# Parte 5 — Ingeniería de Datos

8 clases · Parte 5 del programa

# Parte 5 — Ingeniería de Datos

> *8 clases · bundle consolidado del currículo v3.*

## Índice de clases



---

## Clase 208 — Clase 208 — Pipelines ETL/ELT con Airflow

> *Parte: 5 — Ingeniería de Datos · Fuente: Reis & Housley Fundamentals of Data Engineering (O'Reilly, 2022) cap. 8 + Airflow docs 2.10+. Duración estimada: 80 min.*

#### Objetivo

Escribir DAGs de Airflow 2.x con la API moderna (TaskFlow + @dag/@task decorators), entender la diferencia entre ETL (transform antes de cargar) y ELT (cargar al warehouse y transformar ahí), y orquestar un pipeline extract → load → transform → notify con retries, SLAs y backfill.

#### Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Escribir un DAG con TaskFlow API (@dag, @task) y entender el grafo resultante.
- Diferenciar ETL (clásico) de ELT (moderno, warehouse-first) y elegir según contexto.
- Configurar retries, SLAs, trigger rules (all_success, one_failed, none_failed), y XComs.
- Hacer backfill (airflow dags backfill) para reprocesar fechas históricas sin duplicar.
- Diagnosticar Task stuck in queued, Worker died, DAG not appearing con airflow dags list, logs, y airflow.cfg.

#### Temas

| # | Tema | Por qué importa |
|---|---|---|
| 1 | ETL vs ELT — cuándo cada uno | El warehouse moderno cambió el default. |
| 2 | DAG = grafo dirigido acíclico | Vocabulario fundacional. |
| 3 | TaskFlow API vs Operators clásicos | Código más limpio en Airflow 2.x. |
| 4 | XComs — pasar data entre tasks | Su uso correcto y sus límites (no para GBs |
| 5 | Schedule + catchup + backfill | Reprocesar histórico sin duplicar. |

| | | |
|---|---|---|
| 6 | Sensors, hooks, providers | Esperar eventos externos, conectar a siste |

#### Definiciones y características

- ETL (Extract-Transform-Load): transformás en Python/Spark antes de cargar al warehouse. Patrón clásico cuando el warehouse era caro.
- ELT (Extract-Load-Transform): cargás raw al warehouse y transformás con SQL ahí. Patrón moderno (BigQuery/Snowflake/DuckDB son baratos para compute SQL).
- DAG: grafo de tareas con dependencias. En Airflow se define con @dag decorator + @task per tarea.
- TaskFlow API: introducida en Airflow 2.0. Devuelve valores de @task que se vuelven inputs de otras @task — Airflow infiere XComs automático.
- XCom (Cross-Communication): mecanismo para pasar pequeños valores entre tasks. Default backend: metadata DB. No usar para GBs — pasá referencias (paths S3) en su lugar.
- Catchup: si un DAG con schedule='@daily' se prende un lunes, ¿corre todos los días desde su start_date? Si catchup=True: sí (reprocesa retroactivo). Si False: arranca desde hoy. Default True — fuente de sorpresas caras.
- Backfill: reprocesar fechas específicas. airflow dags backfill --start-date 2026-06-01 --end-date 2026-06-15 my_dag.
- Sensor: tarea que espera un evento externo (archivo en S3, fila en DB) con polling. reschedule mode evita ocupar worker slot mientras espera.
- Hook: wrapper sobre un cliente externo (S3Hook, PostgresHook). Lee credenciales desde Connections UI.
- Provider: paquete con operators/hooks/sensors para una integración (apache-airflow-providers-amazon, apache-airflow-providers-google).

#### Dataset / recursos

- Stack ejemplo: Airflow 2.10+ con docker-compose oficial.
- Pipeline target: extrae CSV → carga a DuckDB → transforma con SQL → publica métricas a Slack.
- Librerías: apache-airflow>=2.10, duckdb, pandas.

#### Ejercicios

1. DAG mínimo TaskFlow: 3 tasks: extract (descarga CSV), transform (limpia con pandas), load (escribe a DuckDB). Ver el grafo en /graph.
2. Schedule + catchup: schedule='@daily', start_date=days_ago(7), catchup=True. Verificá que Airflow crea 7 runs históricos. Cambiar a catchup=False y observar diferencia.
3. Retries + SLA: agregá retries=3, retry_delay=timedelta(minutes=2), sla=timedelta(minutes=10) al transform. Simulá falla con raise Exception("flaky") y observá reintento.
4. XCom: extract devuelve un dict pequeño (filename + row count). transform lo recibe como argumento (TaskFlow autoinjecta). Verificá en la UI tab "XCom".
5. Backfill: airflow dags backfill --start-date 2026-06-01 --end-date 2026-06-05 my_dag. Confirmá que se ejecutan los 5 días sin duplicar (gracias a idempotencia con execution_date como key).

#### Homework verificable

Repo con docker-compose Airflow + DAG que:

1. Corre cada hora (schedule='@hourly').
2. Extrae data de una API pública (ej. CoinGecko BTC price), la carga a DuckDB.
3. Transforma con SQL (run_sql task usando DuckDBHook).
4. Calcula métrica (avg price 24h) y la postea a Slack via SlackWebhookOperator.
5. SLA de 5 min en extract, retries=3.

6. README con docker-compose up, comandos kubectl, captura de la UI.

Criterio de aceptación: el DAG corre 24 veces seguidas (1 día) sin fallar, los datos en DuckDB son idempotentes (re-run para misma hora no duplica), Slack recibe el mensaje cada hora.

#### Errores comunes

| Síntoma / mensaje | Causa y cómo arreglar |
|---|---|
| DAG no aparece en la UI | Sintaxis error en el archivo. Fix: python |
| Catchup explota: 365 runs creados al arran | catchup=True (default) con start_date de h |
| Task stuck in queued forever | Worker no levantó, o pool agotado. Fix: ai |
| XCom serialization error con DataFrame | XCom default es JSON; un DataFrame no es J |
| start_date en el futuro hace que no corra | El scheduler ignora DAGs con start_date > |
| Variables de entorno no llegan al worker | airflow.cfg o Docker compose con env vars |

#### Preguntas frecuentes

¿Airflow, Prefect, Dagster, Mage — cuál elijo en 2026?

