

Problème : Analyse d'un Grand Ensemble de Données de Ventes

Importation des bibliothèques nécessaires

Le code importe plusieurs bibliothèques Python utilisées pour la gestion des fichiers, la manipulation des données et la gestion des types :

- `h5py` : Permet de travailler avec des fichiers HDF5, un format utilisé pour stocker des données numériques complexes.
- `os` : Fournit une interface pour interagir avec le système d'exploitation (par exemple, manipulation des fichiers).
- `time` : Utilisé pour mesurer le temps d'exécution des opérations.
- `typing` : Permet de spécifier les types de données dans le code (facilite la lisibilité et la gestion du code).
- `sqlite3` : Utilisé pour interagir avec une base de données SQLite.
- `pandas` : Bibliothèque essentielle pour la manipulation de données sous forme de DataFrame (tableaux bidimensionnels).
- `modin.pandas` : C'est une version parallèle de pandas qui peut accélérer les opérations sur les DataFrames en tirant parti de plusieurs cœurs de CPU ou d'un cluster de machines.

In [35]:

```
import h5py as hdf5
import os
import time
from typing import Optional
import sqlite3
import pandas as pd
# import modin.pandas as mpd

# 📁 Chemin du fichier CSV et colonnes à utiliser
path_file: str = "sales_data.csv"
use_cols: list[str] = ['customer_id', 'product_id', 'quantity', 'price']

# 💾 Types de données pour une gestion efficace de La mémoire
dtypes: dict[str, str] = {
    'customer_id': 'uint32',
    'product_id': 'uint16',
    'quantity': 'uint8',
    'price': 'float32',
```

```
}
```

```
# 📏 Configuration de la taille des chunks pour le traitement par lot
```

```
fraction: float = 0.01
```

```
chunk_size_rows: int = 100000
```

```
# 📊 DataFrames globaux pour stocker les données
```

```
data: pd.DataFrame = pd.DataFrame()
```

```
data_transaction: pd.DataFrame = pd.DataFrame()
```

les décorateurs :



Gestion des erreurs avec le décorateur `check_fun_error`

Ce décorateur est utilisé pour capturer et gérer les erreurs qui surviennent lors de l'exécution d'une fonction. Si une exception est levée dans la fonction décorée, elle sera interceptée, et un message d'erreur personnalisé sera affiché, suivi de l'arrêt du programme.

In [36]:

```
# 🎯 Décorateur pour la gestion des erreurs
```

```
def check_fun_error(fun):
    def wrapper(*args, **kwargs):
        try:
            return fun(*args, **kwargs)
        except Exception as e:
            print(f"🔴 An error occurred in function '{fun.__name__}': {str(e)}")
            exit(1)
    return wrapper
```



Mesure du temps d'exécution avec le décorateur `timing`

Ce décorateur mesure et affiche le temps d'exécution d'une fonction. Il est utile pour analyser les performances des fonctions qui peuvent être coûteuses en termes de temps de calcul, en particulier lors du traitement de grandes quantités de données.

```
In [37]: # ⏳ Décorateur pour mesurer Le temps d'exécution d'une fonction
def timing(func):
    def wrapper(*args, **kwargs):
        start_time: float = time.time()
        result = func(*args, **kwargs)
        end_time: float = time.time()
        print(f"⌚ Temps d'exécution de {func.__name__}: {end_time - start_time:.4f} secondes")
        return result
    return wrapper
```

🔧 Connexion à une base de données avec le décorateur connect_to_db

Ce décorateur facilite la gestion de la connexion à une base de données SQLite. Il crée une connexion, passe cette connexion et le curseur de la base de données aux fonctions qui en ont besoin, puis ferme la connexion à la fin de l'exécution de la fonction

```
In [39]: # 🚧 Décorateur pour gérer La connexion à La base de données SQLite
def connect_to_db(func):
    def wrapper(*args, **kwargs):
        try:
            # Établissement de la connexion à la base de données
            connection = sqlite3.connect('sales.db')
            cursor = connection.cursor()
            kwargs['connection'] = connection
            kwargs['cursor'] = cursor
            print("🚧 Connexion à la base de données réussie")

            return func(*args, **kwargs)

        except Exception as e:
            # Gestion des erreurs
            print(f"🔴 Une erreur est survenue dans la fonction '{func.__name__}': {str(e)}")

        finally:
            # Fermeture de la connexion à la base de données
            if 'connection' in locals():
                print("🚧 Fermeture de la connexion à la base de données")
                connection.commit()
```

```
        connection.close()
    return wrapper
```

