeur par produit
final_result = best_sellers.join(worst_sellers, "product_id").orderBy("product_id")

# Affichage des résultats
final_result.show()
```




# Conclusion

Ce TP nous a permis d'explorer les fonctionnalités essentielles de PySpark pour l'analyse de données distribuées. Nous avons vu comment :

1. Configurer et utiliser PySpark dans un environnement de développement
2. Travailler avec les structures de données fondamentales (RDD et DataFrame)
3. Effectuer des opérations d'analyse complexes sur un jeu de données commercial

Les points clés à retenir sont :

- PySpark combine la puissance de traitement d'Apache Spark avec la simplicité et la richesse de l'écosystème Python
- Les DataFrames offrent une interface intuitive pour manipuler des données structurées
- PySpark SQL permet d'exécuter des requêtes SQL directement sur les données
- Les opérations de fenêtrage (window functions) sont très utiles pour les analyses comparatives

Ce TP a démontré l'intérêt de PySpark pour l'analyse de données à grande échelle, en particulier pour les cas d'usage impliquant des calculs complexes sur des volumes importants de données structurées.