- Airflow: estándar de la industria, max madurez, syntax verbosa.
- Prefect 3: API Python moderna, hybrid execution (Clase 209).
- Dagster: data-aware (assets), mejor lineage.
- Mage: notebook-first.

Para empresas grandes / equipos data engineering serios: Airflow. Para equipos chicos / ML-focused: Prefect o Dagster son más rápidos.

¿ETL o ELT?

ELT cuando: warehouse moderno (BigQuery/Snowflake), data analyst con SQL, transformaciones cambian seguido. ETL cuando: data muy sensible (no podés cargar raw al warehouse), transformaciones complejas (geo, ML inference), warehouse caro. La industria converge a ELT con dbt para SQL transforms post-load.

¿XCom para pasar un DataFrame de 500 MB?

No. XCom default está en la metadata DB (Postgres/MySQL) — vas a saturarla. Pasá un path S3 / GCS por XCom, y cada task lee/escribe del storage. O configurá Custom XCom Backend con S3.

¿Tengo que aprender los Operators clásicos (PythonOperator, BashOperator)?

Para mantener DAGs viejos: sí. Para nuevos: TaskFlow API es mejor. Pero algunos casos (operators provider-specific como BigQueryInsertJobOperator) siguen siendo más limpios sin TaskFlow.

¿Cómo testeo un DAG?

Tres niveles: (1) import test: python my_dag.py no rompe. (2) unit test de tasks: extraer la lógica a funciones puras + tests. (3) integration: airflow dags test my_dag 2026-06-01 corre el DAG sin scheduler.

¿Airflow corre el Python de mi DAG en cada heartbeat?

Sí — el scheduler parsea todos los DAGs cada min_file_process_interval segundos. No pongas código pesado al top level (requests.get(...) en import time es un anti-pattern). Todo lo costoso va dentro de tasks.

#### Referencias

- Reis, J. & Housley, M. Fundamentals of Data Engineering (O'Reilly, 2022), cap. 8 — Queries, Modeling,

and Transformation.
- Airflow docs 2.x — TaskFlow API.
- Awesome Apache Airflow.
- dbt-core — el complemento natural para ELT.
- Designing Data-Intensive Applications (Kleppmann, 2017) — fundamentos.

#### Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.
- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

---

## Clase 209 — Clase 209 — Pipelines con Prefect o Dagster

> *Parte: 5 — Ingeniería de Datos · Fuente: Prefect 3 docs + Dagster docs + Reis & Housley cap. 8. Duración estimada: 75 min.*

#### Objetivo

Construir el mismo pipeline de Clase 208 con Prefect 3 (API Python moderna, hybrid execution) y con Dagster (asset-oriented, mejor lineage). Entender qué problemas resuelven mejor que Airflow y cuándo elegir cada uno.

#### Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Escribir un flow Prefect con @flow/@task, deployments, work pools y workers.
- Definir assets Dagster (@asset) y entender la diferencia entre "task-oriented" (Airflow/Prefect) y "asset-oriented" (Dagster).
- Configurar hybrid execution Prefect: control plane en cloud, workers en tu infra (sin enviar data sensible).
- Usar Dagster's UI para ver lineage automático: qué asset depende de qué, cuándo se materializó cada uno.
- Decidir Airflow vs Prefect vs Dagster según contexto (equipo, escala, tipo de pipeline).

#### Temas

| # | Tema | Por qué importa |
|---|---|---|
| 1 | Prefect 3: flows, tasks, deployments | Reemplaza DAGs con código Python idiomátic |
| 2 | Work pools + workers | Hybrid execution: control en cloud, comput |
| 3 | Dagster: asset-oriented vs task-oriented | Lineage automático, mejor para data produc |
| 4 | Software-defined assets (SDA) | Cada asset es código + metadata + checks. |
| 5 | Scheduling: cron, interval, event-driven | Las 3 formas de disparar. |
| 6 | Cuándo migrar de Airflow | Costo de migración vs beneficio. |

#### Definiciones y características

- Prefect Flow: equivalente al DAG. Decorador @flow. Puede invocar tasks (@task) y sub-flows.
- Prefect Task: unidad de trabajo. Tiene retries, cache, timeout configurables. Diferente a Airflow: las tasks pueden tener loops/condicionales sin XCom hackery.

- Deployment: la receta de "cómo y cuándo correr este flow" (schedule, parameters, infra). Prefect 3 los empaqueta como código (flow.deploy(...)).
- Work pool: cola lógica donde los deployments mandan runs. Los workers la consumen.
- Worker: proceso que corre los runs. Puede vivir en tu laptop, K8s, ECS, etc. Cloud variant: control plane managed; workers tuyos (hybrid).
- Dagster Asset: objeto persistente versionado (una tabla, un modelo, un dashboard). Definido con @asset. Las dependencias entre assets se infieren del código.
- Op (Dagster): equivalente más cercano a "task" — unidad atómica. Los assets generalmente componen ops.
- Materializar: ejecutar el código que produce el asset (re-genera el output). Dagster trackea cuándo se materializó cada asset por última vez.
- Sensor (Dagster/Prefect): trigger basado en eventos (nuevo archivo, mensaje en queue) en vez de schedule.

#### Dataset / recursos

- Mismo pipeline de Clase 208 (BTC price), implementado dos veces.
- Librerías: prefect>=3, dagster>=1.7, duckdb, pandas, requests.

#### Ejercicios

1. Prefect flow: copiá la lógica del DAG Airflow al patrón Prefect: @flow def btc_pipeline(): notify(transform(load(extract()))). Corré python btc.py directo (no necesita scheduler).
2. Deployment Prefect: flow.serve(name="btc-hourly", cron="0 "). Dejá corriendo, observá ejecuciones programadas en localhost:4200.
3. Dagster assets: convertí las funciones a @asset def btc_price(), @asset def daily_avg(btc_price). Dagster infiere daily_avg depende de btc_price. UI muestra el grafo.
4. Materializar: en Dagster UI, click "Materialize" sobre btc_price. Solo se ejecuta ese asset; daily_avg queda "stale" hasta que se materialice también.
5. Comparativa: mismo pipeline en Airflow + Prefect + Dagster. Compará LOC, claridad, UI, velocidad de feedback dev.

