Clase 210 — PySpark para datasets grandes

Parte: 5 — Ingeniería de Datos · Fuente: Chambers & Zaharia Spark: The Definitive Guide (O'Reilly, 2018) + PySpark docs 3.5+. Duración estimada: 90 min.

Clase 210 — PySpark para datasets grandes

> *Parte: 5 — Ingeniería de Datos · Fuente: Chambers & Zaharia Spark: The Definitive Guide (O'Reilly, 2018) + PySpark docs 3.5+. Duración estimada: 90 min.*

Objetivo

Procesar datasets que no entran en RAM con PySpark 3.5+: DataFrames con lazy evaluation, Spark SQL, particionado, joins eficientes (broadcast vs shuffle), y entender cuándo Spark gana vs pandas/Polars (>10 GB) y cuándo pierde (<1 GB, dev local).

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Crear una SparkSession (local o cluster) y leer Parquet/CSV/JSON con schema inference o explícito.
- Diferenciar transformations (lazy: select, filter, groupBy) de actions (eager: count, collect, show, write).
- Optimizar joins: broadcast join (tabla chica × tabla grande) vs sort-merge join (dos tablas grandes).
- Particionar correctamente (partitionBy("date") al escribir, evitar partitionBy con cardinalidad alta).
- Diagnosticar performance con Spark UI: stages, shuffle data, skew.

Temas

#	Tema	Por qué importa
1	RDD vs DataFrame vs SQL — 3 APIs	DataFrame es el default; SQL si tu equipo
2	Lazy evaluation + DAG de ejecución	Optimización que pandas no tiene.
3	Particionado: al leer, al escribir, en mem	Donde se gana o pierde 10× perf.
4	Joins: broadcast vs shuffle vs sort-merge	El bottleneck más común.
5	Caching / persist	Cuándo materializar; cuándo NO.
6	Spark UI: stages, shuffle, skew	Diagnosis sin esto = ciego.

Definiciones y características

- SparkSession: entrypoint a Spark. SparkSession.builder.appName("x").getOrCreate().
- DataFrame: tabla distribuida (RDD de Rows con schema). API similar a pandas pero lazy.
- Transformation: operación que devuelve otro DataFrame sin ejecutar (df.filter(...), df.select(...)). Construye el DAG.
- Action: operación que dispara ejecución (df.count(), df.show(), df.write...). Spark optimiza el DAG entero y luego ejecuta.
- Partition: chunk del DataFrame que vive en un task. Por default ~200 (config spark.sql.shuffle.partitions). Demasiadas → overhead; pocas → no paraleliza.
- Broadcast join: si una tabla es chica (<10 MB default), Spark la copia a todos los executors → join sin shuffle. Usar broadcast() hint para forzar.
- Sort-merge join: dos tablas grandes → shuffle ambas por la key + sort + merge. Costoso pero general.
- Skew: cuando una key tiene 10× más filas que el promedio → un task tarda 10×. Mitigación: salting,

AQE (Adaptive Query Execution).

- AQE (Adaptive Query Execution, Spark 3+): runtime optimization — re-particiona, coalesce shuffle, handle skew. Habilitado por default desde 3.2.
- Catalyst: optimizer de queries SQL/DataFrame. Reescribe el plan lógico (predicate pushdown, column pruning) antes de ejecutar.

Dataset / recursos

- Local mode: pyspark.SparkSession.builder.master("local[*]") — usa todos los cores.
- Dataset: NYC TLC Yellow Taxi 2024 (parquet, ~10 GB) — clásico para Spark demos.
- Librerías: pyspark>=3.5, pyarrow.

Ejercicios

1. Spark session local: spark = SparkSession.builder.master("local[4]").appName("demo").getOrCreate(). Cargá un parquet, mostrá schema con df.printSchema().
2. Lazy vs eager: df2 = df.filter(...).select(...) (rápido, no ejecuta). df2.count() (lento, ejecuta). Mirá en Spark UI (localhost:4040) los stages.
3. Broadcast join: cargá taxi (10 GB) y zones (1 KB). Hacé taxi.join(broadcast(zones), "zone_id"). Compará con taxi.join(zones, "zone_id") sin hint — debería ser igual gracias a AQE auto-broadcast.
4. Particionado al escribir: df.write.partitionBy("date").parquet("out/"). Verificá estructura out/date=2024-01-01/part-*.parquet. Lecturas con filtro WHERE date='2024-01-01' solo leen ese subdirectorio.
5. Skew: simulá una key skewed (90% rows con user_id=1). Hacé groupBy → observá UI: 1 task tarda 90% del tiempo. Mitigá con salting: agregar columna random salt = (rand() * 10).cast("int"), group por (user_id, salt), después sumar.

