Clase 211 — Polars como alternativa moderna

Parte: 5 — Ingeniería de Datos · Fuente: docs Polars + Polars vs pandas benchmark. Duración estimada: 75 min. · Esta clase es complementaria a la Clase 008 (Polars básico, Parte 0). Acá el foco es producción: lazy API, streaming engine, Arrow zero-copy, integración con DuckDB/Parquet.

Clase 211 — Polars como alternativa moderna

> *Parte: 5 — Ingeniería de Datos · Fuente: docs Polars + Polars vs pandas benchmark. Duración estimada: 75 min. · Esta clase es complementaria a la Clase 008 (Polars básico, Parte 0). Acá el foco es producción: lazy API, streaming engine, Arrow zero-copy, integración con DuckDB/Parquet.*

Objetivo

Reemplazar pandas en pipelines productivos por Polars 1.x: 5-30× más rápido, multi-threaded por default, lazy API que optimiza la query antes de ejecutar, y streaming engine que procesa datasets mayores que RAM. Identificar los pocos casos donde pandas sigue ganando (ecosistema, statsmodels, sklearn pre-Arrow).

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Convertir scripts pandas a Polars (eager API: pl.DataFrame) y medir speedup.
- Usar lazy API (pl.scan_parquet + .collect()) para que el optimizador haga predicate pushdown + column pruning.
- Procesar archivos mayores que RAM con .collect(engine="streaming") (1.x rename de streaming=True).
- Hacer zero-copy interop con Arrow/DuckDB/pandas.
- Decidir Polars vs DuckDB vs pandas vs PySpark según tamaño + caso de uso.

Temas

#	Tema	Por qué importa
1	Eager vs Lazy API	El optimizador de queries es lo que hace g
2	Expressions: paralelización implícita	pl.col('x').mean() corre en todos los core
3	scan_parquet/scan_csv + predicate pushdow	Lee solo lo necesario del disco.
4	Streaming engine para datasets > RAM	Out-of-core sin Spark.
5	Arrow interop con DuckDB/pandas	Zero-copy → cero overhead.
6	when().then().otherwise() y over()	Window functions sin SQL.

Definiciones y características

- Polars: DataFrame library escrita en Rust, con Apache Arrow como memory model. Diseñada para columnar + paralelo + lazy.
- Eager API: pl.DataFrame(...). Similar a pandas — ejecuta cada operación.
- Lazy API: pl.scan_parquet("x.parquet").filter(...).select(...).collect(). Construye un plan; lo optimiza; ejecuta una vez. El default que querés en producción.
- Expression: una operación columnar (pl.col('x') * 2 + pl.col('y')). Reutilizable, componible, paralelizable.
- Streaming engine (1.x): ejecuta lazy queries en chunks, permitiendo procesar archivos mayores que RAM. .collect(engine="streaming"). Apto para la mayoría de las queries, no para todas (joins arbitrarios pueden no soportarse — chequear plan).
- Predicate pushdown: el optimizer empuja los WHERE lo más temprano posible. Si filtrás por

date='2024-01-15' después de un join, Polars filtra antes y reduce data.
- Column pruning: si solo select-ás 3 columnas, Polars no lee las otras del Parquet. pandas no hace esto.
- Arrow zero-copy: pl.from_pandas(df, rechunk=False) o df.to_arrow() → no se copia memoria, solo se cambia el "label" del buffer.

Dataset / recursos

- Mismo NYC Yellow Taxi de Clase 210 (parquet, ~150 MB/mes, ~10 GB/año).
- Librerías: polars>=1.5, pyarrow, opcional duckdb para integración.

Ejercicios

1. Eager vs Lazy benchmark: misma agregación con pl.read_parquet(...) (eager) y pl.scan_parquet(...).collect() (lazy). Compará tiempos. La diferencia es chica con dataset chico; aumenta dramáticamente con datasets >GB.
2. Pandas → Polars: tomá un script pandas existente, traducí a Polars. Medí speedup. Casos comunes: groupby().agg() → group_by().agg(), .apply → expressions.
3. Streaming: con un parquet de 5 GB (descargar 12 meses NYC Taxi), correr una agregación con .collect() y luego con .collect(engine="streaming"). Comparar RAM peak (memory_profiler).
4. Predicate pushdown explícito: pl.scan_parquet("data/").filter(pl.col("date") == "2024-01-15").select("fare").collect() vs pl.read_parquet("data/").filter(...).select(...). Mirá el plan con .explain().
5. Polars + DuckDB: hacer la query principal en Polars; pasar el resultado a DuckDB con con.from_arrow(df.to_arrow()) para hacer una consulta SQL compleja.

