

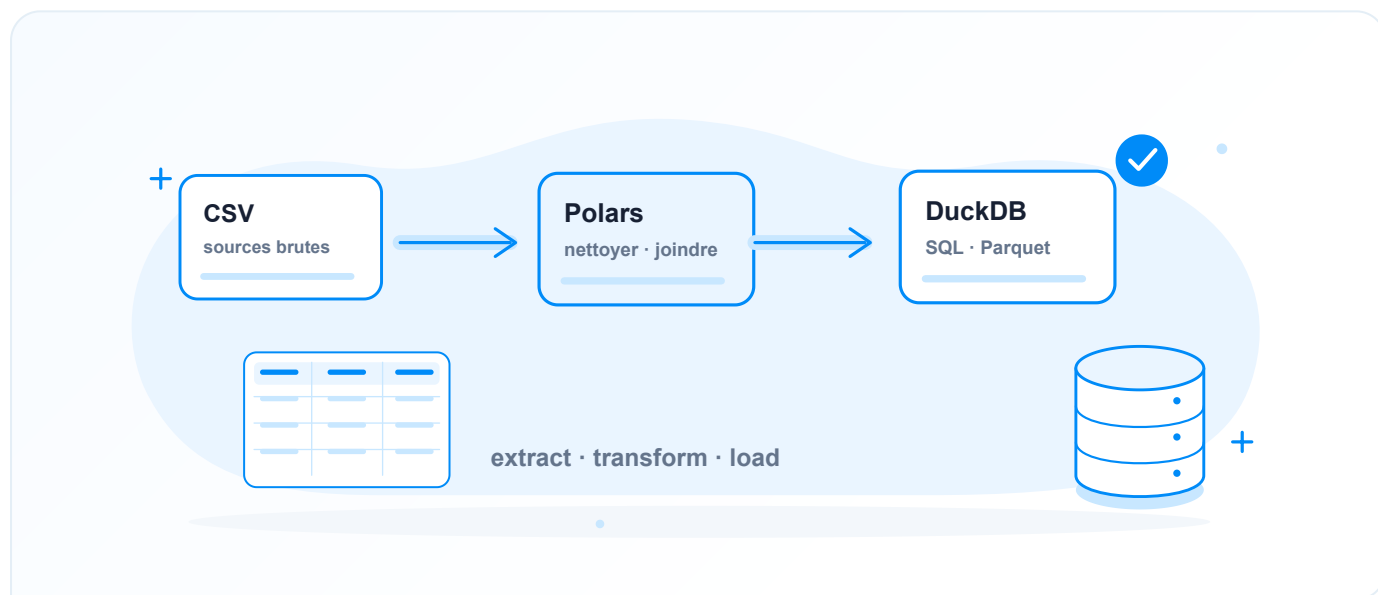
Mini-série

Data engineering · ETL · Polars · DuckDB

Construire un pipeline ETL avec Polars et DuckDB

Construire un pipeline ETL local qui extrait des ventes CSV, les transforme avec Polars, les écrit en Parquet et les interroge avec DuckDB.

Auteur **Youness Boutyour**

**DURÉE**

3 épisodes · 50 min /
épisode

NIVEAU

Intermédiaire

FORMAT

Mini-série · 3 épisodes

Data engineering · ETL · Polars · DuckDB · publié le 2026-10-02 · local

Objectifs et livrables

Un reporting mensuel devient vite fragile quand les commandes, les clients et les règles métier sont traités dans des scripts séparés. Dans cette mini-série, nous allons construire un pipeline ETL local à partir d'un cas simple mais réaliste : calculer le chiffre d'affaires payé par mois et par région à partir de deux fichiers CSV.