1. Échantillonnage et Sous-ensemble de Données

- Tâche :
 - Charger un échantillon aléatoire de 1 % des lignes du fichier `sales_data.csv`.
 - Sélectionner uniquement les colonnes `customer_id`, `product_id`, `quantity`, et `price`.
 - Spécifier les types de données appropriés pour réduire la consommation de mémoire.

```
In [ ]: # 📈 Chargement du fichier CSV en chunks (morceaux)
@check_fun_error
def load_file_with_chunks(path_file : str) -> None:
    global data
    data = pd.DataFrame()
    try:
        for ch in pd.read_csv(path_file, usecols=use_cols, dtype=dtypes, chunksize=10000):
            sampled_chunk = ch.sample(frac=fraction, random_state=4)
            data = pd.concat([data, sampled_chunk], ignore_index=True)
    finally:
        if len(data) > 0:
            print("✅ Les données ont été chargées avec succès.")
        else:
            print("⚠️ Un problème est survenu, les données sont vides.")

    if len(data) == 0:
        load_file_with_chunks(path_file=path_file)

print("Taille des données: ", data.info())
```

Les données ont été chargées avec succès.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 4 columns):
 #   Column      Non-Null Count  Dtype  
---  --          -----          ----- 
 0   customer_id 10000 non-null   uint32 
 1   product_id   10000 non-null   uint16 
 2   quantity     10000 non-null   uint8  
 3   price        10000 non-null   float32 
dtypes: float32(1), uint16(1), uint32(1), uint8(1)
memory usage: 107.6 KB
Taille des données: None
```

2. Conversion en Formats de Fichiers Efficaces

- Tâche :
 - Convertir l'échantillon de données en formats Feather et Parquet.
 - Comparer la taille des fichiers et mesurer le temps de chargement pour chaque format.

```
In [41]: # 📈 Convertir les données en format Feather ou Parquet
@check_fun_error
@timing
def convert_to_feather_or_parquet(type_convert: str) -> None:
    global data
    type_convert = type_convert.lower()
    if type_convert == "feather":
        print("📦 Conversion au format Feather")
        return data.to_feather('data.feather')
    elif type_convert == "parquet":
        print("📦 Conversion au format Parquet")
        return data.to_parquet('data.parquet')
    else:
        print("⚠️ Format non pris en charge. Veuillez choisir entre 'feather' ou 'parquet'.")
```



```
# 📄 Lire les fichiers en différents formats (Feather, Parquet, CSV)
@check_fun_error
```

```

@timing
def read_files_feather_parquet_csv(type_file: str) -> Optional[pd.DataFrame]:
    type_file = type_file.lower()
    data.to_csv('data.csv', index=False)
    if type_file == "feather":
        print("📖 Lecture d'un fichier Feather")
        return pd.read_feather("data.feather")
    elif type_file == "parquet":
        print("📖 Lecture d'un fichier Parquet")
        return pd.read_parquet("data.parquet")
    elif type_file == "csv":
        print("📖 Lecture d'un fichier CSV")
        return pd.read_csv("data.csv")
    else:
        print("⚠ Type de fichier non pris en charge. Choisissez parmi 'feather', 'parquet', ou 'csv'.")
        return None

# 📈 Comparer les tailles de fichiers
@check_fun_error
def compare_size_files() -> None:

    data.to_csv('data.csv', index=False)
    file_size_csv = os.path.getsize('data.csv')
    file_size_feather = os.path.getsize('data.feather')
    file_size_parquet = os.path.getsize('data.parquet')

    print(f"📏 Taille du fichier CSV: {file_size_csv:.2f} octets")
    print(f"📏 Taille du fichier Feather: {file_size_feather:.2f} octets")
    print(f"📏 Taille du fichier Parquet: {file_size_parquet:.2f} octets")

    # Comparaison des tailles des fichiers
    if file_size_csv > file_size_feather and file_size_csv > file_size_parquet:
        print("📁 Le fichier CSV est le plus grand.")
    elif file_size_feather > file_size_csv and file_size_feather > file_size_parquet:
        print("📁 Le fichier Feather est le plus grand.")
    elif file_size_parquet > file_size_csv and file_size_parquet > file_size_feather:
        print("📁 Le fichier Parquet est le plus grand.")
    else:
        print("📁 Plusieurs fichiers ont des tailles équivalentes.")