Homework verificable

Notebook + reporte:

1. Pipeline PySpark que: lee NYC Taxi (parquet), filtra outliers, agrega por pickup_zone y hour, escribe parquet particionado por pickup_date.
2. Comparación de performance: misma agregación en (a) pandas (si entra), (b) Polars, (c) PySpark. Reportar tiempo y RAM peak.
3. Identificar 1 stage skewed en Spark UI y aplicar salting para mitigar — mostrar antes/después.
4. Output final con .write.bucketBy(20, "zone_id").parquet(...) para acelerar joins futuros.
5. README con cuándo elegir cada uno: criterios objetivos (tamaño, latencia, equipo).

Criterio de aceptación: el alumno justifica con números por qué Spark gana en su dataset y muestra una optimización (broadcast/salting/partitioning) con impacto medido.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
OutOfMemoryError: Java heap space	Executor sin RAM. Fix: --driver-memory 4g
df.show() tarda 5 minutos	Estás computando todo el DataFrame para mo
Job tarda 30 min en una agregación de 1 GB	Sospechosos: shuffle excesivo, default par

Caused by: BindException: Address already	Spark UI en :4040 conflicto. Fix: .config(
to_pandas() falla con OOM	Estás trayendo TODO el DataFrame al driver
Resultados distintos entre runs	UDFs no-determinísticos, o monotonically_i
partitionBy("user_id") con 1M users crea 1	High-cardinality partitioning es anti-patt

Preguntas frecuentes

¿PySpark, pandas, Polars o DuckDB?

Decisión por tamaño + uso:

- <1 GB local: pandas o Polars (Polars 5-10× más rápido).
- 1-50 GB single machine: Polars o DuckDB.
- >50 GB single machine: Polars streaming, DuckDB out-of-core, o PySpark local.
- >500 GB y/o cluster: PySpark.

¿PySpark en local o necesito cluster?

master("local[*]") corre Spark en tu laptop usando todos los cores — perfecto para dev/test con datasets hasta unos GB. Para producción TB: Databricks, EMR, GCP Dataproc, Azure Synapse, o K8s con Spark Operator.

¿UDF Python vs Spark SQL functions?

SQL functions (F.col, F.when, F.regexp_extract) son mucho más rápidas — corren en JVM. UDFs Python serializan a Python por row (lento). Si no podés evitarlo: Pandas UDFs (vectorizadas, ~10× UDF normal).

¿Por qué df.cache() no aceleró?

Cache es lazy también. Tenés que disparar una action (df.count()) para materializarlo. Después las siguientes actions reutilizan el cache. Y: si tu dataset no entra en memoria, cache no ayuda — usá df.persist(StorageLevel.DISK_ONLY).

¿Spark 3 vs 4 vs Databricks Runtime?

Spark 3.5 es el estándar open-source en 2026. Spark 4 trae Spark Connect (cliente liviano vs server JVM), Variant type, mejoras en streaming. Databricks Runtime es Spark + extensiones propietarias (Photon engine 2-5× más rápido). Para empezar: Spark 3.5 open-source.

¿Cuándo usar Spark SQL vs DataFrame API?

Equivalentes en performance (mismo Catalyst). DataFrame API: refactor-friendly, type hints. SQL: mejor si tu equipo es SQL-first o querés migrar de un warehouse. Mezclables: spark.sql("SELECT * FROM tbl").filter(...).

Referencias

- Chambers, B. & Zaharia, M. Spark: The Definitive Guide (O'Reilly, 2018) — algo viejo pero los conceptos siguen.
- PySpark docs 3.5+ — empezar por Quickstart: DataFrame.
- Adaptive Query Execution deep dive.
- Spark Performance Tuning (official).
- pyspark-stubs — type hints (incluidos en 3.4+).

Siguiente clase

Clase 211 — Polars como alternativa moderna

Apéndice: notebook (primer bloque)

Requiere: pip install pyspark==3.5.3. Usaremos modo local con todos los cores.

```
import os, tempfile, shutil
from pathlib import Path
WORK = Path(tempfile.gettempdir()) / 'spark_demo'
if WORK.exists(): shutil.rmtree(WORK)
WORK.mkdir(); os.chdir(WORK)

from pyspark.sql import SparkSession, functions as F
from pyspark.sql.functions import broadcast, col, rand, when

spark = (SparkSession.builder
        .master('local[*]')
        .appName('clase210')
        .config('spark.sql.shuffle.partitions', '8')   # bajo para dataset chico
        .config('spark.ui.port', '4042')
        .getOrCreate())
print('Spark version:', spark.version, '| cores:', spark.sparkContext.defaultParallelism)
```

Archivos complementarios

- notebook.ipynb