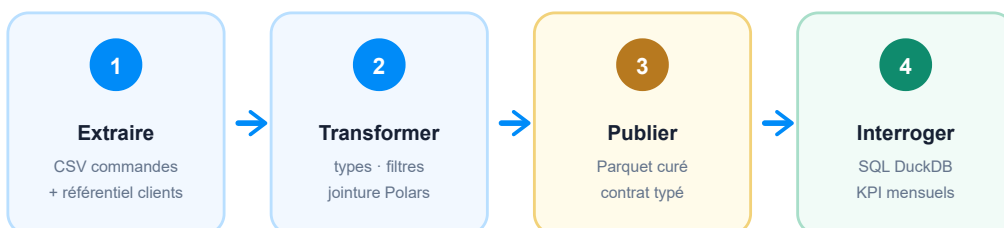
Le pipeline commencera par extraire les commandes et le référentiel clients avec Polars. Nous poserons un contrat de données, convertirons les types, normaliserons les statuts et les canaux, exclurons les commandes annulées, puis enrichirons chaque vente par une jointure client. Le résultat sera publié en Parquet, format colonnaire adapté à l'analyse.

DuckDB prendra ensuite le relais : il lira le Parquet, créera une table locale et exécutera une requête SQL d'agrégation. Les contrôles et les tests vérifieront que les lignes annulées sont bien écartées, que le montant calculé est positif et que les livrables attendus sont produits. L'archive contient le code, les données d'exemple, les tests et une commande unique pour rejouer tout le flux.

Les logos de Polars et DuckDB proviennent de leurs dépôts officiels. Le schéma ETL est une composition vectorielle locale créée pour cette mini-série ; il reprend uniquement les étapes et les responsabilités du pipeline. Ces visuels identifient les outils et le concept ; ils ne constituent pas une communication conjointe ni une indication d'endorsement.

De la source brute au KPI

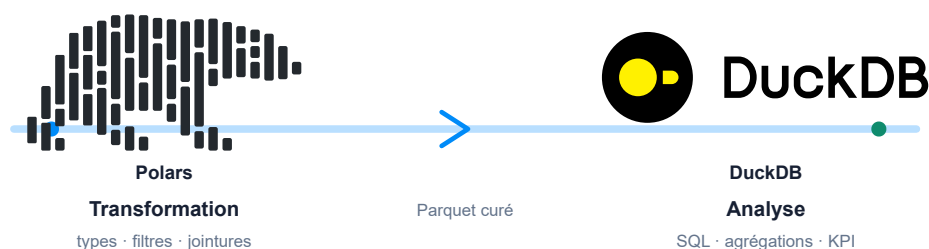
Chaque étape produit une sortie vérifiable avant de passer à la suivante



Flux ETL de référence : extraire, transformer, publier, puis interroger les données.

Les technologies utilisées

Deux rôles distincts, reliés par le fichier Parquet



Polars prépare le DataFrame ; DuckDB charge le Parquet et exécute les agrégations SQL.

Faire circuler une donnée fiable jusqu'au KPI

Schéma du pipeline ETL qui sépare les sources brutes, les transformations Polars, le Parquet curé et les agrégations DuckDB.



Chaque frontière du pipeline a une responsabilité : préparer, publier, puis interroger les données.

Lire le pipeline avant de coder

Le schéma met en évidence les frontières où une donnée peut changer de forme. Les CSV représentent les sources brutes, Polars applique les règles de typage et de jointure, Parquet publie un résultat curé, puis DuckDB exécute les agrégations. Une erreur de montant doit donc être recherchée dans la transformation avant d'être attribuée à la requête SQL.

Les contrôles ont deux fonctions différentes : les assertions vérifient les invariants du pipeline, tandis que les indicateurs finaux vérifient que le résultat est exploitable. En rejouant le flux depuis les mêmes fichiers, vous pouvez comparer les sorties sans dépendre d'un état caché dans un notebook.

Quelques repères techniques

ETL, Parquet et séparation des responsabilités

Extraire signifie lire les sources sans y appliquer silencieusement les règles métier. Transformer consiste à typer, nettoyer, joindre, filtrer et dériver les colonnes nécessaires. Charger signifie publier un résultat stable dans un format ou un moteur que les consommateurs peuvent interroger.