#### Homework verificable

Repo con el mismo pipeline implementado en los 3 frameworks:

1. airflow/dags/btc.py (de Clase 208).
2. prefect/btc.py con @flow/@task y deployment programado.
3. dagster/btc.py con @asset definitions y un Definitions object.
4. README comparativo: LOC, complejidad de setup, calidad de UI, lineage support, cuándo elegir cada uno.
5. Bonus: GitHub Actions que corre los 3 en CI y verifica que producen el mismo output.

Criterio de aceptación: los 3 pipelines producen idénticos resultados sobre el mismo input; el README compara honestamente fortalezas/debilidades.

#### Errores comunes

| Síntoma / mensaje | Causa y cómo arreglar |
|---|---|
| prefect server start falla | Otro proceso en :4200. Fix: --port 4201. |
| Deployment no se ejecuta automáticamente | Workers no están corriendo o pool mal conf |
| Dagster asset materialization "stale" pero | Auto-materialization no está habilitada. F |
| Re-import circular entre @assets | Dagster intenta resolver dependencias en i |

| | |
|---|---|
| Performance lento en Prefect con muchas ta | El backend default está en SQLite. Fix: pa |
| Mixed Airflow + Prefect + Dagster en el mi | Cada uno tiene su propio ecosistema. Fix: |

#### Preguntas frecuentes

Airflow vs Prefect vs Dagster en una frase

- Airflow: el camión de carga industrial — feo pero llega.
- Prefect: el Tesla — UI moderna, código limpio, hybrid execution.
- Dagster: la BMW — lineage hermoso, ideal cuando pensás en data products.

¿Vale la pena migrar desde Airflow?

Calcular: (costo de migrar N DAGs) vs (ahorro mensual en mantenimiento + horas dev). Si Airflow funciona y nadie está sufriendo: no. Si nuevos pipelines: empezar con Prefect/Dagster, dejar los viejos donde están.

¿Prefect Cloud o self-hosted?

Cloud: control plane managed, free tier generoso, hybrid execution (workers tuyos). Self-hosted: prefect server start corre todo local — para dev. Para prod: Cloud es mucho más práctico.

¿Dagster's asset model es overkill para pipelines simples?

Para 3-5 tareas en cascada: sí, Prefect es más simple. Para data warehouse con 100+ tablas modeladas: Dagster brilla (lineage, freshness, partition awareness).

¿Cómo manejo secrets en Prefect/Dagster?

- Prefect: Secret.load("my-key").get() desde blocks (cifrados en backend).
- Dagster: EnvVar("MY_SECRET") o resources con ConfigurableResource.

Ambos integran con AWS Secrets Manager / GCP Secret Manager / Vault.

¿Puedo usar Dagster con dbt?

Sí — dagster-dbt carga modelos dbt como assets Dagster automático. Lineage atraviesa Python → dbt → SQL → tablas. Es el sweet spot del stack moderno.

#### Referencias

- Prefect 3 docs — empezar por Quickstart.
- Dagster docs — Concepts → Assets.
- Reis & Housley Fundamentals of Data Engineering (O'Reilly, 2022) cap. 8.
- Prefect vs Airflow (Prefect, 2024) — sesgado pero útil.
- dagster-dbt integration — el combo más usado.

#### Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.
- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

---

## Clase 210 — 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+).


#### Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.
- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

---

## Clase 211 — 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.

#### Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.
- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

---

## Clase 212 — Clase 212 — Data warehouses: BigQuery, Snowflake, DuckDB

> *Parte: 5 — Ingeniería de Datos · Fuente: Reis & Housley Fundamentals of Data Engineering (O'Reilly, 2022) cap. 6 + 9 + docs oficiales. Duración estimada: 80 min.*

#### Objetivo

Consultar y operar 3 data warehouses modernos desde Python: BigQuery (GCP, serverless, separación compute/storage), Snowflake (multi-cloud, virtual warehouses, time travel), DuckDB (embedded, OLAP local, no requiere server). Decidir cuál usar según escala, presupuesto y latencia.

#### Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Conectar a BigQuery (google-cloud-bigquery), Snowflake (snowflake-connector-python), DuckDB (duckdb) y ejecutar queries.
- Diseñar tablas con particionado (BigQuery: PARTITION BY date_field) y clustering (BigQuery, Snowflake) para reducir costos y latencia.
- Usar COPY INTO (Snowflake) y LOAD DATA (BigQuery) para bulk ingest desde S3/GCS.
- Aprovechar time travel (SELECT ... AT(TIMESTAMP => ...) Snowflake) para auditar / recuperar.
- Decidir DW: BigQuery (serverless, GCP-first), Snowflake (multi-cloud, separation, sharing), DuckDB (local/embedded), Redshift (legacy AWS).

#### Temas

| # | Tema | Por qué importa |
|---|---|---|
| 1 | Compute/storage separation | Por qué los DW modernos son baratos: solo |
| 2 | Particionado vs clustering | Reducir bytes leídos. |
| 3 | BigQuery: SQL standard + UDF + ML | El más simple para empezar. |
| 4 | Snowflake: virtual warehouses, time travel | Features únicos. |
| 5 | DuckDB: el DW que entra en pip install | Análisis local, ETL liviano, embed en app. |
| 6 | Costo: cómo NO gastar miles | SELECT * sin partition filter = bankruptcy |

#### Definiciones y características

- Data warehouse: DB OLAP optimizada para queries analíticas (agregados sobre millones/billones de filas). Diferente de OLTP (Postgres, MySQL) optimizada para transacciones.
- Compute/storage separation: storage en blob (S3/GCS/Azure) — barato y elástico. Compute (virtual warehouse / slot) se enciende para queries. Pagás storage barato + compute por uso.
- Particionado: la tabla está dividida por una columna (típicamente date). Query con WHERE date='2024-01-15' lee solo esa partición → menos bytes → menos $.
- Clustering (BigQuery) / Cluster keys (Snowflake): orden físico dentro de cada partición según N columnas. Mejora queries con WHERE col_clustered.
- Slot / Virtual Warehouse: unidad de compute que paralela tu query. BigQuery: slots compartidos o reservados. Snowflake: tamaño de VW (XS/S/M/L/XL...) en $/hora.
- Time travel (Snowflake): consultar el estado pasado de una tabla (SELECT ... AT(TIMESTAMP => '2026-06-15 00:00:00')). Retención 1-90 días según tier.
- Data sharing (Snowflake): compartir tablas con otra cuenta Snowflake sin copiar data (live read).
- DuckDB: DB OLAP embedded (como SQLite pero columnar + vectorizada). No requiere server. Lee/escribe Parquet/CSV/Arrow nativo. Ideal para dev local + ETL liviano + análisis ad-hoc.

