

DuckDB

Data Science (CC3084) - 2026

DuckDB

Semestre 02, 2026

De Dónde Venimos

Las dos lecciones anteriores resolvieron el problema del volumen repartiendo el trabajo entre muchas máquinas.

- Hadoop distribuye bloques replicados sobre HDFS y coordina recursos con YARN.
- Spark mantiene los resultados intermedios en memoria y organiza el cómputo como un grafo dirigido acíclico.
- Ambos exigen un clúster, un administrador de recursos y un costo de operación permanente.

Queda una pregunta sin responder. Un conjunto de datos de cincuenta gigabytes en formato Parquet no necesita veinte máquinas, pero tampoco cabe cómodamente en un DataFrame de pandas. Esa franja intermedia, donde los datos superan la memoria de una laptop pero no justifican un clúster, es exactamente el territorio de DuckDB.

DuckDB

DuckDB es un sistema de gestión de bases de datos relacional, embebido y orientado al análisis, diseñado para ejecutar consultas SQL sobre grandes volúmenes de datos en una sola máquina.

Características

- Embebido: no corre como un proceso separado, sino dentro del proceso que lo usa, igual que SQLite.
- Analítico: está construido para consultas que recorren millones de filas y agregan columnas completas.
- Sin dependencias externas: se compila en cualquier sistema operativo y arquitectura, incluido el navegador mediante DuckDB-Wasm.
- Transaccional: ofrece garantías ACID mediante un control de concurrencia multiversión optimizado para cargas masivas.
- Portable: existen interfaces para Python, R, Java, C, C++, Go, Rust y Node.js.
- Libre: se distribuye bajo licencia MIT y la propiedad intelectual pertenece a la DuckDB Foundation.

Lo que significa ser embebido

No hay servidor que instalar, ni puerto que abrir, ni credenciales que administrar. La base de datos es una biblioteca que se carga en memoria dentro del programa que la invoca.

La consecuencia importante para el análisis de datos es la transferencia. Cuando el motor vive en el mismo proceso que el análisis, los datos no viajan por un socket, así que una consulta puede leer un DataFrame de Python sin copiarlo.

OLTP y OLAP

Son dos formas de usar una base de datos, con prioridades opuestas.

- OLTP, Online Transaction Processing o procesamiento de transacciones en línea. Registra la operación diaria de un negocio, donde cada instrucción toca pocas filas y la prioridad es escribir de forma segura y concurrente. Corresponde a PostgreSQL, MySQL u Oracle.
- OLAP, Online Analytical Processing o procesamiento analítico en línea. Responde preguntas sobre el histórico completo, donde cada consulta recorre millones de filas y agrega columnas enteras. Corresponde a DuckDB, BigQuery o Snowflake.

Comparación de OLTP y OLAP

La diferencia entre DuckDB y una base de datos tradicional no es de tamaño, es de propósito.

Aspecto	OLTP (PostgreSQL, MySQL)	OLAP (DuckDB)
Carga típica	Muchas operaciones pequeñas	Pocas consultas que recorren todo
Acceso	Fila completa por identificador	Pocas columnas, millones de filas
Almacenamiento	Por filas	Por columnas
Optimizado para	Escritura concurrente	Lectura y agregación

Aspecto	OLTP (PostgreSQL, MySQL)	OLAP (DuckDB)
Ejemplo de consulta	Recuperar un pedido	Promediar ventas de cinco años

Preguntar el promedio de una columna en un almacén por filas obliga a leer todas las columnas de todas las filas. En un almacén por columnas se lee únicamente esa columna.

Motor vectorizado por columnas

Un motor tradicional procesa una fila a la vez y paga el costo de interpretar la operación en cada una. DuckDB procesa vectores de aproximadamente dos mil valores de la misma columna en cada llamada.