Polars est utilisé ici pour ses DataFrames et ses expressions vectorisées. Les opérations sont décrites par colonnes : conversion de date, normalisation de chaînes, calcul de `quantity × unit_price` et jointure avec le référentiel. DuckDB est utilisé comme moteur analytique embarqué : il exécute du SQL directement dans le processus Python et sait lire le Parquet sans serveur séparé.

Parquet stocke les colonnes avec leurs types et permet de ne lire que les champs utilisés par une requête. Le pipeline garde donc trois frontières explicites : les CSV bruts, le Parquet curé et la base DuckDB locale. Cette séparation rend les erreurs plus faciles à localiser et permet de rejouer uniquement la partie nécessaire.

Pourquoi ne pas utiliser un LLM pour transformer les données

Les règles de chiffre d'affaires doivent être déterministes. Un LLM pourrait aider à expliquer une anomalie ou à produire une requête à valider, mais il ne doit pas décider silencieusement qu'une commande est payée ou modifier un montant. La mini-série n'utilise donc aucun modèle génératif ; le fichier `llm_client.py` de l'archive documente cette limite si un agent est ajouté plus tard.

Packages et modules utilisés

Le pipeline reste volontairement court : deux packages externes portent les traitements Data et SQL, puis quelques modules de la bibliothèque standard de Python assurent les fichiers, la ligne de commande et les tests. Cette séparation permet de savoir exactement ce qui doit être installé et ce qui est déjà fourni par Python.

Package ou module	Installation	Rôle dans la mini-série
<code>polars</code>	<code>polars ≥ 1.20</code> , <code><2</code>	Lire les CSV, typer les colonnes, normaliser les valeurs, filtrer les commandes payées, joindre le référentiel clients et écrire le Parquet.
<code>duckdb</code>	<code>duckdb ≥ 1.4</code> , <code><2</code>	Ouvrir la base analytique locale, lire le Parquet avec <code>read_parquet</code> et exécuter les agrégations SQL.
<code>pathlib</code>	Bibliothèque standard	Construire des chemins portables vers <code>data/</code> , <code>src/</code> et <code>outputs/</code> .
<code>argparse</code> et <code>json</code>	Bibliothèque standard	Exposer les options de <code>src/pipeline.py</code> et écrire <code>run-summary.json</code> .

Package ou module	Installation	Rôle dans la mini-série
<code>unittest</code>	Bibliothèque standard	Vérifier les lignes curées, les fichiers Parquet, la base DuckDB et les KPI produits.

Le fichier `requirements.txt` ne contient donc ni `pandas`, ni `pyarrow`, ni serveur de base de données. Polars possède le moteur nécessaire pour lire et écrire le Parquet utilisé ici ; DuckDB le relit directement pour les requêtes. Les versions sont bornées par une version majeure afin d'éviter une rupture silencieuse pendant la reproduction du tutoriel.

Sortie / exemple

```
polars ≥ 1.20, <2
duckdb ≥ 1.4, <2
```

L'installation se fait avec le même interpréteur que celui qui exécutera le pipeline : `python -m pip install -r requirements.txt`. La commande `python -c "import polars, duckdb"` vérifie les imports, mais elle ne garantit pas encore que les colonnes, les types et les résultats métier sont corrects ; ces contrôles arrivent dans les étapes suivantes.

1. Préparer les sources et l'environnement

Cette première étape installe les deux bibliothèques, inspecte les sources et vérifie leur schéma. Le jeu de données contient 28 commandes, dont 24 payées et 4 annulées, ainsi que huit clients répartis dans quatre régions. Les statuts comportent volontairement des espaces et des variations de casse pour rendre la transformation observable.

