Pandas vs Polars — Sheet pratique (avec données synthétiques)

Georf MIGUIAMA BAMBA

1) Définitions

Pandas
Bibliothèque Python pour la manipulation et l'analyse de données, avec des structures **DataFrame** et **Series** adaptées au nettoyage, à la transformation et à l'exploration des données.

Polars
Bibliothèque moderne (Python/Rust) pour le traitement de données, fondée sur **Apache Arrow**, **multi-threadée**, et proposant un mode **lazy** (`scan_*` → `collect`) qui applique des optimisations automatiques (predicate/projection pushdown).

Références : Polars — Lazy API, Polars — Coming from Pandas, Pandas DataFrame (docs), Cheatsheet Pandas→Polars (Rho Signal)

2) Différences clés (venir de Pandas)

- **Index** : Pandas utilise un index de ligne; **Polars n'utilise pas d'index** (positions de ligne), ce qui rend les opérations plus explicites.
- **Mémoire** : Pandas s'appuie surtout sur **NumPy**; Polars suit le format colonnaire **Apache Arrow**.
- **Parallélisme & Lazy** : Polars utilise le **multi-thread** et un **plan d'exécution lazy** (`scan_*`, `collect`, `.explain()`).
- **Performance** : Polars est généralement plus rapide et plus économe en mémoire sur gros volumes.

Réfs : Coming from Pandas, Lazy API

3) Génération du dataset (paramètres fournis)

```python
import numpy as np
from datetime import datetime, timedelta

n_rows = 1000
categories = ["Electronics", "Furniture", "Stationery", "Kitchen", "Toys"]
products = [
    "Laptop", "Mouse", "Chair", "Desk", "Pen", "Notebook",
    "Monitor", "Lamp", "Keyboard", "Cable"
]

np.random.seed(42)
data = {
    "id": np.arange(1, n_rows + 1),
    "date": [
        (datetime(2025, 1, 1) + timedelta(days=np.random.randint(0, 90))).strftime('%Y-%m
        for _ in range(n_rows)
    ],
    "category": np.random.choice(categories, n_rows),
    "product": np.random.choice(products, n_rows),
    "price": np.random.uniform(5, 1200, n_rows).round(2),
    "quantity": np.random.randint(1, 50, n_rows),
    "active": np.random.choice([True, False], n_rows, p=[0.8, 0.2]),
    "score": np.random.randint(0, 100, n_rows),
}
```

4) Lecture & Lazy

Pandas

```
import pandas as pd

df_pd = pd.DataFrame(data)
df_pd.to_csv("data.csv", index=False)
df_pd_csv = pd.read_csv("data.csv")
# Parquet (requiert pyarrow/fastparquet)
df_pd.to_parquet("data.parquet")
```

Polars

```
import polars as pl

df_pl = pl.DataFrame(data)
df_pl.write_parquet("data.parquet")
df_pl_csv = pl.read_csv("data.csv")
df_pl_parquet = pl.read_parquet("data.parquet")

# Lazy + optimisations de lecture
q = (pl.scan_csv("data.csv")
       .filter(pl.col("price") > 100)
       .select(["id","price","category"]))
df_lazy = q.collect()
```

5) Inspection

Pandas

```
df_pd_csv.shape
df_pd_csv.head(3)
df_pd_csv.dtypes
```

Polars

```
df_pl_csv.shape
df_pl_csv.head(3)
df_pl_csv.dtypes
```

6) Sélection & cast

Pandas

```
df_pd2 = df_pd_csv.assign(price_float=lambda d: d["price"].astype("float64"))
df_pd2[["price","price_float"]].head(3)
```

Polars

```
df_pl2 = df_pl_csv.with_columns(pl.col("price").cast(pl.Float64).alias("price_float"))
df_pl2.select(["price","price_float"]).head(3)
```

7) Filtres

Pandas

```
f_pd = df_pd2[(df_pd2["category"] == "Electronics") & (df_pd2["price"] > 500)]
f_pd.shape
```

Polars

```
f_pl = df_pl2.filter((pl.col("category") == "Electronics") & (pl.col("price") > 500))
f_pl.shape
```

8) Colonnes dérivées

> En Polars, évite de référencer une nouvelle colonne dans la **même** `with_columns`. Procède en **deux passes**.

