Clase 045 — SQL desde Python: sqlite3, SQLAlchemy, DuckDB

Parte: 0 — Prerrequisitos · Fuente: Python stdlib sqlite3 · SQLAlchemy docs · DuckDB Python docs. · Duración estimada: 75 min.

Clase 045 — SQL desde Python: sqlite3, SQLAlchemy, DuckDB

> *Parte: 0 — Prerrequisitos · Fuente: Python stdlib sqlite3 · SQLAlchemy docs · DuckDB Python docs. · Duración estimada: 75 min.*

Objetivo

Que el alumno conecte Python con SQL de las 3 formas que va a encontrar en producción: sqlite3 (stdlib, demo local), SQLAlchemy (ORM/engine genérico para PostgreSQL/MySQL), y DuckDB (columnar embebido para análisis sobre CSV/Parquet sin servidor).

Resultados de aprendizaje

Al finalizar la clase, el alumno podrá:

1. Conectar y consultar con sqlite3 stdlib, usando placeholders ? (NUNCA concatenar SQL).
2. Usar SQLAlchemy create_engine(URL) + pd.read_sql para queries a cualquier RDBMS.
3. Usar DuckDB para hacer SQL sobre DataFrames y CSV/Parquet directamente.
4. Prevenir SQL injection con queries parametrizadas.
5. Decidir entre sqlite/SQLAlchemy/DuckDB según el caso.

Temas

#	Tema	Por qué importa
1	sqlite3 stdlib: connect, cursor, fetchall	Para demos y BDs ligeras.
2	Placeholders ? y :nombre	NUNCA concatenar strings.
3	SQLAlchemy create_engine('postgresql://...	Soporta todos los RDBMS.
4	pd.read_sql y df.to_sql	Pasarela pandas ↔ BD.
5	DuckDB: SQL sobre DataFrames y archivos	duckdb.query('SELECT ... FROM df').
6	Cuándo cada uno	Trade-offs.

Definiciones y características

sqlite3 (stdlib)

: Driver Python para SQLite. Sin dependencias externas. Patrón: connect(...) → cursor() → execute(sql, params) → fetchall().

Parameterized query (? o :name)

: Placeholder donde el driver substituye valores escapados. Única forma segura de pasar datos de usuario — previene SQL injection.

SQLAlchemy

: Toolkit ORM + Core para Python. Backend-agnostic: cambias el URL del engine y migras entre SQLite/Postgres/MySQL sin tocar queries. create_engine('postgresql://...').

DuckDB

: OLAP DB embebida. SQL sobre DataFrames pandas (duckdb.query('SELECT ... FROM df')) y archivos CSV/Parquet directos (FROM 'data.csv'). Mucho más rápido que sqlite3 para analytics.

SQL injection

: Inyección de SQL malicioso via concatenación de strings con input de usuario. Prevención: SIEMPRE parameterized queries, nunca f-string con valores externos.

pd.read_sql / df.to_sql

: Pasarela pandas↔BD. Acepta connection o engine. read_sql_query para queries complejas; read_sql_table para tablas completas.

Dataset / recursos

Penguins descargado a CSV local para DuckDB; datos sintéticos para sqlite/SQLAlchemy.

Ejercicios

1. sqlite3 con placeholders. Crea tabla, inserta 5 filas usando executemany con tuples, consulta con ? placeholder. Demuestra el bug si concatenas.

2. df.to_sql y pd.read_sql. Carga un DataFrame a SQLite y consulta de vuelta.

3. SQLAlchemy engine. Crea engine SQLite. Usa pd.read_sql con engine.

4. DuckDB sobre DataFrame. Carga penguins en df. duckdb.query('SELECT species, AVG(body_mass_g) FROM df GROUP BY species').df().

5. DuckDB sobre CSV. Mismo query pero FROM 'penguins.csv' directo, sin cargar a pandas.

Homework verificable

Notebook con 3 backends del mismo análisis: (a) sqlite3 stdlib + cursor; (b) SQLAlchemy engine + pd.read_sql; (c) DuckDB sobre CSV. Documenta cuándo elegirías cada uno. Demuestra explícitamente el peligro de SQL injection con concatenación vs placeholders.

Criterio de aceptación: Las 3 versiones devuelven el mismo resultado. Demo de injection sin daño real.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
OperationalError: no such table: X después	Falta con.commit(). sqlite3 no auto-commit
Concatené input de usuario en query y func	Hasta que el usuario malicioso prueba '; D
SQLAlchemy 2.0 — Engine.execute no existe	API cambió: ahora with engine.connect() as
DuckDB lee CSV pero pierde tipos	Pandas adivina mejor. Fix: duckdb.read_csv
pd.read_sql lento con N grande	Driver Python carga todo a Python. Fix: ch

Preguntas frecuentes

¿sqlite3, SQLAlchemy o DuckDB?

sqlite3 para demos/tests/scripts locales. SQLAlchemy para producción con Postgres/MySQL. DuckDB para EDA sobre CSV/Parquet sin servidor — el más rápido para analytics.

¿pd.read_sql es seguro contra injection?

Sí si pasas params: read_sql('SELECT * FROM t WHERE x=:val', con, params={'val': user_input}). No si concatenas strings.

¿ORM (SQLAlchemy declarative) o queries directas?

ORM cuando el modelo se usa en muchas partes (web app con N modelos). Queries directas para análisis ad-hoc. Pueden coexistir.

¿DuckDB sobre Parquet vs sobre pandas?

Parquet directo es más rápido (no carga a Python). Pandas cuando ya tienes el DataFrame en memoria. DuckDB es smart: optimiza ambos casos.

¿Cerrar conexión manualmente?

Usa context manager: with sqlite3.connect(...) as con: ... o con.close() en finally. Conexiones dejadas abiertas consumen handles del OS.

Referencias

- Python sqlite3 docs
- SQLAlchemy tutorial
- DuckDB Python API
- OWASP — SQL injection prevention

Siguiente clase

Clase 046 — NoSQL: MongoDB con pymongo

Apéndice: notebook (primer bloque)

Primera celda ejecutable del notebook de la clase.

```
import sqlite3
import pandas as pd
import numpy as np
rng = np.random.default_rng(42)

# DataFrame demo
df = pd.DataFrame({
    'cliente_id': range(1, 11),
    'nombre': [f'Cliente {i}' for i in range(1, 11)],
    'pais': rng.choice(['ES', 'CL', 'MX'], 10),
    'monto': rng.uniform(50, 500, 10).round(2),
})
print(df.head())
```

Archivos complementarios

- notebook.ipynb