Créez l'environnement :

Terminal · Bash

```
mkdir -p etl-polars-duckdb/{data,src,tests,outputs}
cd etl-polars-duckdb
python3.11 -m venv .venv
source .venv/bin/activate
python -m pip install --upgrade pip
python -m pip install -r requirements.txt
python -c "import polars, duckdb; print('Polars', polars.__version__); print('DuckDB', duckdb.__version__)"
```

Les fichiers `data/orders.csv` et `data/customers.csv` sont fournis dans l'archive. Les colonnes importantes sont :

- orders.csv : identifiant de commande, client, date, quantité, prix unitaire, statut et canal ;
- customers.csv : identifiant client, nom, région et segment.

Extrait des sources CSV

Les deux tableaux donnent un aperçu lisible des données avant transformation. Les commandes portent la mesure commerciale ; le référentiel clients apporte les attributs nécessaires à la jointure.

data/orders.csv — commandes

Commande	Client	Date	Quantité	Prix unitaire (€)	Statut	Canal
A001	C001	2026-01-04	3	120,00	paid	web
A002	C005	2026-01-18	2	215,50	PAID	sales
A003	C002	2026-01-06	4	89,90	PAID	partner
A004	C007	2026-01-27	1	620,00	paid	web

data/customers.csv — référentiel clients

Client	Nom	Région	Segment
C001	Atlas Retail	Casablanca	SMB
C002	Rif Services	Tanger	Enterprise
C003	Océan Distribution	Rabat	SMB
C004	Palmeraie Hotels	Marrakech	Enterprise

L'extrait montre déjà les clés de jointure `customer_id` et les variations de statut que Polars devra normaliser.

Avant d'écrire une règle métier, contrôlez les colonnes obligatoires. Le contrôle reste volontairement explicite pour produire une erreur lisible plutôt qu'une exception de jointure difficile à interpréter.

● ● ● Python

```
import polars as pl

orders = pl.read_csv("data/orders.csv")
customers = pl.read_csv("data/customers.csv")

required_orders = {
```

```
"order_id", "customer_id", "order_date", "quantity",
"unit_price", "status", "channel",
}
required_customers = {"customer_id", "customer_name", "region", "segment"}

assert required_orders <= set(orders.columns)
assert required_customers <= set(customers.columns)
print("commandes:", orders.height, "clients:", customers.height)
```

Sortie obtenue

Sortie / exemple

```
Polars 1.44.x
DuckDB 1.5.x
commandes: 28 clients: 8
```

Les versions exactes peuvent évoluer dans les bornes du fichier requirements.txt. Le résultat métier attendu reste 24 lignes curées et 12 groupes mensuels-régionaux.

Commande de vérification

Depuis le dossier de l'archive, vérifiez les en-têtes et le nombre de lignes :

```
Terminal · Bash

python - <<'PY'
import csv
from pathlib import Path

for name in ("orders.csv", "customers.csv"):
    path = Path("data") / name
    with path.open(newline="", encoding="utf-8") as handle:
        rows = list(csv.reader(handle))
    print(name, ":", len(rows) - 1, "lignes;", len(rows[0]), "colonnes")
PY
```

Sortie / exemple

```
orders.csv : 28 lignes; 7 colonnes
customers.csv : 8 lignes; 4 colonnes
```

La commande vérifie seulement la forme. La cohérence des types, des identifiants et des jointures sera contrôlée avec Polars dans l'épisode suivant.

2. Transformer avec Polars et publier le Parquet

La transformation commence par typer les colonnes. Une date qui reste une chaîne peut être triée lexicalement mais mal utilisée dans une agrégation temporelle ; elle est donc convertie en Date. Les quantités et les prix deviennent numériques, les statuts sont nettoyés, puis seules les lignes PAID sont conservées.