# Utilisation des fonctions

```

```

print("#" * 60)
df = read_files_feather_parquet_csv("csv")
df = read_files_feather_parquet_csv("parquet")
df = read_files_feather_parquet_csv("feather")

print("#" * 60)
convert_to_feather_or_parquet("feather")

print("#" * 60)
convert_to_feather_or_parquet("parquet")

print("#" * 60)
compare_size_files()

```

```

#####
# Lecture d'un fichier CSV
⌚ Temps d'exécution de read_files_feather_parquet_csv: 0.0778 secondes
# Lecture d'un fichier Parquet
⌚ Temps d'exécution de read_files_feather_parquet_csv: 0.0336 secondes
# Lecture d'un fichier Feather
⌚ Temps d'exécution de read_files_feather_parquet_csv: 0.0242 secondes
#####
# Conversion au format Feather
⌚ Temps d'exécution de convert_to_feather_or_parquet: 0.0032 secondes
#####
# Conversion au format Parquet
⌚ Temps d'exécution de convert_to_feather_or_parquet: 0.0075 secondes
#####
# Taille du fichier CSV: 205935.00 octets
# Taille du fichier Feather: 110730.00 octets
# Taille du fichier Parquet: 159180.00 octets
📁 Le fichier CSV est le plus grand.

```

3. Utilisation de HDF5

- Tâche :
 - Créer un fichier HDF5 et stocker l'échantillon de données dans une table appelée `sales_sample`.
 - Ajouter une autre table contenant les transactions dont le prix est supérieur à `100 DH`.
 - Lire les 5 premières lignes de la table `sales_sample`.

In [42]:

```
# 📁 Sauvegarder les données dans un fichier HDF5
@check_fun_error
def file_hdf5(file_name: str, data_transaction_supp_100: pd.DataFrame) -> None:

    try:
        # Ouverture du fichier HDF5 en mode écriture
        with hdf5.File(file_name, "w") as hd5_file:
            # Création des datasets dans le fichier HDF5
            hd5_file.create_dataset('sales_sample', data=data)
            hd5_file.create_dataset('sales_high_transaction', data=data_transaction_supp_100)

        print("✅ Données sauvegardées avec succès dans le fichier HDF5.")

    except Exception as e:
        # En cas d'erreur, afficher un message et sortir
        print(f"❌ Une erreur est survenue : {str(e)}")

# 📄 Lire les cinq premières lignes du fichier HDF5
@check_fun_error
def read_first_five_rows_from_hdf5(file_name: str) -> None:
    try:
        # Ouverture du fichier HDF5 en mode lecture
        with hdf5.File(file_name, 'r') as hdf:

            sales_sample_data = hdf['sales_sample'][0:5]
            print(sales_sample_data)
            print('#' * 60)
            df = pd.DataFrame(sales_sample_data, columns=data.columns)
            print(df.head())

    except KeyError:
        # Si le dataset n'existe pas dans le fichier
        print("❌ Clé 'sales_sample' non trouvée dans le fichier HDF5.")
    except Exception as e:
        # En cas d'erreur générale
        print(f"❌ Une erreur est survenue lors de la lecture du fichier HDF5 : {str(e)}")

    data_transaction_supp_100 = data[data.price > 100]
```

```
# Sauvegarde des données dans le fichier HDF5
file_hdf5('sales_data.h5', data_transaction_supp_100=data_transaction_supp_100)

# Lecture des cinq premières lignes du fichier HDF5
read_first_five_rows_from_hdf5('sales_data.h5')
```

Données sauvegardées avec succès dans le fichier HDF5.