Pandas

```
df_pd5 = df_pd_csv.copy()
df_pd5["total"] = df_pd5["price"] * df_pd5["quantity"]
df_pd5["high_value"] = (df_pd5["total"] > 10000)
```

Polars

```
df_pl5 = df_pl_csv.with_columns((pl.col("price")*pl.col("quantity")).alias("total"))
df_pl5 = df_pl5.with_columns((pl.col("total") > 10000).alias("high_value"))
```

9) GroupBy & agrégations

Pandas

```
g_pd = (df_pd5.groupby("category", as_index=False)
              .agg({"quantity":"sum","price":"mean","score":"max"}))
```

Polars

```
g_pl = (df_pl5.group_by("category")
              .agg([pl.col("quantity").sum().alias("quantity_sum"),
                    pl.col("price").mean().alias("price_mean"),
                    pl.col("score").max().alias("score_max")])
              .sort("category"))
```

---

10) Joins

Pandas

```
suppliers = ["Contoso","Northwind","Fabrikam","AdventureWorks"]
prod_unique = sorted(set(df_pd5["product"]))
sup_pd = pd.DataFrame({"product": prod_unique,
                       "supplier": [np.random.choice(suppliers) for _ in prod_unique]})

j_pd = pd.merge(df_pd5, sup_pd, on="product", how="left")
```

Polars

```
sup_pl = pl.DataFrame({"product": prod_unique,
                       "supplier": [np.random.choice(suppliers) for _ in prod_unique]})

j_pl = df_pl5.join(sup_pl, on="product", how="left")
```

---

11) Tri

Pandas

```
s_pd = df_pd5.sort_values(["category","price"], ascending=[True, False]).head(5)
```

Polars

```
s_pl = df_pl5.sort(by=["category","price"], descending=[False, True]).head(5)
```

---

12) Reshape (melt/pivot)

Pandas

```
m_pd = df_pd5.melt(id_vars=["id","category"], value_vars=["quantity","score"],
                   var_name="metric", value_name="value")
```

```
p_pd = (df_pd5.pivot_table(index="category", columns="product", values="quantity",
                           aggfunc="sum", fill_value=0).reset_index())
```

Polars

```
m_pl = (df_pl5.melt(id_vars=["id","category"], value_vars=["quantity","score"])
              .rename({"variable":"metric","value":"value"}))

p_pl = df_pl5.pivot(values="quantity", index="category", columns="product", aggregate_fn=
```

13) Données manquantes

Pandas

```
idx_nan = df_pd5.sample(frac=0.02, random_state=42).index
df_pd_nan = df_pd5.copy()
df_pd_nan.loc[idx_nan, "product"] = None

df_pd_drop = df_pd_nan.dropna(subset=["product"])  # supprime lignes avec product manquan
df_pd_fill = df_pd_nan.fillna({"product":"Unknown"})
```

Polars

```
df_pl_nan = df_pl5.with_columns(
    pl.when(pl.col("id") % 50 == 0).then(pl.lit(None)).otherwise(pl.col("product")).alias
)

df_pl_drop = df_pl_nan.drop_nulls(subset=["product"])  # supprime lignes avec null dans p
df_pl_fill = df_pl_nan.with_columns(pl.col("product").fill_null("Unknown"))
```

14) Datetime & strings

Pandas

```
df_pd_dt = df_pd5.copy()
df_pd_dt["date"] = pd.to_datetime(df_pd_dt["date"])  # parsing

df_pd_dt["month"] = df_pd_dt["date"].dt.month

df_pd_dt["product_upper"] = df_pd_dt["product"].str.upper()
```

Polars

```
df_pl_dt = (df_pl5
            .with_columns(pl.col("date").strptime(pl.Date, fmt="%Y-%m-%d"))
            .with_columns(pl.col("date").dt.month().alias("month"),
                          pl.col("product").str.to_uppercase().alias("product_upper")))
```

15) Mini-benchmark (indicatif, 1000 lignes)

Pandas

```
import time

_t0 = time.time(); _ = df_pd.groupby("category")["quantity"].sum(); t_pd_grp = (time.time
_t0 = time.time(); _ = pd.merge(df_pd, sup_pd, on="product", how="left"); t_pd_join = (ti
print(f"GroupBy sum Pandas: {t_pd_grp:.2f} ms | Join Pandas: {t_pd_join:.2f} ms")
```

Polars

```
_t0 = time.time(); _ = df_pl.group_by("category").agg(pl.col("quantity").sum()); t_pl_grp
_t0 = time.time(); _ = df_pl.join(pl.DataFrame(sup_pd), on="product", how="left"); t_pl_j
print(f"GroupBy sum Polars: {t_pl_grp:.2f} ms | Join Polars: {t_pl_join:.2f} ms")
```

> À cette échelle, les écarts sont limités ; Polars prend l'avantage sur des volumes plus importants grâce au **multi-thread** et au **lazy** (predicate/projection pushdown). Réfs : Lazy API.

---

16) Références

- Polars — Lazy API : https://docs.pola.rs/user-guide/concepts/lazy-api/
- Polars — Coming from Pandas : https://docs.pola.rs/user-guide/migration/pandas/
- Cheatsheet Pandas→Polars (Rho Signal) : https://www.rhosignal.com/posts/polars-pandas-cheatsheet/
- Pandas DataFrame (docs) : https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html

---

Suivez-moi sur mes réseaux

Ou utilisez des emojis :
▶ YouTube | TikTok