- El costo de interpretación se reparte entre miles de valores en lugar de pagarse por fila.
- Los valores de una misma columna tienen el mismo tipo y quedan contiguos en memoria, lo que aprovecha la caché del procesador.
- El paralelismo se reparte por porciones del archivo, de modo que todos los núcleos de la máquina trabajan sobre la misma consulta.

Esta combinación explica por qué una agregación sobre cien millones de filas termina en segundos en una laptop.

Primeros Pasos

Instalación

```
# línea de comandos
brew install duckdb

# Python
pip install duckdb
```

La versión estable de la serie 1.5 lleva el nombre Variegata, y la serie 1.4 es la de soporte extendido. La versión 2.0, prevista para finales de 2026, introduce entrada y salida asíncrona para Parquet y CSV.

Modos de trabajo

```
import duckdb

# base en memoria, desaparece al terminar el proceso
con = duckdb.connect()

# base persistente en un solo archivo
con = duckdb.connect("ventas.duckdb")
```

La base en memoria sirve para exploración y para usar DuckDB como motor de consulta sobre archivos externos. La base persistente guarda tablas, vistas e índices en un único archivo con formato columnar comprimido.

SQL directamente sobre archivos

```
SELECT * FROM 'ventas.csv';
SELECT * FROM 'ventas.parquet';
SELECT * FROM read_csv('ventas.csv', header = true);
```

No hace falta declarar el esquema ni ejecutar una carga previa. DuckDB reconoce la extensión, infiere los tipos y trata el archivo como una tabla.

Grandes Volúmenes de Datos

Aquí está el aporte real de DuckDB al curso. Las secciones siguientes describen los mecanismos que permiten consultar conjuntos de datos mucho más grandes que la memoria disponible.

Consultar sin cargar

```
CREATE VIEW ventas AS  
SELECT * FROM read_parquet('datos/ventas.parquet');
```

La vista no copia nada. Cada consulta contra `ventas` lee el archivo en el momento, así que un archivo de doscientos gigabytes se consulta sin ocupar doscientos gigabytes de disco adicionales.

Proyección y filtro empujados

Dos optimizaciones automáticas determinan cuánto se lee realmente del disco.

- Proyección empujada: de un archivo con ciento veinte columnas, una consulta que pide tres lee solamente esas tres.
- Filtro empujado: Parquet guarda el valor mínimo y máximo de cada grupo de filas en un mapa de zonas, y DuckDB descarta grupos completos sin descomprimirlos.

```
SELECT region, SUM(monto)  
FROM 'ventas.parquet'  
WHERE anio = 2025  
GROUP BY region;
```

De un archivo de cincuenta columnas y mil millones de filas, esta consulta puede terminar leyendo menos del dos por ciento de los bytes.

Varios archivos como una sola tabla

```
-- patrón glob
SELECT COUNT(*) FROM 'datos/ventas/*.parquet';

-- lista explícita
SELECT * FROM read_parquet(['enero.parquet', 'febrero.parquet']);

-- de qué archivo vino cada fila
SELECT filename, COUNT(*) FROM 'datos/ventas/*.parquet' GROUP BY filename;
```

Un directorio con mil archivos se consulta como una tabla única. La columna virtual `filename` indica el origen de cada fila y está disponible sin configuración desde la versión 1.3.

Particionado tipo Hive

El particionado tipo Hive codifica los valores de una columna en los nombres de los directorios, con el formato `clave=valor`.

```
ventas/
├─ anio=2024/
│   ├─ mes=1/datos.parquet
│   └─ mes=2/datos.parquet
└─ anio=2025/
    ├─ mes=1/datos.parquet
    └─ mes=2/datos.parquet
```

```
SELECT SUM(monto)
FROM read_parquet('ventas/**/*.*parquet', hive_partitioning = true)
WHERE anio = 2025 AND mes = 1;
```

`anio` y `mes` no existen dentro de los archivos, se deducen de la ruta. DuckDB aplica el filtro sobre esas columnas antes de abrir nada, así que de los cuatro

archivos del ejemplo abre uno. Con trescientos sesenta y cinco particiones diarias, la diferencia es leer un archivo en lugar de trescientos sesenta y cinco.