Homework verificable

1. Pipeline en Polars que: lee 12 meses NYC Taxi, filtra outliers, calcula avg fare por borough × hour, escribe parquet particionado.
2. Benchmark contra: pandas (si el dataset entra), DuckDB (con.execute(query).pl()), PySpark (Clase 210). Reportar tiempo + RAM peak.
3. Una query usando streaming engine que procesa >RAM (forzar bajando psutil.virtual_memory().available / 4 con docker-compose limit).
4. Una query con window functions (pl.col("fare").rolling_mean(window_size=7).over("borough")) que en pandas requeriría 20 LOC.
5. README con cuándo Polars vs cuándo otra cosa.

Criterio de aceptación: el alumno justifica con números (tiempo + RAM) por qué Polars es el default para su workload, y muestra al menos 1 caso donde DuckDB o pandas ganan.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
Polars not faster than pandas en mi caso	Probablemente: (1) dataset muy chico (<100
to_pandas() falla con pyarrow	Conflicto versiones pyarrow. Fix: pip inst
Lazy query no ejecuta nada	Olvidaste .collect(). Fix: cualquier query
MemoryError con streaming	Algunos operadores no soportan streaming (

API diferente a pandas confunde	groupby → group_by, value_counts → .value_
Date parsing devuelve Object o String	Schema inference falló. Fix: pl.scan_csv("

Preguntas frecuentes

¿Pandas se queda? ¿Migrar todo a Polars?

Pandas tiene 15 años de ecosistema: sklearn, statsmodels, plotly aceptan DataFrames nativos. Polars 1.x se integra (vía Arrow) pero algunas libs requieren conversión. Para pipelines de ETL/feature engineering: migrar a Polars. Para modeling/análisis exploratorio: pandas sigue cómodo.

¿Polars vs DuckDB?

Solapan, son complementarios.

- Polars: DataFrame API, más natural si pensás en código Python. Pipelines de transformación.
- DuckDB: SQL-first, OLAP optimized, ideal para análisis ad-hoc, queries complejas, joins masivos.

Combinables: Polars para transformación, DuckDB para queries finales analíticas.

¿Lazy o eager por default?

En producción: lazy siempre. En notebooks exploratorios: eager está bien (es como pandas). El mantra: "si vas a hacer más de 2 ops sobre el DataFrame, usá lazy".

¿Streaming engine es estable?

Polars 1.0 (julio 2024) lo marcó estable. Algunos operadores aún no lo soportan; el plan con .explain() dice qué nodos van streaming y cuáles in-memory.

¿Cómo manejo NaN/null en Polars vs pandas?

Polars distingue null (missing) de NaN (float). pandas los mezcla (problema histórico). En Polars: pl.col('x').is_null() vs pl.col('x').is_nan(). Más limpio pero requiere ajuste mental.

¿Multi-process o multi-thread?

Polars usa multi-thread (Rust + Rayon) — sin GIL. Mejor que pandas + multiprocessing. Para escalar a multi-machine: PySpark o Dask.

Referencias

- Polars docs — empezar por Getting started.
- Polars vs pandas migration guide.
- DB benchmark (DuckDB Labs) — Polars vs DuckDB vs pandas vs Spark en 5/50/500 GB.
- Modern Polars (book): https://kevinheavey.github.io/modern-polars/
- hyperscan Arrow ecosystem — zero-copy interop.

Siguiente clase

Clase 212 — Data warehouses: BigQuery, Snowflake, DuckDB

Apéndice: notebook (primer bloque)

Requiere: pip install polars pyarrow pandas duckdb.

```
import polars as pl, pandas as pd, time, tempfile, os
from pathlib import Path

WORK = Path(tempfile.gettempdir()) / 'polars_demo'
WORK.mkdir(exist_ok=True)
print('Polars version:', pl.__version__)
```

Archivos complementarios

- notebook.ipynb