#### Dataset / recursos

- BigQuery: bigquery-public-data.new_york_taxi_trips.tlc_yellow_trips_2018 (GB-scale, gratis para query con free tier).
- DuckDB: local con parquets de NYC Taxi.
- Snowflake: trial 30 días con $400 de crédito.
- Librerías: duckdb>=1.0, google-cloud-bigquery, snowflake-connector-python.

#### Ejercicios

1. DuckDB local: con = duckdb.connect("warehouse.duckdb"). Cargá un parquet con CREATE TABLE trips AS SELECT  FROM 'trips.parquet'. Hacé SELECT COUNT(), AVG(fare) FROM trips. Compará tiempo vs Polars (Clase 211).
2. BigQuery query: from google.cloud import bigquery; client = bigquery.Client(project="..."). client.query("SELECT borough, COUNT(*) FROM bigquery-public-data.new_york_taxi_trips.tlc_yellow_trips_2018 GROUP BY borough"). Crítico: usar LIMIT en exploración, no escanear 100 GB sin querer.
3. Particionado en BQ: crear tabla mi_proyecto.dataset.trips_partitioned con PARTITION BY DATE(pickup_datetime). Query con WHERE DATE(pickup_datetime) = '2018-01-15' → mostrá "bytes processed" con/sin partition filter.
4. Snowflake time travel: CREATE TABLE x AS SELECT .... Insertá data. DELETE FROM x WHERE .... SELECT * FROM x AT(OFFSET => -60) (60s atrás) — los datos vuelven.
5. DuckDB queryando S3 directo: con.execute("SELECT  FROM 's3://bucket/path/.parquet' LIMIT 10") sin descargar nada — DuckDB lee remoto con HTTPFS.

#### Homework verificable

Proyecto con:

1. Pipeline que extrae datos de una API → carga a un DW (elegir 1: BQ free tier, DuckDB, Snowflake trial) → corre 5 queries analíticas.
2. Comparativa de costo: misma query escaneando (a) tabla sin particionar (5 GB), (b) tabla particionada con filter (50 MB). Reportar $ estimado.
3. Tabla con clustering / cluster keys sobre 2 columnas que mejoran una query frecuente. Mostrar mejora de performance.
4. Setup de service account con permisos mínimos (BQ Data Viewer, no Admin).
5. README explicando cuál DW elegirías para 3 escenarios: startup 5 dev, empresa multi-cloud, análisis personal local.

Criterio de aceptación: el alumno demuestra entender el modelo de costo (bytes scanned × $/TB) y propone arquitectura razonable para cada escenario.

#### Errores comunes

| Síntoma / mensaje | Causa y cómo arreglar |
|---|---|
| Factura BigQuery $500 inesperados | Alguien hizo SELECT  sin partition filter |
| Query Snowflake lenta — escalar el warehou | Posible, pero antes: revisar el plan (EXPL |
| DuckDB OOM en query agregada | Default memory limit (80% RAM). Fix: SET m |
| Permission denied BigQuery | El service account no tiene rol. Fix: en G |
| Snowflake Statement timeout | Tu VW es chico. Fix: subir tamaño temporal |
| JOIN lento entre dos tablas grandes | Falta cluster key compartida. Fix: cluster |
| Particionado por columna high-cardinality | BQ permite max 4000 particiones. Fix: part |

#### Preguntas frecuentes

¿BQ vs Snowflake vs Redshift vs Databricks SQL?

- BigQuery: serverless puro, ideal en GCP, modelo "pay per query".
- Snowflake: separación compute/storage explícita, time travel, data sharing nativo. Multi-cloud.
- Redshift: el viejo (2012). AWS-native. RA3 instances son competitivos pero menos elegantes.
- Databricks SQL (sobre lakehouse): si ya tenés Spark en Databricks, integrado.

Para empezar: BigQuery (GCP) o Snowflake (multi-cloud).

¿DuckDB como warehouse de producción?

Para una sola máquina y team chico: sí (hasta cientos de GB). Para múltiples usuarios escribiendo concurrente y escala TB+: no — usá Snowflake/BQ. MotherDuck ofrece DuckDB managed cloud con escalado.

¿Cómo evito quemar $500 sin querer en BQ?

(1) Project Settings → Quotas → Query usage limit per day. (2) Linter pre-commit que rechaza SELECT * sin partition. (3) Crear materialized views para queries repetidas — pagás 1 vez. (4) dry_run=True antes de ejecutar.

¿DELETE en un DW columnar es caro?

Sí. Las tablas columnares no están optimizadas para DELETE individual. Patrón: marcar como deleted_at

(soft delete) o reescribir partition (INSERT OVERWRITE). Snowflake/BQ lo manejan razonablemente; Redshift sufre.

¿Cómo cargo 1 TB a un warehouse?

(1) Subí archivos a blob storage (S3/GCS). (2) Usá comando bulk del DW: LOAD DATA (BQ), COPY INTO (Snowflake), COPY (Redshift). NUNCA INSERT ... VALUES por row.

¿Costo storage es relevante?

En 2026: ~$23/TB/mes en BQ/Snowflake. Para 10 TB = $230/mes. Para 1 PB: $23K/mes — empieza a importar. Para datasets enormes: lifecycle policies (mover a Glacier/Coldline después de N días).

#### Referencias

- Reis & Housley Fundamentals of Data Engineering (O'Reilly, 2022) cap. 6 (storage) y 9 (queries).
- BigQuery docs — empezar por Quickstarts.
- Snowflake docs — Quickstarts + Cost optimization.
- DuckDB docs + MotherDuck (DuckDB managed).
- Designing Data-Intensive Applications (Kleppmann, 2017) — fundamentos OLAP.

#### Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.
- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

---

## Clase 213 — Clase 213 — Streaming intro: Kafka, Kinesis

> *Parte: 5 — Ingeniería de Datos · Fuente: Kreps, Building a Real-Time Data Pipeline + Narkhede, Shapira, Palino Kafka: The Definitive Guide (O'Reilly, 2ª ed., 2021) + Reis & Housley cap. 7. Duración estimada: 85 min.*

#### Objetivo

Entender el modelo streaming vs batch (Clase 208), producir y consumir mensajes en Kafka (con confluent-kafka o kafka-python), comparar contra AWS Kinesis Data Streams (managed equivalent), y reconocer los 3 problemas clásicos de streaming: exactly-once, out-of-order events, backpressure.

#### Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Diferenciar batch (datos llegan en bloques) de streaming (datos llegan continuos, baja latencia).
- Producir mensajes a un topic Kafka con keys (garantiza orden por key, distribuye load).
- Consumir con consumer groups: partitions distribuidas entre consumers, offset management.
- Diseñar para at-least-once (default razonable) y entender qué requiere exactly-once (transactional API).
- Decidir Kafka (self-hosted o Confluent Cloud) vs Kinesis (AWS) vs Pub/Sub (GCP) vs Event Hubs (Azure).

#### Temas

| # | Tema | Por qué importa |
|---|---|---|

| | | |
|---|---|---|
| 1 | Batch vs streaming — el espectro real | Spectrum: batch → micro-batch → streaming |
| 2 | Kafka model: topic, partition, offset | Vocabulario obligatorio. |
| 3 | Producer: keys, acks, idempotence | Garantías que querés desde el día 1. |
| 4 | Consumer groups + rebalancing | Cómo escalan los consumers. |
| 5 | Delivery semantics: at-most/at-least/exact | Trade-offs reales con código. |
| 6 | Kinesis comparison | Mismo modelo, vendor-specific. |

#### Definiciones y características

- Topic: log durable, particionado, append-only. Análogo a una "tabla" en streaming.
- Partition: subset ordenado del topic. Cada partition tiene un único consumer dentro de un consumer group. Más partitions → más paralelismo.
- Offset: posición del consumer dentro de una partition. Persistido en Kafka (__consumer_offsets topic) o externamente.
- Producer: escribe mensajes a un topic. Puede especificar key (decide partition vía hash → mensajes con misma key van a la misma partition → orden garantizado por key).
- Consumer: lee mensajes de un topic en un consumer group. Si un consumer muere, otro toma sus partitions (rebalancing).
- Consumer group: lógicamente "un consumidor". 3 consumers en el mismo grupo se dividen las partitions; 3 grupos distintos cada uno ve todos los mensajes.
- At-most-once: mensaje se pierde si falla algo. acks=0.
- At-least-once: mensaje puede llegar duplicado. acks=all + enable_auto_commit=False + commit después de procesar. Default razonable.
- Exactly-once: requiere idempotent producer (enable.idempotence=true) + transactional API (Kafka Streams o Flink). Caro, complejo, no siempre necesario.
- Backpressure: cuando consumer no procesa tan rápido como producer escribe → lag crece. Acción: escalar consumers, o aplicar throttling al producer.
- Kinesis Data Stream: AWS-managed. "Shard" ≈ partition, "iterator" ≈ offset, KCL (Kinesis Client Library) ≈ consumer group. Mismo modelo conceptual, distintos nombres.

#### Dataset / recursos

- Kafka local: docker-compose con Kafka (KRaft mode, sin Zookeeper) + Kafka UI.
- Stream sintético: producer genera "click events" cada 100 ms.
- Librerías: confluent-kafka>=2.5 (más robusta) o kafka-python>=2.0 (más simple), faker para data sintética.

#### Ejercicios

1. Setup local: docker-compose con Kafka + Kafka UI. Crear topic clicks con 4 partitions. Verificar con docker exec kafka kafka-topics --list ....
2. Producer: script Python que produce 1000 mensajes con key=user_id, value={"page":"/foo","ts":...}. Verificar en Kafka UI que mensajes con mismo user_id caen en la misma partition.
3. Consumer 1 instancia: consumer.subscribe(['clicks']) + loop for msg in consumer. Procesar = print(msg.value()). Commitear offset cada 100 mensajes.
4. Consumer group, 2 instancias: levantar 2 consumers con mismo group.id. Confirmar que se reparten las 4 partitions (2-2). Matar uno, observar rebalancing — el otro toma las 4.
5. At-least-once explícito: enable.auto.commit=False, procesar mensaje, consumer.commit(). Si crash entre procesar y commit → duplicate al reiniciar.

#### Homework verificable

Sistema con:

1. Producer Python que simula 100 eventos/s durante 1 min (sintéticos con faker).
2. 3 consumers en el mismo group consumiendo de un topic con 6 partitions.
3. Cada consumer procesa y escribe a una tabla DuckDB (idempotente: PK = (user_id, event_ts)).
4. Métrica de consumer lag monitoreada (chequear kafka-consumer-groups --describe).
5. Demonstración: matar 1 consumer durante la corrida y verificar rebalancing + cero pérdida de mensajes.
6. README comparando con un equivalente en Kinesis (snippet de código sin necesidad de cuenta AWS).

Criterio de aceptación: 6000 mensajes producidos → 6000 mensajes en DuckDB (sin duplicados gracias a PK), consumer lag estabiliza <1000, rebalancing funcionó.

#### Errores comunes

| Síntoma / mensaje | Causa y cómo arreglar |
|---|---|
| Topic does not exist aunque acabás de crea | Auto-create topics está deshabilitado y cr |
| Consumer lag crece infinito | Consumer no procesa rápido suficiente. Fix |
| Mensajes duplicados al reiniciar consumer | Falta commit del último offset procesado. |
| UnknownTopicOrPartitionError intermitente | Rebalancing en curso. Fix: catch y reinten |
| Performance horrible con enable.idempotenc | Idempotente requiere ordering por key, baj |
| Producer.flush() se cuelga | Network unreachable. Fix: agregar timeout, |
| Kafka UI no muestra topics | UI conectada a broker antes que esté ready |

#### Preguntas frecuentes

¿Cuándo necesito streaming en vez de batch?

Cuando: (1) latencia importa (fraud detection: minutos), (2) volume es alto y constante (logs, telemetría), (3) data es continuamente generada (clickstream, IoT). Si tus datos llegan en cron diario y no urge: batch. La regla: batch primero, streaming cuando duela.

Kafka self-hosted o Confluent Cloud?

Self-hosted: control total, costo en EC2/operaciones. Confluent Cloud (managed): no operás brokers, pagás más. Para empezar: Confluent Cloud free tier. Para escala >100 MB/s sostenido: el TCO favorece self-hosted o Confluent Cloud Enterprise.

Kafka vs Kinesis vs Pub/Sub?