Escribir datos particionados

```
COPY ventas TO 'salida' (FORMAT parquet, PARTITION_BY (anio, mes));
```

`PARTITION_BY` no admite expresiones, así que una partición derivada de una fecha requiere crear antes la columna correspondiente.

Procesamiento mayor que la memoria

Una consulta que agrupa mil millones de filas construye una tabla de hash que puede no caber en memoria. En pandas eso termina en un error de memoria y el trabajo se pierde. DuckDB escribe el exceso a un archivo temporal y continúa.

El administrador de búferes trabaja por bloques y usa particionamiento por radix para decidir qué fragmentos del estado intermedio van a disco. El rendimiento se degrada de forma gradual en lugar de colapsar.

Operador	Vuelca a disco	Ejemplo
Agrupación	Sí	<code>GROUP BY cliente</code>
Unión	Sí	<code>JOIN</code> por llave hash
Orden	Sí	<code>ORDER BY fecha</code>
Ventana	Sí	<code>OVER (PARTITION BY region)</code>

Configuración de memoria

```
SET memory_limit = '8GB';  
SET temp_directory = '/ruta/disco/rapido/tmp';  
SET threads = 8;  
SET preserve_insertion_order = false;
```

- `memory_limit` define el tope de memoria y por omisión vale el ochenta por ciento de la memoria física.
- `temp_directory` decide dónde se escribe el volcado, y conviene apuntarlo a un disco de estado sólido local.
- `threads` limita los hilos, útil cuando demasiada concurrencia degrada el rendimiento.
- `preserve_insertion_order` permite reordenar el resultado cuando no hay `ORDER BY`, lo que reduce el consumo de memoria al importar y exportar.

Límites del volcado a disco

El volcado a disco no es ilimitado y conviene conocer sus fronteras.

- Las funciones de agregación que acumulan colecciones, como `list()` y `string_agg()`, mantienen todo su estado en memoria.
- Las agregaciones que requieren orden necesitan el conjunto completo antes de empezar.
- Una consulta con varios operadores bloqueantes simultáneos todavía puede agotar la memoria.

La medida preventiva es la misma en todos los casos, que consiste en pedir menos columnas, filtrar lo antes posible y particionar los datos de forma razonable.

Datos remotos

```
INSTALL httpfs;
LOAD httpfs;

SELECT COUNT(*) FROM read_parquet('s3://bucket/ventas/*.parquet');
SELECT * FROM read_parquet('https://servidor.org/datos.parquet') LIMIT 10;
```

La extensión `httpfs` permite consultar archivos en S3 o en cualquier servidor HTTP. Las optimizaciones de proyección y filtro siguen aplicando, de modo que se descargan únicamente los fragmentos necesarios en lugar del archivo completo.

DuckDB y Python

Sustitución de referencias

DuckDB detecta los nombres que no existen en su catálogo y los busca entre las variables del programa. El mecanismo se llama sustitución de referencias, o replacement scan.

```
import duckdb
import pandas as pd

ventas = pd.read_csv("ventas.csv")

resultado = duckdb.sql("""
    SELECT region, SUM(monto) AS total
    FROM ventas
    GROUP BY region
    ORDER BY total DESC
""").df()
```

`ventas` es un DataFrame, no una tabla registrada, y la consulta lo lee sin copiarlo. Esto convierte a DuckDB en un reemplazo directo de las operaciones de agrupación y unión de pandas cuando el volumen crece.

Devolver resultados

Método	Devuelve
<code>.df()</code>	DataFrame de pandas
<code>.arrow()</code>	Tabla de Apache Arrow
<code>.pl()</code>	DataFrame de Polars
<code>.fetchall()</code>	Lista de tuplas de Python

El camino inverso también funciona, porque el resultado de una consulta se puede escribir directamente a Parquet o CSV con `COPY`.

