Clase 043 — SQL fundamental: SELECT, WHERE, JOIN, GROUP BY, HAVING

Parte: 0 — Prerrequisitos · Fuente: SQL for Data Scientists (Tanimura) caps. 1-3 · SQLite docs · DuckDB docs. · Duración estimada: 120 min.

Clase 043 — SQL fundamental: SELECT, WHERE, JOIN, GROUP BY, HAVING

> *Parte: 0 — Prerrequisitos · Fuente: SQL for Data Scientists (Tanimura) caps. 1-3 · SQLite docs · DuckDB docs. · Duración estimada: 120 min.*

Objetivo

Que el alumno escriba consultas SQL no triviales — SELECT con filtros, JOINs (inner/left), agregaciones con GROUP BY y filtros sobre agregados con HAVING. Y entienda el orden de ejecución lógico (FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT), que es lo que confunde a todo el mundo al principio.

Resultados de aprendizaje

Al finalizar la clase, el alumno podrá:

1. Escribir SELECT con filtros WHERE y operadores (=, <>, IN, BETWEEN, LIKE, IS NULL).
2. Hacer JOIN (INNER, LEFT, RIGHT, FULL) y reconocer cuándo cada uno.
3. Agrupar y agregar con GROUP BY + COUNT, SUM, AVG, MAX, MIN.
4. Filtrar agregados con HAVING (no se puede con WHERE).
5. Recitar el orden lógico: FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT.

Temas

#	Tema	Por qué importa
1	SELECT, FROM, WHERE	Lo básico, sin trampa.
2	Operadores WHERE	=, <>, IN, BETWEEN, LIKE, IS NULL.
3	JOINs (inner/left/right/full)	Análogos a pandas merge.
4	GROUP BY + agregadas	COUNT/SUM/AVG/MAX/MIN.
5	HAVING vs WHERE	HAVING filtra después de GROUP BY.
6	ORDER BY, LIMIT, OFFSET	Final del pipeline.
7	Orden lógico ≠ orden escrito	El gran malentendido.

Definiciones y características

SQL (Structured Query Language)

: Lenguaje declarativo para bases de datos relacionales. Describes qué quieres (no cómo) y el motor lo ejecuta. Estandarizado pero con dialectos (SQLite, PostgreSQL, MySQL, BigQuery).

Orden lógico vs escrito

: Escribes: SELECT-FROM-WHERE-GROUP-HAVING-ORDER. Ejecuta: FROM-WHERE-GROUP-HAVING-SELECT-ORDER-LIMIT. Por eso WHERE SUM(...) falla (aún no

agrupado) — usa HAVING.

JOIN

: Combina filas de 2+ tablas por una key. Tipos: INNER (intersección), LEFT (todo left + match right), RIGHT (espejo), FULL OUTER (todo unión), CROSS (producto cartesiano).

GROUP BY + HAVING

: GROUP BY: agrupa filas por valor(es). HAVING: filtra los grupos después de agregar (no se puede con WHERE).

Funciones de agregación

: Operan sobre grupos: COUNT(*), COUNT(DISTINCT x), SUM, AVG, MIN, MAX, STDDEV. Devuelven UN valor por grupo.

DuckDB

: Motor SQL embebido (como SQLite) pero columnar y optimizado para analytics. Lee CSV/Parquet directo (FROM 'file.csv'). Drop-in para queries analíticas, mucho más rápido que SQLite en agregados.

Dataset / recursos

SQLite en memoria con 2 tablas sintéticas: clientes (10 filas) y ordenes (30 filas). Generado en el notebook con sqlite3 stdlib. Sin descarga.

Ejercicios

1. SELECT básico. Lista de clientes con país = 'ES'.

2. JOIN. Cada orden con el nombre del cliente.

3. LEFT JOIN. Todos los clientes, sumando órdenes (NaN si no tienen).

4. GROUP BY + HAVING. Clientes con más de 3 órdenes y monto total > 200.

5. Orden lógico. Explica con tus palabras por qué WHERE total > 100 no funciona si total es SUM(monto) — necesitas HAVING.

Homework verificable

Notebook con SQLite en memoria: (a) crea 2 tablas y carga datos sintéticos; (b) 5 consultas progresivas (filter, join, group, having, top-N); (c) explica el orden lógico con un ejemplo; (d) mismo ejercicio con DuckDB (sustituye sqlite3.connect(':memory:')).

Criterio de aceptación: Las 5 consultas producen el resultado esperado; DuckDB devuelve igual.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
column "x" must appear in GROUP BY clause	Seleccionaste col no agregada ni en GROUP
WHERE SUM(monto) > 100 falla	WHERE corre ANTES de GROUP. Fix: usa HAVIN
INNER JOIN pierde filas que esperaba ver	La key no matchea (NULL, tipos, espacios).

COUNT(col) devuelve menos que COUNT(*)	COUNT(col) ignora NULL en esa columna. Fix
SELECT * después de JOIN trae cols duplica	Ambas tablas tienen id. Fix: aliasea: SELE

Preguntas frecuentes

¿COUNT(*) o COUNT(1)?

Equivalentes en motores modernos (parser optimiza). COUNT(*) es más legible — úsalo.

¿Cuándo DISTINCT?

Cuando hay filas duplicadas que no deberían contarse. Cuidado: en SELECT con muchas cols puede ser caro. Mejor agrupa con GROUP BY si vas a agregar después.

¿UNION o UNION ALL?

UNION quita duplicados (más caro). UNION ALL mantiene todo (más rápido). Usa ALL si sabes que no hay duplicados (más común).

¿SQLite o PostgreSQL para aprender?

SQLite para arrancar (sin servidor, 1 archivo). El SQL es 90% igual. Migras a PostgreSQL cuando necesites: tipos avanzados, concurrencia, escala, JSON nativo.

¿Cómo trato fechas en SQL?

Cada motor su dialecto. SQLite: strings ISO '2024-01-15' + date(), strftime(). PostgreSQL: tipo DATE/TIMESTAMP nativo. Estándar: DATE '2024-01-15'.

Referencias

- Tanimura, SQL for Data Scientists, caps. 1-3.
- SQLite SELECT docs
- DuckDB docs

Siguiente clase

Clase 044 — SQL avanzado: CTEs, window functions, subqueries correlacionadas

Apéndice: notebook (primer bloque)

Primera celda ejecutable del notebook de la clase.

```
import sqlite3
import pandas as pd

con = sqlite3.connect(':memory:')
con.executescript('''
CREATE TABLE clientes (
    cliente_id INTEGER PRIMARY KEY,
    nombre TEXT NOT NULL,
    pais TEXT,
    plan TEXT
```

```
);
CREATE TABLE ordenes (
    orden_id INTEGER PRIMARY KEY,
    cliente_id INTEGER REFERENCES clientes(cliente_id),
    fecha DATE,
    monto REAL
);

INSERT INTO clientes (nombre, pais, plan) VALUES
    ('Ana',  'ES', 'pro'),
    ('Bob',  'ES', 'free'),
    ('Cris', 'CL', 'pro'),
    ('Dan',  'MX', 'free'),
    ('Eli',  'ES', 'pro');

INSERT INTO ordenes (cliente_id, fecha, monto) VALUES
    (1, '2024-01-15', 120),
    (1, '2024-02-20',  80),
    (1, '2024-03-12', 150),
    (1, '2024-04-05',  60),
# ... (truncado)
```

Archivos complementarios

- notebook.ipynb