Python

```
import polars as pl

orders = pl.read_csv("data/orders.csv")
customers = pl.read_csv("data/customers.csv")

orders = orders.with_columns(
    pl.col("order_id").cast(pl.Utf8).str.strip_chars(),
    pl.col("customer_id").cast(pl.Utf8).str.strip_chars(),
    pl.col("order_date").str.strptime(pl.Date, "%Y-%m-%d", strict=True),
    pl.col("quantity").cast(pl.Int64),
    pl.col("unit_price").cast(pl.Float64),
    pl.col("status").cast(pl.Utf8).str.strip_chars().str.to_uppercase(),
    pl.col("channel").cast(pl.Utf8).str.strip_chars().str.to_lowercase(),
)

customers = customers.with_columns(
    pl.col("customer_id").cast(pl.Utf8).str.strip_chars(),
    pl.col("region").cast(pl.Utf8).str.strip_chars(),
    pl.col("segment").cast(pl.Utf8).str.strip_chars(),
)

paid_orders = orders.filter(pl.col("status") == "PAID")
```

Vérifiez ensuite l'unicité du référentiel avant la jointure. Une duplication d'un customer_id multiplierait les commandes et gonflerait le chiffre d'affaires sans produire d'erreur de type. La contrainte validate="m:1" demande à Polars de vérifier qu'il y a plusieurs commandes au maximum pour un client unique.

Python

```
if customers.get_column("customer_id").n_unique() != customers.height:
    raise ValueError("customer_id doit être unique")

curated = (
    paid_orders
    .join(customers, on="customer_id", how="inner", validate="m:1")
    .with_columns(
        (pl.col("quantity") * pl.col("unit_price"))
        .round(2)
        .alias("revenue_eur"),
        pl.col("order_date").dt.strftime("%Y-%m").alias("month"),
    )
)
```

```
.select(
    "order_id", "order_date", "month", "customer_id",
    "customer_name", "region", "segment", "channel",
    "quantity", "unit_price", "revenue_eur",
)
.sort(["order_date", "order_id"])
)
curated.write_parquet("outputs/sales_curated.parquet")
```

Le Parquet est un contrat de sortie : il contient uniquement des ventes payées, avec les attributs du client et les colonnes calculées dont DuckDB aura besoin. Dans un pipeline plus volumineux, la même logique peut être exprimée en mode lazy avec `scan_csv`, puis optimisée avant `collect()`. Ici, `read_csv` rend le chemin immédiatement observable sur un petit jeu de données.

Sortie obtenue

Sortie / exemple

shape: (24, 11)

order_id	order_date	month	region	revenue_eur
---	---	---	---	---
str	date	str	str	f64
A001	2026-01-04	2026-01	Casablanca	360.0
A003	2026-01-06	2026-01	Tanger	359.6
A005	2026-01-08	2026-01	Rabat	370.0
...	...	...	...	...

Les quatre commandes annulées ne sont pas dans le Parquet. Les noms de colonnes et le type de `order_date` rendent également le contrat visible avant le passage à DuckDB.

Commande de vérification

```
python - <<'PY'
import polars as pl

frame = pl.read_parquet("outputs/sales_curated.parquet")
assert frame.height == 24
assert round(frame["revenue_eur"].sum(), 2) == 16400.30
assert frame["month"].n_unique() == 3
assert frame["region"].n_unique() == 4
```

```
print("Parquet curé: PASS")
print("lignes:", frame.height, "CA:", frame["revenue_eur"].sum())
PY
```

Sortie / exemple

```
Parquet curé: PASS
lignes: 24 CA: 16400.3
```

3. Charger dans DuckDB et produire les KPI

DuckDB peut lire le Parquet directement avec `read_parquet`. Le pipeline crée une base locale `analytics.duckdb` et une table `sales`. La table est reconstruite à chaque exécution avec `CREATE OR REPLACE`, ce qui rend la commande rejouable sans accumuler les lignes d'une exécution précédente.

Python

```
from pathlib import Path
import duckdb

parquet_path = Path("outputs/sales_curated.parquet")
database_path = Path("outputs/analytics.duckdb")

connection = duckdb.connect(str(database_path))
connection.execute(
    "CREATE OR REPLACE TABLE sales AS SELECT * FROM read_parquet(?)",
    [str(parquet_path)],
)
```

La requête suivante calcule quatre indicateurs par mois et par région : chiffre d'affaires, unités vendues et nombre de commandes distinctes. DuckDB exécute le SQL dans le processus Python ; aucun serveur ni fichier de configuration distant n'est nécessaire.