- Kafka: open-source, multi-cloud, ecosistema enorme (Kafka Connect, Kafka Streams, ksqlDB).
- Kinesis: AWS-native, integrado con Lambda/Firehose. Más limitado (shards manualmente escalados; Kinesis On-Demand mejora esto).
- Pub/Sub: GCP-native, simpler API (pull/push, no shards expuestos), gestionado al 100%.

Para multi-cloud o vendor-neutral: Kafka. Para AWS/GCP-native simple: Kinesis/Pub/Sub.

¿Cuándo necesito exactly-once?

Casi nunca. At-least-once + idempotent consumer (idempotency via PK en DB) resuelve el 95% de los casos. Exactly-once con transactional API tiene 20-30% overhead. Reservarlo para: pagos, billing.

¿Cómo manejo schemas evolucionando?

Schema Registry (Confluent, AWS Glue, Apicurio): producer registra schema Avro/Protobuf, consumer lo lee. Compatibilidad checks evita producer rompiendo consumers downstream.

¿Streaming SQL? ¿Flink, Spark Streaming, ksqlDB?

Para queries continuas (windowed aggregates, joins de streams):

- Kafka Streams: lib Java/Scala, embed en tu app.
- ksqlDB: SQL sobre Kafka topics.
- Apache Flink: streaming-first, stateful, exactly-once.
- Spark Structured Streaming: micro-batch (latencia ~seconds), reusa skills de Spark.

Para Python: PyFlink y Spark Structured Streaming son los más usados.

#### Referencias


- Narkhede, Shapira, Palino Kafka: The Definitive Guide (O'Reilly, 2ª ed., 2021).
- Kafka docs — Quickstart, Producer/Consumer configs.
- Confluent Cloud free tier.
- Reis & Housley Fundamentals of Data Engineering (O'Reilly, 2022) cap. 7.
- Awesome Kafka.


#### Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.
- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

---

## Clase 214 — Clase 214 — Formatos columnares: Parquet, Avro

> *Parte: 5 — Ingeniería de Datos · Fuente: docs Parquet 2.x + Avro spec + Reis & Housley cap. 6. Duración estimada: 70 min.*

#### Objetivo

Elegir formato de almacenamiento según el patrón de lectura: Parquet (columnar, OLAP, queries analíticas), Avro (row-based, OLTP/streaming, schema evolution), ORC (columnar, Hive ecosystem). Entender por qué Parquet es 5-100× más rápido que CSV para queries analíticas, y por qué Avro domina en Kafka.

#### Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Convertir CSV → Parquet con pandas.to_parquet / polars.write_parquet / pyarrow.
- Diferenciar row-based (CSV, JSON, Avro) de columnar (Parquet, ORC) y elegir según query pattern.
- Aprovechar column pruning + predicate pushdown + row group pruning de Parquet (Polars/DuckDB/Spark lo hacen automático).
- Aplicar compresión (snappy default, zstd mejor ratio, gzip mejor compat) y entender trade-off CPU vs tamaño.
- Definir un schema Avro y usarlo en Kafka con Schema Registry (concept).

#### Temas

| # | Tema | Por qué importa |
|---|---|---|

| | | |
|---|---|---|
| 1 | Row vs columnar | OLTP vs OLAP en el storage. |
| 2 | Parquet anatomy: file > row group > column | Lo que permite predicate pushdown. |
| 3 | Compresión: snappy, zstd, gzip, lz4 | Trade-off CPU vs tamaño. |
| 4 | Dictionary encoding, RLE | Por qué Parquet con strings repetidos pesa |
| 5 | Avro: schema-first, row-based, compact bin | Por qué domina en Kafka. |
| 6 | Schema evolution: forward/backward/full co | Cuándo se rompe; cómo manejarlo. |

#### Definiciones y características

- Row-based (CSV, JSON, Avro): cada fila es un bloque contiguo. Eficiente para SELECT * de pocas filas; ineficiente para agregados sobre 1 columna de 1M filas.
- Columnar (Parquet, ORC): cada columna es un bloque contiguo. Eficiente para SELECT col1, col2 FROM tbl WHERE col1 > X — lee solo col1 y col2.
- Row group (Parquet): conjunto de filas (default ~128 MB). Stats (min/max/null_count) por columna por row group — habilitan predicate pushdown sin abrir el row group.
- Column chunk: la parte de una columna dentro de un row group. Físicamente contigua → vectorizable.
- Page: subdivisión de column chunk (~1 MB). Unidad de compresión + read.
- Predicate pushdown: si filtrás WHERE col > 100 y el row group tiene max(col)=50, se skipea sin leer.
- Dictionary encoding: si una columna tiene ≤2^16 valores distintos, Parquet guarda un diccionario + índices. Strings repetidos: huge win.
- RLE (Run-Length Encoding): si los valores se repiten (AAAA BBBB), guarda (A,4)(B,4). Bueno para columnas con poca varianza.
- Avro: formato row-based binario con schema embedded o en Schema Registry. Diseñado para streaming + schema evolution.
- Schema evolution: agregar/quitar campos sin romper consumers viejos. Avro lo hace bien (default values, alias). Parquet también soporta pero limitado.

#### Dataset / recursos

- Dataset: NYC Taxi (CSV ~2 GB/mes) — convertir a Parquet (~150 MB/mes).
- Librerías: pyarrow>=15, polars, duckdb, fastavro (Avro).

#### Ejercicios

1. CSV → Parquet: descargá NYC Taxi 1 mes en CSV. Cargá con pandas, escribí Parquet snappy. Comparar tamaños (CSV vs Parquet) y tiempo de query (COUNT WHERE borough='Manhattan').
2. Compresión benchmark: mismo dataset, escribir con snappy, zstd, gzip, lz4. Reportar tamaño + tiempo de lectura. (Hint: zstd suele ganar en ratio; snappy en speed).
3. Predicate pushdown: en DuckDB, EXPLAIN ANALYZE SELECT  FROM parquet WHERE pickup_date='2024-01-15'. Compará "rows read" vs SELECT  FROM parquet sin WHERE.
4. Row groups y stats: con pyarrow.parquet.ParquetFile(path).metadata, inspeccioná min/max/null_count por columna por row group.
5. Avro schema + roundtrip: definí schema Avro para evento de click. Serializá 1000 eventos con fastavro, deserializá. Compará tamaño con JSON puro.