```
[[7.3973000e+04 9.2000000e+02 3.0000000e+00 4.11279999e+02]
 [8.4362000e+04 2.0600000e+02 1.0000000e+01 2.34100006e+02]
 [2.4619000e+04 7.1120000e+03 5.0000000e+00 3.37130005e+02]
 [5.2100000e+03 3.9200000e+03 7.0000000e+00 1.04580002e+02]
 [7.4271000e+04 4.9920000e+03 2.0000000e+00 9.2000000e+01]]
#####
customer_id  product_id  quantity      price
0            73973.0    920.0       3.0  411.279999
1            84362.0    206.0      10.0  234.100006
2            24619.0    7112.0      5.0   337.130005
3            5210.0     3920.0      7.0   104.580002
4            74271.0    4992.0      2.0   92.000000
```

4. Lecture par Morceaux

- Tâche :
 - Lire le fichier `sales_data.csv` par morceaux de `100 000` lignes.
 - Filtrer les transactions ayant une quantité supérieure à `10` pour chaque morceau.
 - Combiner les résultats filtrés dans un seul DataFrame et calculer la valeur totale des ventes (`quantité * prix`) .

In [43]:

```
# 📈 Traitement du fichier CSV par morceaux et renvoi des résultats
@check_fun_error
def read_file_from_rows(file_path: str, chunk_size: int) -> pd.DataFrame:
    for chunk in pd.read_csv(file_path, chunksize=chunk_size):
        yield chunk

# 💼 Combiner les transactions et calculer la valeur totale
@check_fun_error
def combine_and_calcul_total(min_qty: int) -> None:
    global data_transaction
```

```
data_transaction = pd.DataFrame()

# Lecture du fichier par morceaux et filtrage
for chunk in read_file_from_rows('sales_data.csv', chunk_size=chunk_size_rows):

    data_transaction = pd.concat([data_transaction, chunk[chunk.quantity > min_qty]], ignore_index=False)

# Calcul de la valeur totale des transactions
data_transaction['total_value'] = data_transaction['price'] * data_transaction['quantity']
total = data_transaction['total_value'].sum()

# Affichage des 10 premières lignes et du total
print(f"📊 Données (Premières {chunk_size_rows} lignes) : \n", data_transaction.head(10))
print("סכום Total:", total)

# Exemple d'utilisation avec une quantité minimale de 5
combine_and_calcul_total(5)
```

📊 Données (Premières 100000 lignes) :

```
transaction_id  customer_id  product_id  quantity  price  \
0              1            15796        111       7  336.48
4              5            6266         7572      7  39.62
5              6            82387        9041      6  60.97
10             11           16024        7658      8  26.89
11             12           41091        3227      9 290.79
14             15            770         3597     10 149.93
16             17           62956        9581      7 380.04
21             22           53708        4655      8 376.55
23             24           28694        9645      7 338.83
25             26           93017        1958      7  81.08

transaction_date  region  total_value
0    2023-10-03  North America  2355.36
4    2023-07-04  North America  277.34
5    2021-11-26  North America  365.82
10   2021-08-11  Australia    215.12
11   2020-09-22  South America 2617.11
14   2022-10-16  North America 1499.30
16   2023-08-10  Asia          2660.28
21   2020-01-04  Australia    3012.40
23   2023-08-07  Australia    2371.81
25   2020-09-27  North America  567.56

💡 Valeur totale: 1021523122.110003
```

5. Chargement dans une Base de Données SQLite

- Tâche :
 - Créer une base de données SQLite et charger l'intégralité du fichier `sales_data.csv` dans une table appelée `sales`.
 - Exécuter une requête SQL pour extraire les transactions dans la région `Europe` avec un prix supérieur à `50 DH`.
 - Calculer la valeur totale des ventes pour ces transactions.

```
In [44]: import pandas as pd
import sqlite3
from typing import Optional

# 📈 Convertir un fichier CSV en base de données SQL
```

```
@check_fun_error
@connect_to_db
def convert_csv_to_sql_db(connection: sqlite3.Connection, cursor: Optional[sqlite3.Cursor] = None) -> None:

    # Conversion des données en base de données SQL (Table 'sales')
    data_transaction.to_sql('sales', connection, if_exists='replace', index=False)
    print("✅ Données CSV converties en base de données SQL avec succès!")