SQL

```
SELECT
    month,
    region,
    ROUND(SUM(revenue_eur), 2) AS revenue_eur,
    SUM(quantity) AS units,
    COUNT(DISTINCT order_id) AS orders
FROM sales
GROUP BY month, region
ORDER BY month, revenue_eur DESC;
```

Dans le code final, `fetchall()` est converti en `DataFrame` Polars avant l'écriture de `outputs/kpis_by_region.csv`. Cette conversion conserve un seul format de travail côté Python et permet de réutiliser les mêmes contrôles que pour le Parquet.

Sortie obtenue

■ Sortie / exemple

ETL terminé : 24 lignes curées, 16400.30 EUR, 12 agrégations

month,region,revenue_eur,units,orders

2026-01,Rabat,1330.0,7,2

2026-01,Marrakech,1085.0,5,2

2026-01,Tanger,979.6,5,2

2026-01,Casablanca,791.0,5,2

Les quatre régions apparaissent chaque mois dans les données d'exemple. Dans un projet réel, une région absente devrait être distinguée d'une région à zéro chiffre d'affaires ; cette décision doit être documentée avant de construire un dashboard.

Commande de vérification

Contrôlez la table DuckDB avec une requête indépendante du code de production :

Terminal · Bash

```
python - <<'PY'
import duckdb

connection = duckdb.connect("outputs/analytics.duckdb", read_only=True)
rows = connection.execute("""
    SELECT COUNT(*) AS rows, COUNT(DISTINCT region) AS regions,
           ROUND(SUM(revenue_eur), 2) AS revenue
    FROM sales
""").fetchone()
assert rows == (24, 4, 16400.30)
print("DuckDB:", rows[0], "lignes;", rows[1], "régions;", rows[2], "EUR")
connection.close()
PY
```

■ Sortie / exemple

DuckDB: 24 lignes; 4 régions; 16400.3 EUR

4. Rejouer le pipeline et automatiser les contrôles

Le point d'entrée complet regroupe les trois étapes afin qu'une nouvelle livraison puisse être rejouée avec une commande. Les chemins sont paramétrables pour remplacer les données d'exemple par un autre export :

Terminal · Bash

```
python -m src.pipeline \  
  --orders data/orders.csv \  
  --customers data/customers.csv \  
  --output-dir outputs
```

Les tests couvrent à la fois la règle métier et la présence des livrables. Le premier test vérifie que les lignes annulées ne contaminent pas le total. Le second exécute le pipeline dans un dossier temporaire et contrôle le Parquet, la base DuckDB et le fichier KPI.

Terminal · Bash

```
python -m unittest discover -s tests -p 'test_*.py' -v
```

Sortie obtenue

Sortie / exemple

```
test_pipeline_writes_parquet_database_and_kpis ... ok  
test_transform_removes_unpaid_orders_and_adds_revenue ... ok  
  
-----  
Ran 2 tests in 0.0xxs  
  
OK
```

Commande de vérification

Terminal · Bash

```
find outputs -maxdepth 1 -type f -printf '%f\n' | sort
```

Sortie / exemple

analytics.duckdb
kpis_by_region.csv
run-summary.json
sales_curated.parquet

Exemples de sorties

Le résumé JSON permet à un ordonnanceur ou à un agent de vérifier la livraison sans parser tout le fichier KPI :

JSON

```
{
  "input_orders": 28,
  "input_customers": 8,
  "curated_rows": 24,
  "kpi_rows": 12,
  "total_revenue_eur": 16400.3,
  "outputs": [
    "sales_curated.parquet",
    "analytics.duckdb",
    "kpis_by_region.csv"
  ]
}
```

Extrait de outputs/kpis_by_region.csv

Mois	Région	Chiffre d'affaires (€)	Unités	Commandes
2026-01	Rabat	1 330,00	7	2
2026-01	Marrakech	1 085,00	5	2
2026-01	Tanger	979,60	5	2
2026-01	Casablanca	791,00	5	2
2026-02	Marrakech	2 015,00	9	2

Cet extrait permet de relier le fichier de sortie aux agrégations SQL produites par DuckDB.