#### Homework verificable

Notebook + reporte:

1. Benchmark sobre 1 año de NYC Taxi:

- CSV (raw), Parquet snappy, Parquet zstd, Avro snappy, JSON gzipped.
- Reportar para cada uno: tamaño en disco, tiempo de query COUNT *, tiempo de SELECT col WHERE

filter.

1. Demonstración de schema evolution: tabla Avro con schema v1 (5 campos), agregar campo opcional → v2. Consumer viejo (con v1) lee data de v2 sin romperse.
2. Justificación de elección: para 3 casos de uso (analytics warehouse, Kafka stream, archive a largo plazo), recomendar formato y compresión.
3. Cálculo de costos: pasar de CSV a Parquet en S3, asumir 100 GB/mes generados. ¿$ ahorrados en S3 + transferencia?

Criterio de aceptación: el alumno demuestra con números (no opinión) por qué Parquet es ~5-50× más rápido para queries analíticas, y produce esquema Avro funcional con evolución backward-compatible.

#### Errores comunes

| Síntoma / mensaje | Causa y cómo arreglar |
|---|---|
| Parquet más grande que CSV | (1) Schema con muchas columnas vacías. (2) |
| Lectura Parquet lenta — más lenta que CSV | Estás haciendo SELECT * (no aprovecha colu |
| ArrowInvalid: Schema mismatch al concatena | Dos parquets con types ligeramente distint |
| Avro consumer rompe al evolucionar schema | Cambio no-compatible (renombrar campo sin |
| partitionBy('user_id') en Parquet crea 1M | Particionado high-cardinality. Fix: partic |
| zstd no se lee en sistema legacy | Algunos viejos solo soportan snappy/gzip. |

#### Preguntas frecuentes

¿Parquet o ORC?

Parquet: dominante fuera del ecosistema Hadoop puro. ORC: nativo del ecosistema Hive/Hortonworks (algo declining). Para 2026 greenfield: Parquet sin pensar.

¿Parquet o Delta Lake / Iceberg / Hudi?

Parquet es solo formato de archivo. Delta Lake, Iceberg, Hudi son "table formats" sobre Parquet que agregan: ACID transactions, time travel, schema evolution rica, MERGE/UPSERT. Para data lakes serios: una de estos sobre Parquet. Para análisis ad-hoc: Parquet pelado alcanza.

¿CSV alguna vez tiene sentido?

(1) Interoperabilidad humana (Excel, miras con less). (2) Tamaño chico (<1 MB) — no vale la pena la complejidad. (3) Output de un proceso para humano. Para todo lo demás: Parquet o JSON.

¿Avro o Protobuf en Kafka?

Ambos serializan binario con schema. Avro: schema embeddable o Registry, mejor evolution semantics. Protobuf: más rápido, mejor ecosystem gRPC. Confluent Kafka favorece Avro (Schema Registry built-in); gRPC users favorecen Protobuf.

¿Cuánto comprime zstd vs snappy?

Aprox: snappy reduce 50-60%, zstd reduce 65-75%. Zstd CPU cost ~2-3× snappy en compresión, ~1× en decompresión. Para almacenamiento long-term: zstd. Para alta throughput: snappy.

¿Cómo manejo Parquet con timezone-aware datetimes?

Parquet 2.x soporta timestamp(unit=us, tz='UTC'). Polars y PyArrow lo respetan. Cuidado: pandas <2.0 a veces hace conversión silenciosa. Estandarizar en UTC siempre.

#### Referencias

- Parquet docs — file format spec.
- Avro spec — schema language.
- PyArrow Parquet API.
- Apache Iceberg — table format moderno.
- CSV is dead, long live Parquet (mejor para 2026).

#### Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.
- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

---

## Clase 215 — Clase 215 — Modelado dimensional: star/snowflake schemas

> *Parte: 5 — Ingeniería de Datos · Fuente: Kimball & Ross The Data Warehouse Toolkit (Wiley, 3ª ed.) + Reis & Housley cap. 8. Duración estimada: 75 min.*

#### Objetivo

Diseñar el esquema de un data warehouse usando modelado dimensional (Kimball): una fact table central + dimension tables alrededor (star schema), o dimensiones normalizadas (snowflake schema). Es el modelo que han usado los data warehouses serios desde los 90s y sigue vigente en BigQuery/Snowflake/dbt en 2026.

#### Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Identificar fact tables (eventos medibles: sale, click, payment) vs dimension tables (entidades descriptivas: customer, product, date).
- Diseñar un star schema con fact + dimensions desnormalizadas.
- Manejar slowly changing dimensions (SCD): Tipo 1 (overwrite), Tipo 2 (history con valid_from/valid_to), Tipo 3 (current + previous).
- Crear una date dimension completa (con feriados, fiscal year, etc.) — la dimensión más reutilizada.
- Diferenciar OLAP (modelado dimensional, query analíticas) de 3NF (modelado normalizado, OLTP).

#### Temas

| # | Tema | Por qué importa |
|---|---|---|
| 1 | Star schema: fact + dims | El patrón fundamental. |
| 2 | Snowflake schema: dims normalizadas | Cuándo agregar el normalizado. |
| 3 | Surrogate keys vs natural keys | Por qué usar IDs autoincrement en DW. |
| 4 | Slowly Changing Dimensions (SCD 1/2/3) | Cómo manejar history. |
| 5 | Date dimension | La que casi siempre olvidás generar. |
| 6 | Granularidad de la fact | "Una fila por click" vs "una fila por sesi |

#### Definiciones y características

- Fact table: tabla con métricas numéricas + foreign keys a dimensions. Ej: fact_sales(date_key,

product_key, customer_key, store_key, qty, revenue). Filas chicas, MUCHAS (millones-billones).
- Dimension table: descripción de las entidades. Ej: dim_product(product_key, sku, name, category, brand). Filas grandes (muchas columnas), POCAS (miles-millones).
- Star schema: 1 fact + N dims, todas conectadas directo a la fact. Cada dim es plana (denormalizada). El default — mejor performance, joins simples.
- Snowflake schema: las dims están normalizadas (dim_product → dim_category → dim_segment). Ahorra storage en dims gigantes; complica queries.
- Surrogate key: ID autoincrement único para la dimensión (product_key=42), distinto del natural key del sistema source (sku="ABC-123"). Permite SCD Tipo 2 sin colisión.
- SCD Tipo 1 (overwrite): cambia el valor in-place. No mantiene historia. Útil para correcciones tipo "fix typo en nombre".
- SCD Tipo 2 (history): inserta nueva fila con valid_from, valid_to, is_current. La fact apunta al product_key vigente al momento del evento.
- SCD Tipo 3 (current + previous): columnas current_value + previous_value. Maneja solo 1 cambio histórico. Raro en práctica.
- Date dimension: tabla precalculada con 1 row por día (date_key, year, quarter, month, day_of_week, is_weekend, is_holiday, fiscal_year, ...). Evita derivar en cada query.
- Granularidad (grain): el nivel de detalle de la fact. "1 row per transaction" vs "1 row per day per customer". Definir el grain ES la decisión de diseño más importante.