# 🔍 Sélectionner des transactions spécifiques depuis la base de données
@check_fun_error
@connect_to_db
def select_transaction_group_by_eur(cursor: sqlite3.Cursor, connection: Optional[sqlite3.Connection] = None) -> pd.DataFrame:

    # Exécution de la requête SQL pour filtrer les transactions par région Europe
    stm = cursor.execute("""SELECT * FROM sales WHERE region='Europe' """)
    # Récupération des résultats et conversion en DataFrame
    rows = pd.DataFrame(stm.fetchall(), columns=data_transaction.columns)
    return rows

# Conversion du fichier CSV en base de données SQL
convert_csv_to_sql_db()

# Sélectionner et afficher les transactions de la région 'Europe'
rows = select_transaction_group_by_eur()
print(rows.head())
```

```

💡 Connexion à la base de données réussie
✅ Données CSV converties en base de données SQL avec succès!
💡 Fermeture de la connexion à la base de données
💡 Connexion à la base de données réussie
💡 Fermeture de la connexion à la base de données
transaction_id  customer_id  product_id  quantity  price  transaction_date \
0              59          67122       5281      9    23.44  2020-11-08
1              79          40398       5476      7    37.09  2022-10-01
2              83          55592       1249      9   336.71  2022-01-19
3              84          89813       7023     10    79.86  2023-05-26
4              94          39100       1501     10   318.90  2022-05-17

region  total_value
0 Europe      210.96
1 Europe      259.63
2 Europe     3030.39
3 Europe      798.60
4 Europe     3189.00

```

Réponses aux Questions du TP

1. Échantillonnage et Sous-ensemble de Données

- Pourquoi est-il utile de charger un échantillon aléatoire de données plutôt que l'ensemble complet ?
 - Permet d'économiser la mémoire.
 - Accélère les analyses.
 - Permet de tester les scripts sur un sous-ensemble représentatif avant de travailler sur l'ensemble des données.
- Comment la spécification des types de données peut-elle réduire la consommation de mémoire ?
 - En choisissant des types de données plus petits, comme int32, float32 ou category.

2. Conversion en Formats de Fichiers Efficaces

- Quels sont les avantages des formats Feather et Parquet par rapport au format CSV ?

- **Efficacité en taille** : Feather et Parquet compressent les données, réduisant ainsi la taille des fichiers.
 - **Rapidité** : Ces formats permettent une lecture et une écriture plus rapides.
 - **Structure binaire** : Mieux adaptés pour manipuler des données complexes avec des métadonnées, contrairement au CSV, qui est un format texte.
- Quand préférer Feather à Parquet?
 - **Feather** : Idéal pour un usage rapide en Python, particulièrement avec pandas pour des manipulations en mémoire.
 - **Parquet** : Préféré pour les projets nécessitant une compatibilité avec plusieurs outils (Hadoop, Spark) et une meilleure compression pour les données volumineuses.
-

3. Utilisation de HDF5

-Qu'est-ce qu'un fichier HDF5 et comment est-il structuré ?

- Un fichier HDF5 est un format binaire hiérarchique conçu pour organiser et stocker de grandes quantités de données.
- Structure :
 - **Groupes** : Similaires à des dossiers.
 - **Datasets** : Similaires à des fichiers.

-Pourquoi utiliser HDF5 plutôt qu'un fichier CSV ?

- **Efficacité** : Lecture/écriture plus rapides pour de grands volumes de données.
 - **Compression** : Réduction de la taille des fichiers.
 - **Flexibilité** : Supporte des structures complexes et permet d'accéder à des parties spécifiques sans charger tout le fichier.
-

4. Lecture par Morceaux

- Pourquoi lire un fichier volumineux par morceaux ?
 - Pour éviter de dépasser la capacité de la mémoire vive (RAM).
 - Permet un traitement progressif des données volumineuses.
- Comment filtrer et combiner des données provenant de morceaux ?

- Utiliser une boucle avec pandas.read_csv() et le paramètre chunksize.
 - Appliquer des filtres sur chaque morceau.
 - Combiner les résultats à l'aide de pd.concat().
-

5. Chargement dans une Base de Données

- Quels sont les avantages de stocker des données dans une base de données SQLite plutôt que dans un fichier CSV ?
 - **Requêtes complexes** : Permet d'exécuter des requêtes SQL pour analyser efficacement les données.
 - **Structure** : Organisation claire avec des types définis.
 - **Performances** : Accès et filtrage des données spécifiques plus rapides.
- Comment exécuter des requêtes SQL sur une base de données SQLite à partir de Python ?

- Utiliser la bibliothèque sqlite3 :

```
import sqlite3
```

```
connection = sqlite3.connect('nom_base_donne.db')
cursor = connection.cursor()
cursor.execute("Requete")
connection.commit()
connection.close()
```

Mohamed BELANNAB