DuckDB Frente a Otras Herramientas

Comparación

Aspecto	pandas	DuckDB	Spark
Modelo	En memoria	Mayor que la memoria	Distribuido
Infraestructura	Ninguna	Ninguna	Clúster
Lenguaje	API de Python	SQL	SQL y varias API
Uniones de cien millones de filas	Suele fallar	Resuelve en disco	Resuelve en el clúster

Aspecto	pandas	DuckDB	Spark
Costo de operación	Nulo	Nulo	Alto

La escala decide la herramienta

Volumen	Herramienta razonable
Menos de un gigabyte	pandas, por comodidad
Un gigabyte a un terabyte	DuckDB en una máquina
Decenas de terabytes o más	Spark en un clúster

Las mediciones publicadas ubican a DuckDB entre cinco y diez veces más rápido que pandas en agrupaciones sobre más de cien millones de filas. Spark en modo local pierde frente a DuckDB, porque paga la coordinación de un clúster sin tener uno.

El error habitual en la industria consiste en levantar un clúster para un problema de cincuenta gigabytes. La regla práctica es empezar por la máquina completa y recurrir al clúster solo cuando esa máquina ya no alcanza.

Cuándo no usar DuckDB

- Cargas transaccionales con escritura concurrente intensa, que corresponden a una base OLTP.
- Un solo archivo de base de datos admite un proceso escritor a la vez.
- Volúmenes de decenas de terabytes con crecimiento sostenido, donde el clúster sí se justifica.

- Servicio de consultas a muchos usuarios simultáneos, que requiere una arquitectura cliente-servidor.

Errores Comunes

- Cargar todo a una tabla antes de consultar, cuando la vista sobre Parquet basta y evita duplicar el almacenamiento.
- Filtrar después de leer, que obliga a abrir las trescientas sesenta y cinco particiones para conservar una.
- Particionar en exceso, porque miles de archivos diminutos degradan la lectura más de lo que ayuda la poda.
- Dejar el directorio temporal en un disco lento, lo que convierte el volcado en el cuello de botella.
- Usar CSV para datos grandes, ya que sin mapas de zonas ni compresión por columna desaparecen las optimizaciones.
- Esperar que `list()` y `string_agg()` vuelquen a disco, cuando su estado permanece completo en memoria.

Resumen

- DuckDB es una base de datos analítica embebida que corre dentro del proceso que la usa, sin servidor ni administración.
- Su almacenamiento por columnas y su ejecución vectorizada están pensados para recorrer millones de filas y agregar columnas completas.
- Consulta archivos Parquet y CSV en su lugar, sin carga previa, mediante vistas y patrones glob.

- La proyección y el filtro empujados reducen la lectura a las columnas y los grupos de filas que la consulta necesita.
- El particionado tipo Hive traslada el filtro al sistema de archivos, así que las particiones irrelevantes nunca se abren.
- La agrupación, la unión, el orden y las funciones de ventana vuelcan a disco cuando el estado intermedio excede la memoria.
- `memory_limit`, `temp_directory` y `threads` gobiernan ese comportamiento, con un tope por omisión del ochenta por ciento de la memoria física.
- Consulta DataFrames de pandas sin copiarlos y devuelve resultados a pandas, Arrow o Polars.
- En la franja de un gigabyte a un terabyte supera a pandas y evita el costo de un clúster de Spark.

Bibliografía

- Why DuckDB. https://duckdb.org/why_duckdb
- Tuning Workloads. https://duckdb.org/docs/stable/guides/performance/how_to_tune_workloads
- Reading Parquet Files. <https://duckdb.org/docs/stable/data/parquet/overview>
- Hive Partitioning. https://duckdb.org/docs/stable/data/partitioning/hive_partitioning
- SQL on Pandas. https://duckdb.org/docs/stable/guides/python/sql_on_pandas
- Larger-than-Memory Processing. <https://motherduck.com/glossary/larger-than-memory-processing/>