#### Dataset / recursos

- Caso ejemplo: e-commerce con tablas operacionales orders, order_items, products, customers. Transformar a star schema.
- Librerías: DuckDB/SQL puro (las tablas son SQL, no Python).

#### Ejercicios

1. Identificar grain: dado un dataset de pedidos de e-commerce, decidí el grain de tu fact. ¿"1 row per order"? ¿"1 row per order line"? Elegí, justificá.
2. Date dimension: SQL que genera dim_date con 5 años de días, columnas year, quarter, month, day_of_week_iso, is_weekend, is_holiday_us, fiscal_year. (generate_series de Postgres/DuckDB).
3. Star schema: del e-commerce, diseñá fact_sales(date_key, product_key, customer_key, store_key, qty, revenue, discount), dim_product, dim_customer, dim_store, dim_date. SQL completo.
4. SCD Tipo 2 en dim_customer: cliente cambia de ciudad. Tu pipeline detecta el cambio → UPDATE la fila vigente con valid_to=NOW(), is_current=FALSE → INSERT nueva fila con valid_from=NOW(), is_current=TRUE. Toda venta del cliente queda asociada a su ciudad al momento de la compra.
5. Query típica: "Revenue por brand × month × is_weekend" — escribíla con JOINs entre fact + dims + dim_date. Comparala con la equivalente en tablas no-modeladas (más JOINs, más subqueries).

#### Homework verificable

Repo con:

1. SQL completo para crear: dim_date, dim_customer (SCD 2), dim_product, dim_store, fact_sales.
2. Script de carga: lee tablas operacionales (Postgres/CSV) → transforma → carga al DW (DuckDB local).
3. Demonstración de SCD 2: un cliente que cambia de ciudad; venta antes y después del cambio quedan correctamente asociadas a la ciudad de ese momento.
4. 5 queries analíticas representativas: revenue por segment × quarter, top 10 products, cohort retention, etc.
5. README explicando: grain elegido, decisiones de denormalización, qué dim usaría snowflake (si

alguna).

Criterio de aceptación: las 5 queries corren limpio; SCD 2 está correctamente implementada (verificable con un test que valida revenue agregado por ciudad-en-momento).

#### Errores comunes

| Síntoma / mensaje | Causa y cómo arreglar |
|---|---|
| Star schema con grain mixto (algunas filas | Confusión sobre el grain. Fix: definir UNA |
| Customer cambia de ciudad y todas las vent | Usaste SCD Tipo 1. Fix: SCD Tipo 2 con sur |
| Date queries lentas: WHERE EXTRACT(month F | Index no se usa con función. Fix: usar dim |
| Fact table tiene 200 columnas | Estás metiendo dimensiones como columnas. |
| Snowflake schema con 10 niveles de jerarqu | Over-engineering. Fix: empezar con star (p |
| dbt jobs explotan: full refresh diario de | Falta incremental loading. Fix: dbt increm |

#### Preguntas frecuentes

¿Modelado dimensional sigue vigente con BigQuery/Snowflake?

Sí. Aunque el compute moderno es elástico, queries siguen siendo más rápidas y baratas con star schema. dbt promueve el patrón. La diferencia con los 90s: no necesitás crear índices manualmente (los DW modernos lo manejan).

¿Star o snowflake?

Star por default. Snowflake solo si tu dim tiene jerarquía profunda + millones de rows + repetición de strings que duele en storage. Para dim_product con category repetido 10K veces: snowflake (sacar dim_category). Para customer.country repetido 1M veces: dejar en star (no vale la pena el JOIN extra).

¿Qué grain elijo?

El más fino posible que tenga sentido. "1 row per transaction line" mejor que "1 row per order" — siempre podés agregar al sumar, no podés desagregar después. Excepción: si la fact crece en TBs por día, agregar a daily_sales_by_product puede ser necesario.

¿dim_date o calcular en query?

dim_date siempre. Razones: (1) feriados, fiscal year, business days requieren lookup, no aritmética. (2) Performance: JOIN a dim_date es más rápido que EXTRACT/CASE. (3) Consistencia: un único lugar define "qué es weekend".

¿Data Vault, OBT (One Big Table), Activity Schema, Anchor model — y modelos alternativos?

- Data Vault: para enterprise con many sources, schema cambiando seguido. Más complejo.
- OBT: una tabla con todo desnormalizado (extremo de star). Bueno para una sola dashboard; explota con más casos.
- Activity Schema: 1 fact universal de "actividades" + 1 dim por entidad. Patrón moderno (Narrator).
- Anchor model: 6NF extremo, raro fuera de academia.

Para 90% de los casos: star schema sigue ganando.

¿dbt es necesario?

No estrictamente, pero es el estándar. dbt convierte SQL transforms en código (versionado, testeado, lineage). Si vas a hacer >5 modelos: usá dbt.

#### Referencias

- Kimball, R. & Ross, M. The Data Warehouse Toolkit (Wiley, 3ª ed., 2013) — la biblia del modelado dimensional.
- Reis & Housley Fundamentals of Data Engineering (O'Reilly, 2022) cap. 8.
- dbt docs — el ecosistema moderno.
- The Open Source Data Stack Conference — talks de Kimball moderno con DuckDB/dbt.
- dim_date SQL generator — dbt package listo para usar.

#### Material descargable

- Guía explicativa (PDF) — versión imprimible con todo el contenido de la clase.
- Presentación (PPTX) — deck PowerPoint listo para proyectar en clase.
- Notebook ejecutable (.ipynb) — abrilo desde el laboratorio del programa o desde Jupyter.

---

## Cierre de la parte

Fin del bundle consolidado de Parte 5 — Ingeniería de Datos · 8 clases.