Le fichier CSV peut ensuite être consommé par un dashboard, envoyé vers un stockage objet ou comparé à une livraison précédente. La mini-série s'arrête avant l'orchestration : elle fournit une unité ETL testable, pas un ordonnanceur de production.

Dépannage et diagnostics

ModuleNotFoundError: No module named polars ou duckdb — activez `.venv`, puis réinstallez `requirements.txt` avec le même interpréteur que celui qui exécute `python -m src.pipeline`.

ComputeError lors de la conversion de date — vérifiez le format `YYYY-MM-DD` et recherchez les lignes vides ou les valeurs non conformes avant `strptime`. Le mode `strict=True` est volontaire : une date silencieusement remplacée par `null` peut modifier les agrégations.

Le chiffre d'affaires est trop élevé — contrôlez l'unicité de `customers.customer_id` et la cardinalité `m:1` de la jointure. Une ligne client dupliquée répète chaque commande correspondante.

La base DuckDB contient d'anciennes lignes — la table est recrée avec `CREATE OR REPLACE`. Dans un projet réel, donnez à chaque livraison un répertoire de sortie ou une version immuable.

Le fichier Parquet n'est pas trouvé — exécutez l'étape Polars avant l'étape DuckDB et vérifiez `outputs/sales_curated.parquet`. Les chemins relatifs sont résolus depuis le dossier du projet.

Vérification finale

Depuis la racine de l'archive, les commandes suivantes doivent toutes réussir :

Terminal · Bash

```
python -m unittest discover -s tests -p 'test_*.py' -v
python -m src.pipeline
python - <<'PY'
import polars as pl
frame = pl.read_parquet("outputs/sales_curated.parquet")
assert frame.height == 24
assert round(frame["revenue_eur"].sum(), 2) == 16400.30
print("contrat Parquet: PASS")
PY
```

Sortie / exemple

```
OK
ETL terminé : 24 lignes curées, 16400.30 EUR, 12 agrégations
contrat Parquet: PASS
```

La mini-série est validée lorsque les résultats restent identiques après un nouveau lancement dans un dossier de sortie propre. La sortie attendue est déterministe pour les CSV fournis, tandis que les durées dépendent de la machine.

Limites et prolongements

Le pipeline utilise des fichiers locaux et une exécution en mémoire adaptée à un petit jeu de données. Pour un volume important, remplacez `read_csv` par `scan_csv`, ajoutez des contrôles de qualité par lot et mesurez la mémoire pendant `collect()`. La partition du Parquet peut également être étudiée si les requêtes filtrent souvent par mois ou région.

La base DuckDB est locale et reconstruite à chaque exécution. Une mise en production demanderait une stratégie d'écriture atomique, une rétention des livraisons, une table de contrôle des versions et un système d'orchestration. Ajoutez ensuite des tests de données, une alerte sur la variation du chiffre d'affaires et un catalogue de schéma.

Enfin, un agent ou un LLM peut demander une agrégation, mais la requête produite doit être validée, limitée aux tables autorisées et exécutée en lecture seule. Les règles de transformation et les montants restent du ressort du code ETL testé.

Références

- [Polars — documentation officielle](#)
- [Polars — guide des expressions et du lazy mode](#)
- [DuckDB — API Python officielle](#)
- [DuckDB — lecture et écriture de fichiers Parquet](#)
- [Polars — dépôt officiel des logos](#)
- [DuckDB — dépôt officiel des logos](#)